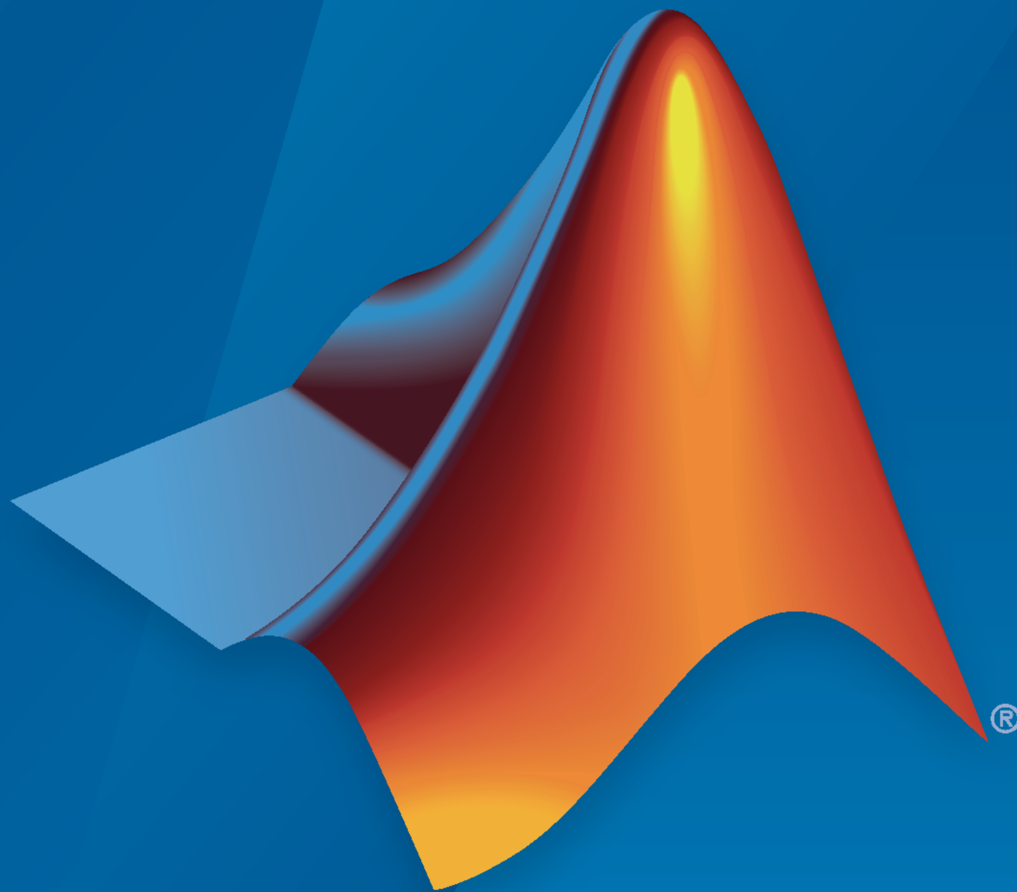
Data Acquisition Toolbox™

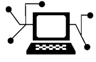
User's Guide



MATLAB® & SIMULINK®

R2023a

How to Contact MathWorks



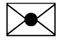
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Data Acquisition Toolbox™ User's Guide

© COPYRIGHT 2005–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

May 1999	First printing	New for Version 1
November 2000	Second printing	Revised for Version 2 (Release 12)
June 2001	Third printing	Revised for Version 2.1 (Release 12.1)
July 2002	Online only	Revised for Version 2.2 (Release 13)
June 2004	Online only	Revised for Version 2.5 (Release 14)
October 2004	Online only	Revised for Version 2.5.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.6 (Release 14SP2)
September 2005	Online only	Revised for Version 2.7 (Release 14SP3)
October 2005	Reprint	Version 2.1 (Notice updated)
November 2005	Online only	Revised for Version 2.8 (Release 14SP3+)
March 2006	Fourth printing	Revised for Version 2.8.1 (Release 2006a)
September 2006	Online only	Revised for Version 2.9 (Release 2006b)
March 2007	Online only	Revised for Version 2.10 (Release 2007a)
May 2007	Fifth printing	Minor revision for Version 2.10
September 2007	Online only	Revised for Version 2.11 (Release 2007b)
March 2008	Online only	Revised for Version 2.12 (Release 2008a)
October 2008	Online only	Revised for Version 2.13 (Release 2008b)
March 2009	Online only	Revised for Version 2.14 (Release 2009a)
September 2009	Online only	Revised for Version 2.15 (Release 2009b)
March 2010	Online only	Revised for Version 2.16 (Release 2010a)
September 2010	Online only	Revised for Version 2.17 (Release 2010b)
April 2011	Online only	Revised for Version 2.18 (Release 2011a)
September 2011	Online only	Revised for Version 3.0 (Release 2011b)
March 2012	Online only	Revised for Version 3.1 (Release 2012a)
September 2012	Online only	Revised for Version 3.2 (Release 2012b)
March 2013	Online only	Revised for Version 3.3 (Release 2013a)
September 2013	Online only	Revised for Version 3.4 (Release 2013b)
March 2014	Online only	Revised for Version 3.5 (Release 2014a)
October 2014	Online Only	Revised Version 3.6 (Release 2014b)
March 2015	Online only	Revised for Version 3.7 (R2015a)
September 2015	Online only	Revised for Version 3.8 (Release 2015b)
March 2016	Online only	Revised for Version 3.9 (Release 2016a)
September 2016	Online only	Revised for Version 3.10 (Release 2016b)
March 2017	Online only	Revised for Version 3.11 (Release 2017a)
September 2017	Online only	Revised for Version 3.12 (Release 2017b)
March 2018	Online only	Revised for Version 3.13 (Release 2018a)
September 2018	Online only	Revised for Version 3.14 (Release 2018b)
March 2019	Online only	Revised for Version 4.0 (Release 2019a)
September 2019	Online only	Revised for Version 4.0.1 (Release 2019b)
March 2020	Online only	Revised for Version 4.1 (Release 2020a)
September 2020	Online only	Revised for Version 4.2 (Release 2020b)
March 2021	Online only	Revised for Version 4.3 (Release 2021a)
September 2021	Online only	Revised for Version 4.4 (Release 2021b)
March 2022	Online only	Revised for Version 4.5 (Release 2022a)
September 2022	Online only	Revised for Version 4.6 (Release 2022b)
March 2023	Online only	Revised for Version 4.7 (Release 2023a)

Contents

1 Introduction to Data Acquisition

Data Acquisition Toolbox Product Description **1-2**

Product Capabilities **1-3**
Understanding Data Acquisition Toolbox **1-3**
Supported Hardware **1-3**

Anatomy of a Data Acquisition Experiment **1-4**
System Setup **1-4**
Calibration **1-4**
Trials **1-4**

Data Acquisition System **1-5**
Overview **1-5**
Data Acquisition Hardware **1-6**
Sensors **1-7**
Signal Conditioning **1-9**
The Computer **1-11**
Software **1-11**

Analog Input Subsystem **1-13**
Function of the Analog Input Subsystem **1-13**
Sampling **1-13**
Quantization **1-15**
Channel Configuration **1-18**
Transferring Data from Hardware to System Memory **1-20**

Making Quality Measurements **1-22**
What Do You Measure? **1-22**
Accuracy and Precision **1-22**
Noise **1-25**
Matching the Sensor Range and A/D Converter Range **1-25**
How Fast Should a Signal Be Sampled? **1-26**

Selected Bibliography **1-29**

2 Using Data Acquisition Toolbox Software

Installation Information **2-2**
Prerequisites **2-2**
Toolbox Installation **2-2**

Hardware and Driver Installation 2-2

Access Your Hardware 2-3
Connect to Your Hardware 2-3
Examine Your Hardware Resources 2-3
Acquire Audio Data 2-4
Generate Audio Data 2-4
Acquire and Generate Digital Data 2-5

3 Introduction to the DataAcquisition Interface

The DataAcquisition Object 3-2

Get Command-Line Help 3-3

4 Using the DataAcquisition Interface

Interface Workflow 4-2
Working a DataAcquisition 4-2
DataAcquisition Interface and Data Acquisition Toolbox 4-2

Digital Input and Output 4-3

Discover Hardware Devices 4-4

Create a DataAcquisition Interface 4-5

Channel Properties 4-7
Get Property Information 4-7
All Channels 4-7
Analog Input and Output Channels 4-8
Other Analog Measurements 4-9
Digital Channels 4-13
Counter Channels 4-14
Audio Channels 4-16
Function Generator Channels 4-16

5 Support Package Installer

Install Hardware Support Package for Vendor Support 5-2
Install Support Packages 5-2
Update or Uninstall Support Packages 5-2

6 Analog Input and Output

Acquire Data in the Foreground **6-2**

Acquire Data from Multiple Channels **6-3**

Acquire Data in the Background with Live Plot Updates **6-4**

Acquire Bridge Measurements **6-5**

Acquire Sound Pressure Data **6-7**

Acquire IEPE Data **6-9**

Generate Signals in the Foreground **6-11**

Generate Signals on Multiple Channels **6-12**

Generate Signals in the Background **6-13**

Generate Signals in the Background Continuously **6-14**

Acquire Data and Generate Signals Simultaneously **6-16**

Acquire Data with Analog Input Recorder **6-17**

Generate Signals with Analog Output Generator **6-21**

7 Analog Devices Active Learning Module

Analog Devices ADALM1000 Hardware **7-2**

Generate and Measure Signals with Analog Devices ADALM1000 . . . . . . **7-3**
- Updated Function Syntax **7-3**
- Source Voltage and Measure Current **7-3**
- Generate a Pulse **7-4**
- Generate Waveforms **7-5**

8 Counter Input and Output

Analog and Digital Counters **8-2**

Acquire Counter Input Data **8-3**
- Add Counter Input Channel **8-3**

Acquire a Single Count **8-3**
Acquire a Single Frequency Count **8-4**
Acquire Counter Input Data in the Foreground **8-4**

Generate Pulse Data on a Counter Channel **8-6**
Add Counter Output Channels **8-6**
Generate Pulses on a Counter Output Channel **8-6**

9 Digital Operations

Digital Channels **9-2**
Digital Clocked Operations **9-2**
Access Digital Subsystem Information **9-2**

Acquire Non-Clocked Digital Data **9-4**

Acquire Digital Data Using a Shared Clock **9-5**

Acquire Digital Data Using an External Clock **9-6**

Acquire Digital Data Using a Counter Output Channel as External Clock **9-8**
Generate a Clock Using a Counter Output Channel **9-8**
Use Counter Clock to Acquire Clocked Digital Data **9-9**

Acquire Digital Data Using an External Clock via Chassis PFI Terminal **9-11**

Acquire Digital Data in Hexadecimal Values **9-12**

Generate Non-Clocked Digital Data **9-13**

Generate Digital Output Using Decimal Data Across Multiple Lines . . . **9-14**

Generate and Acquire Data on Bidirectional Channels **9-15**

Generate Signals on Both Analog and Digital Channels **9-16**

10 Multichannel Audio

Audio Input and Output **10-2**
Multichannel Audio Scan Rate **10-2**
Audio Measurement Range **10-2**
Acquire Audio Data **10-2**

11 Waveform Function Generation

Digilent Analog Discovery Devices **11-2**

Digilent Function Waveform Generator Channels **11-3**

Waveform Types **11-5**

Generate a Standard Waveform Using Function Waveform Generator Channels **11-8**

12 Triggers and Clocks

Trigger Connections **12-2**
When to Use Triggers **12-2**
External Triggering **12-2**

Acquire Voltage Data Using a Digital Trigger **12-4**

Clock Connections **12-5**
When to Use Clocks **12-5**
Import Scan Clock from External Source **12-5**
Export Scan Clock to External System **12-5**

13 Synchronization

Synchronization **13-2**
Shared Triggers and Shared Scan Clocks **13-2**
Source and Destination Devices **13-3**
Automatic Synchronization **13-4**
Synchronization Scenarios **13-4**

Multiple-Device Synchronization Using USB or PXI Devices **13-7**
Acquire Synchronized Data Using USB Devices **13-7**
Synchronize Counter Outputs from Multiple Devices **13-8**
Synchronize DSA PXI Devices Using AutoSyncDSA **13-8**
Acquire Synchronized Data Using PXI Devices **13-9**

Synchronize with PFI on CompactDAQ Chassis Without Terminals . . . **13-11**

Multiple-Chassis Synchronization with CompactDAQ Devices **13-12**

Synchronize DSA Devices **13-13**
PXI DSA Devices **13-13**
Hardware Restrictions **13-13**

PCI DSA Devices 13-14
Synchronize DSA PCI Devices 13-14
Handle Filter Delays with DSA Devices 13-15

14 Transition Your Code to New Interfaces

Transition Your Code from Session to DataAcquisition Interface 14-2
Transition Common Workflow Commands 14-2
Acquire Analog Data 14-3
Use Triggers 14-4
Initiate an Operation When Number of Scans Exceeds Specified Value 14-5
Analog Output Generator Code 14-6

15 Functions

16 Apps

17 Blocks

A Troubleshooting Your Hardware

Troubleshooting Tips A-2
Find Devices and Create a DataAcquisition Interface A-2
Is My Device Driver Supported? A-3
Cannot Find Hardware Vendor A-3
Cannot Detect My Device A-4
Why Doesn’t My NI Hardware Work? A-5
Why Was My DataAcquisition Object Deleted? A-5
What Is a Reserved Hardware Error? A-5
Network Device Appears Unsupported A-5
Cannot Add Channels A-6
ADC Overrun Error with External Clock A-6
Cannot Add Clock Connection to PXI Devices A-6
Cannot Complete Long Foreground Acquisition A-6

Cannot Use PXI 4461 and 4462 Together **A-6**
Cannot Get Correct Scan Rate with Digilent Devices **A-7**
Cannot Simultaneously Acquire and Generate with myDAQ Devices **A-7**
Simultaneous Analog Input and Output Not Synchronized Correctly **A-7**
Counter Single Scan Returns NaN **A-7**
External Clock Will Not Trigger Scan **A-7**
Why Does My S/PDIF Device Time Out? **A-7**
MOTU Device Not Working Correctly **A-7**

Contact MathWorks for Technical Support **A-9**

Set Up Your System for Device Detection **A-10**
NI Devices **A-10**
DirectSound Audio Devices **A-10**
Measurement Computing (MCC) Devices **A-11**
Digilent Analog Discovery Devices **A-11**
Analog Devices ADALM1000 Devices **A-12**

B Hardware Limitations by Vendor

Limitations by Vendor **B-2**

National Instruments Hardware Limitations **B-3**

Digilent Analog Discovery Hardware Limitations **B-4**

Measurement Computing Hardware Limitations **B-5**

Analog Devices ADALM1000 Limitations **B-6**

Examples by Vendor **B-7**

Analog Devices ADALM1000 Examples **B-8**

Digilent Analog Discovery Hardware Examples **B-9**

Measurement Computing Hardware Examples **B-10**

National Instruments Hardware Examples **B-11**
Getting Started and Device Discovery **B-11**
Analog Input and Output **B-11**
Digital Input and Output **B-11**
Counters and Timers **B-11**
Simultaneous and Synchronized Operations **B-12**
Simulink Data Acquisition **B-12**

Windows Sound Card Examples **B-13**

18 Data Acquisition Toolbox Examples

Getting Started with NI Devices . . . 18-3

Getting Started with MCC Devices . . . 18-7

Discover NI Devices . . . 18-10

Discover MCC Devices . . . 18-12

Acquire Data Using NI Devices . . . 18-14

Acquire Continuous and Background Data Using NI Devices . . . 18-18

Acquire Data from Multiple Channels using an MCC Device . . . 18-22

Acquire Data from an Accelerometer . . . 18-26

Measure Strain Using an Analog Bridge Sensor . . . 18-28

Acquire Temperature Data From a Thermocouple . . . 18-31

Acquire Temperature Data From an RTD . . . 18-34

Acquire and Analyze Sound Pressure Data From an IEPE Microphone . . . 18-37

Acquire and Analyze Noisy Clock Signals . . . 18-41

Generate Voltage Signals Using NI Devices . . . 18-51

Generate Signals on NI Devices That Output Current . . . 18-54

Generate Continuous and Background Signals Using NI Devices . . . 18-57

Simultaneously Acquire Data and Generate Signals . . . 18-60

Log Analog Input Data to a File Using NI Devices . . . 18-64

Getting Started Acquiring Data with Digilent Analog Discovery . . . 18-68

Getting Started Generating Data with Digilent Analog Discovery . . . 18-71

Acquiring and Generating Data at the Same Time with Digilent Analog Discovery . . . 18-73

Generate Standard Periodic Waveforms Using Digilent Analog Discovery . . . 18-76

Generate Arbitrary Periodic Waveforms Using Digilent Analog Discovery . . . 18-79

Acquire Continuous Audio Data **18-83**

Generate Audio Signals **18-86**

Generating Multichannel Audio **18-88**

Capture Data with Software-Analog Triggering **18-92**

Count Pulses on a Digital Signal Using NI Devices **18-101**

Measure Frequency Using NI Devices **18-104**

Measure Pulse Width Using NI Devices **18-106**

Generate Pulse Width Modulated Signals Using NI Devices **18-108**

Measure Angular Position with an Incremental Rotary Encoder **18-110**

Control Stepper Motor Using Digital Outputs **18-115**

Communicate with I2C Devices and Analyze Bus Signals Using Digital IO **18-118**

Synchronize NI PCI Devices Using RTSI **18-125**

Start a Multi-Trigger Acquisition on an External Event **18-128**

Perform Live Acquisition, Signal Processing, and Generation **18-130**

Perform Spectral Analysis on Live Data **18-132**

Acquire Data from Two Devices at Different Rates **18-136**

Characterize an LED with ADALM1000 . . . . . **18-139**

Estimate the Transfer Function of a Circuit with ADALM1000 . . . . . **18-143**

Create an App for Analog Triggered Data Acquisition **18-150**

Analog Triggered Data Acquisition Using Stateflow Charts **18-153**

Create an App for Live Data Acquisition **18-157**

Acquire Data Using NI FieldDAQ Device **18-159**

Create an Echometer Using Audio Measurements **18-162**

Get Started Reading a TDMS-File **18-172**

Read Multiple TDMS-Files into MATLAB **18-175**

Read a Large TDMS-File into MATLAB **18-178**

Get Started Writing a TDMS-File **18-180**

Write Timetable Data to TDMS-file **18-184**

Write Metadata to TDMS-File **18-187**

Merge Multiple TDMS-Files **18-190**

Analyze TDMS-Files Using Tall Tables **18-192**

Impulse Response Measurement Using a NI USB-4431 Device **18-197**

Introduction to Data Acquisition

- "Data Acquisition Toolbox Product Description" on page 1-2
- "Product Capabilities" on page 1-3
- "Anatomy of a Data Acquisition Experiment" on page 1-4
- "Data Acquisition System" on page 1-5
- "Analog Input Subsystem" on page 1-13
- "Making Quality Measurements" on page 1-22
- "Selected Bibliography" on page 1-29

Data Acquisition Toolbox Product Description

Connect to data acquisition cards, devices, and modules

Data Acquisition Toolbox provides apps and functions for configuring data acquisition hardware, reading data into MATLAB® and Simulink®, and writing data to DAQ analog and digital output channels. The toolbox supports a variety of DAQ hardware, including USB, PCI, PCI Express®, PXI®, and PXI-Express devices, from National Instruments™ and other vendors.

The toolbox apps let you interactively set up a data acquisition interface and configure it to your hardware. You can then generate equivalent MATLAB code to automate your data acquisition. Toolbox functions give you the flexibility to control the analog input, analog output, counter/timer, and digital I/O subsystems of a DAQ device. You can access device-specific features and synchronize data acquired from multiple devices.

You can analyze data as you acquire it or save it for post-processing. You can also automate tests and make iterative updates to your test setup based on analysis results.

Product Capabilities

In this section...
"Understanding Data Acquisition Toolbox" on page 1-3
"Supported Hardware" on page 1-3

Understanding Data Acquisition Toolbox

Data Acquisition Toolbox enables you to:

- Configure external hardware devices.
- Read data into MATLAB for immediate analysis.
- Generate signals on device output channels.

Data Acquisition Toolbox is a collection of functions, blocks, apps, and a MEX-file (shared library) built on the MATLAB technical computing environment. The toolbox and its support packages also provide several dynamic link libraries (DLLs) called adaptors, which enable you to interface with specific hardware. The toolbox provides you with these main features:

- A framework for bringing live, measured data into the MATLAB workspace using PC-compatible, plug-in data acquisition hardware
- Support for analog input (AI), analog output (AO), and digital I/O (DIO) subsystems, including simultaneous analog I/O conversions
- Support for these popular hardware vendors/devices:
  - National Instruments CompactDAQ chassis
  - National Instruments boards that use NI-DAQmx software
  - Microsoft® Windows® sound cards
  - Digilent® Analog Discovery hardware
  - Measurement Computing™ hardware
  - Analog Devices® ADALM1000
  - Measurement Computingdevices
- Event-driven acquisitions

Supported Hardware

The list of hardware supported by Data Acquisition Toolbox can change in each release.

To see the full list of hardware that the toolbox supports, visit the supported hardware page at `https://www.mathworks.com/hardware-support/data-acquistion-software.html`.

Anatomy of a Data Acquisition Experiment

In this section...

"System Setup" on page 1-4
"Calibration" on page 1-4
"Trials" on page 1-4

System Setup

The first step in any data acquisition experiment is to install the hardware and software. Hardware installation consists of plugging a board into your computer or installing modules into an external chassis. Software installation consists of loading hardware drivers and application software onto your computer. After the hardware and software are installed, you can attach your sensors.

Calibration

After the hardware and software are installed and the sensors are connected, the data acquisition hardware should be *calibrated*. Calibration consists of providing a known input to the system and recording the output. For many data acquisition devices, calibration can be easily accomplished with software provided by the vendor.

Trials

After the hardware is set up and calibrated, you can begin to acquire data. You might think that if you completely understand the characteristics of the signal you are measuring, then you should be able to configure your data acquisition system and acquire the data.

However, your sensor might be picking up unacceptable noise levels and require shielding, or you might need to run the device at a higher rate, or perhaps you need to add an antialias filter to remove unwanted frequency components.

These effects act as obstacles between you and a precise, accurate measurement. To overcome these obstacles, you need to experiment with different hardware and software configurations. In other words, you need to perform multiple data acquisition trials.

Data Acquisition System

In this section...

"Overview" on page 1-5

"Data Acquisition Hardware" on page 1-6

"Sensors" on page 1-7

"Signal Conditioning" on page 1-9

"The Computer" on page 1-11

"Software" on page 1-11

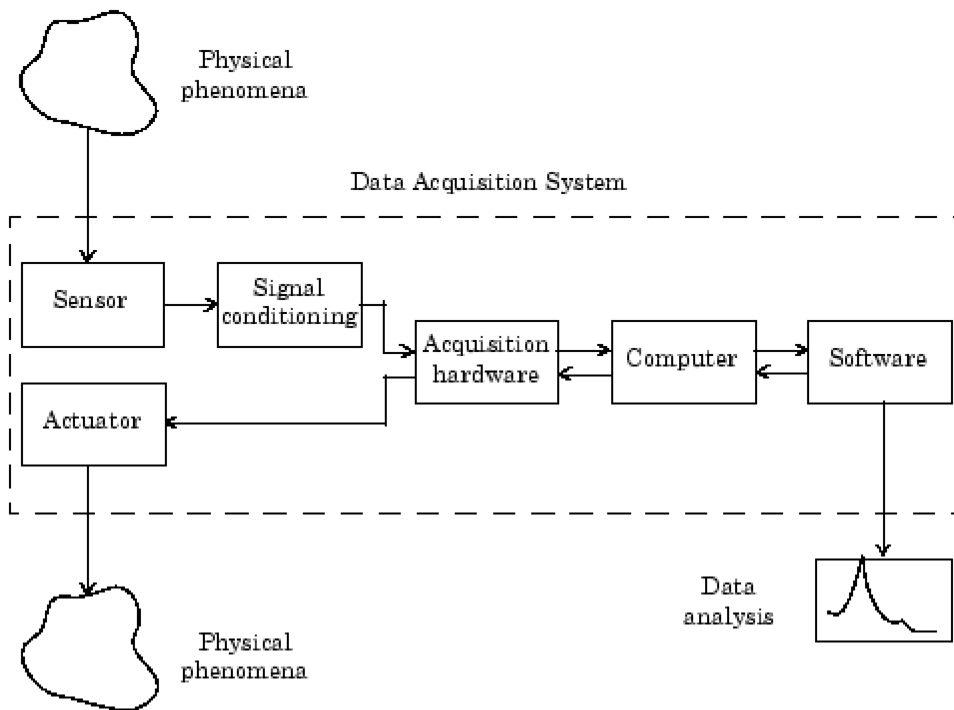
Overview

Data Acquisition Toolbox, with the MATLAB technical computing environment, gives you the ability to generate, measure. and analyze physical phenomena. The purpose of any data acquisition system is to provide you with the tools and resources to do this.

You can think of a data acquisition system as a collection of software and hardware that connects your program to the physical world. A typical data acquisition system consists of these components:

Components	Description
Data acquisition hardware	At the heart of any data acquisition system lies the data acquisition hardware. The main function of this hardware is to convert analog signals to digital signals, and to convert digital signals to analog signals.
Sensors and actuators (transducers)	Sensors and actuators are types of *transducers*. A transducer is a device that converts input energy of one form into output energy of another form. For example, a microphone is a sensor that converts sound energy (in the form of pressure) into electrical energy, while a loudspeaker is an actuator that converts electrical energy into sound energy.
Signal conditioning hardware	Sensor signals are often incompatible with data acquisition hardware. To overcome this incompatibility, the signal must be conditioned. For example, you might need to condition an input signal by amplifying it or by removing unwanted frequency components. Output signals might need conditioning as well.
Computer	The computer provides a processor, a system clock, a bus to transfer data, and memory and disk space to store data.
Software	Data acquisition software allows you to exchange information between the computer and the hardware. For example, typical software allows you to configure the sampling rate of your board, and acquire a predefined amount of data.

The following diagram illustrates the data acquisition components, and their relationships to each other.



The figure depicts the two important features of a data acquisition system:

- Signals are input to a sensor, conditioned, converted into bits that a computer can read, and analyzed to extract meaningful information.

  For example, sound level data is acquired from a microphone, amplified, digitized by a sound card, and stored in the MATLAB workspace for subsequent analysis of frequency content.
- Data from a computer is converted into an analog signal and output to an actuator.

  For example, a vector of data in the MATLAB workspace is converted to an analog signal by a sound card and output to a loudspeaker.

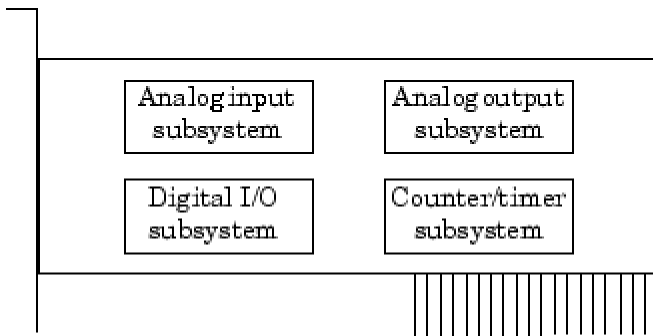
Data Acquisition Hardware

Data acquisition hardware is either internal and installed directly into an expansion slot inside your computer, or external and connected to your computer through an external cable, which is typically a USB cable.

At the simplest level, data acquisition hardware is characterized by the *subsystems* that comprise it. A subsystem is a component of your data acquisition hardware that performs a specialized task. Common subsystems include

- Analog input
- Analog output
- Digital input/output
- Counter/timer

Hardware devices that consist of multiple subsystems, such as the one depicted below, are called *multifunction boards*.



Analog Input Subsystems

Analog input subsystems convert real-world analog input signals from a sensor into bits that can be read by your computer. Perhaps the most common of all subsystems, they are typically available in multichannel devices offering 12 or 16 bits of resolution.

Analog input subsystems are also referred to as AI subsystems, A/D converters, or ADCs.

Analog Output Subsystems

Analog output subsystems convert digital data stored on your computer to a real-world analog signal. These subsystems perform the inverse conversion of analog input subsystems. Typical acquisition boards offer two output channels with 12 bits of resolution, with special hardware available to support multiple channel analog output operations.

Analog output subsystems are also referred to as AO subsystems, D/A converters, or DACs.

Digital Input/Output Subsystems

Digital input/output (DIO) subsystems are designed to input and output digital values (logic levels) to and from hardware. These values are typically handled either as single bits or *lines*, or as a *port*, which typically consists of eight lines.

While most popular data acquisition cards include some digital I/O capability, it is usually limited to simple operations. Special dedicated hardware is often necessary for performing advanced digital I/O operations.

Counter/Timer Subsystems

Counter/timer (C/T) subsystems are used for event counting, frequency and period measurement, and pulse train generation.

Sensors

A sensor converts the physical phenomena of interest into signals that are input to your data acquisition hardware. There are two main types of sensors based on the output they produce: digital sensors and analog sensors.

Digital sensors produce an output signal that is a digital representation of the input signal, and has discrete values of magnitude measured at discrete times. A digital sensor must output logic levels that are compatible with the digital receiver. Some standard logic levels include transistor-transistor logic (TTL) and emitter-coupled logic (ECL). Examples of digital sensors include switches and position encoders.

Analog sensors produce an output signal that is directly proportional to the input signal, and is continuous in both magnitude and time. Most physical variables such as temperature, pressure, and acceleration are continuous in nature and are readily measured with an analog sensor. For example, the temperature of an automobile cooling system and the acceleration produced by a child on a swing both vary continuously.

The sensor you use depends on the phenomena you are measuring. Some common analog sensors and the physical variables they measure are listed below.

Common Analog Sensors

Sensor	Physical Variable
Accelerometer	Acceleration
Microphone	Pressure
Pressure gauge	Pressure
Resistive temperature device (RTD)	Temperature
Strain gauge	Force
Thermocouple	Temperature

When choosing the best analog sensor to use, you must match the characteristics of the physical variable you are measuring with the characteristics of the sensor. The two most important sensor characteristics are:

- The sensor output
- The sensor bandwidth

Note You can use thermocouples and accelerometers without performing linear conversions.

Sensor Output

The output from a sensor can be an analog signal or a digital signal, and the output variable is usually a voltage although some sensors output current.

Current Signals

Current is often used to transmit signals in noisy environments because it is much less affected by environmental noise. The full scale range of the current signal is often either 4-20 mA or 0-20 mA. A 4-20 mA signal has the advantage that even at minimum signal value, there should be a detectable current flowing. The absence of this indicates a wiring problem.

Voltage Signals

The most commonly interfaced signal is a voltage signal. For example, thermocouples, strain gauges, and accelerometers all produce voltage signals. There are three major aspects of a voltage signal that you need to consider:

- **Amplitude**

  If the signal is less than a few millivolts, you might need to amplify it. If it is greater than the maximum range of your analog input hardware (typically ±10 V), you must divide the signal down using a resistor network.

The amplitude is related to the sensitivity (resolution) of your hardware. Refer to Accuracy and Precision on page 1-22 for more information about hardware sensitivity.

- **Frequency**

  Whenever you acquire data, you should decide the highest frequency you want to measure.

  The highest frequency component of the signal determines how often you should sample the input. If you have more than one input, but only one analog input subsystem, then the overall sampling rate goes up in proportion to the number of inputs. Higher frequencies might be present as noise, which you can remove by filtering the signal before it is digitized.

  If you sample the input signal at least twice as fast as the highest frequency component, then that signal will be uniquely characterized. However, this rate might not mimic the waveform very closely. For a rapidly varying signal, you might need a sampling rate of roughly 10 to 20 times the highest frequency to get an accurate picture of the waveform. For slowly varying signals, you need only consider the minimum time for a significant change in the signal.

  The frequency is related to the bandwidth of your measurement. Bandwidth is discussed in “Sensor Bandwidth” on page 1-9.

- **Duration**

  How long do you want to sample the signal for? If you are storing data to memory or to a disk file, then the duration determines the storage resources required. The format of the stored data also affects the amount of storage space required. For example, data stored in ASCII format takes more space than data stored in binary format.

Sensor Bandwidth

In a real-world data acquisition experiment, the physical phenomena you are measuring have expected limits. For example, the temperature of your automobile's cooling system varies continuously between its low limit and high limit. The temperature limits, as well as how rapidly the temperature varies between the limits, depends on several factors including your driving habits, the weather, and the condition of the cooling system. The expected limits might be readily approximated, but there are an infinite number of possible temperatures that you can measure at a given time. As explained in Quantization on page 1-15, these unlimited possibilities are mapped to a finite set of values by your data acquisition hardware.

The *bandwidth* is given by the range of frequencies present in the signal being measured. You can also think of bandwidth as being related to the rate of change of the signal. A slowly varying signal has a low bandwidth, while a rapidly varying signal has a high bandwidth. To properly measure the physical phenomena of interest, the sensor bandwidth must be compatible with the measurement bandwidth.

You might want to use sensors with the widest possible bandwidth when making any physical measurement. This is the one way to ensure that the basic measurement system is capable of responding linearly over the full range of interest. However, the wider the bandwidth of the sensor, the more you must be concerned with eliminating sensor response to unwanted frequency components.

Signal Conditioning

Sensor signals are often incompatible with data acquisition hardware. To overcome this incompatibility, the sensor signal must be conditioned. The type of signal conditioning required

depends on the sensor you are using. For example, a signal might have a small amplitude and require amplification, or it might contain unwanted frequency components and require filtering. Common ways to condition signals include

- Amplification
- Filtering
- Electrical isolation
- Multiplexing
- Excitation source

Amplification

Low-level – less than approximately 100 millivolts – usually need to be amplified. High-level signals might also require amplification depending on the input range of the analog input subsystem.

For example, the output signal from a thermocouple is small and must be amplified before it is digitized. Signal amplification allows you to reduce noise and to make use of the full range of your hardware thereby increasing the resolution of the measurement.

Filtering

Filtering removes unwanted noise from the signal of interest. A noise filter is used on slowly varying signals such as temperature to attenuate higher frequency signals that can reduce the accuracy of your measurement.

Rapidly varying signals such as vibration often require a different type of filter known as an antialiasing filter. An antialiasing filter removes undesirable higher frequencies that might lead to erroneous measurements.

Electrical Isolation

If the signal of interest contains high-voltage transients that could damage the computer, then the sensor signals should be electrically isolated from the computer for safety purposes.

You can also use electrical isolation to make sure that the readings from the data acquisition hardware are not affected by differences in ground potentials. For example, when the hardware device and the sensor signal are each referenced to separate grounds, problems occur if there is a potential difference between the two grounds. This difference can lead to a *ground loop*, which might cause erroneous measurements. Using electrically isolated signal conditioning modules eliminates the ground loop and ensures that the signals are accurately represented.

Multiplexing

A common technique for measuring several signals with a single measuring device is multiplexing.

Signal conditioning devices for analog signals often provide multiplexing for use with slowly changing signals such as temperature. This is in addition to any built-in multiplexing on the DAQ board. The A/D converter samples one channel, switches to the next channel and samples it, switches to the next channel, and so on. Because the same A/D converter is sampling many channels, the effective sampling rate of each individual channel is inversely proportional to the number of channels sampled.

You must take care when using multiplexers so that the switched signal has sufficient time to settle. Refer to Noise on page 1-25 for more information about settling time.

Excitation Source

Some sensors require an excitation source to operate. For example, strain gauges and resistive temperature devices (RTDs) require external voltage or current excitation. Signal conditioning modules for these sensors usually provide the necessary excitation. RTD measurements are usually made with a current source that converts the variation in resistance to a measurable voltage.

The Computer

The computer provides a processor, a system clock, a bus to transfer data, and memory and disk space to store data.

The processor controls how fast data is accepted by the converter. The system clock provides time information about the acquired data. Knowing that you recorded a sensor reading is generally not enough. You might also need to know when that measurement occurred.

Data is transferred from the hardware to system memory via dynamic memory access (DMA) or interrupts. DMA is hardware controlled and therefore extremely fast. Interrupts might be slow because of the latency time between when a board requests interrupt servicing and when the computer responds. The maximum acquisition rate is also determined by the computer's bus architecture. Refer to How Are Acquired Samples Clocked? on page 1-17 for more information about DMA and interrupts.

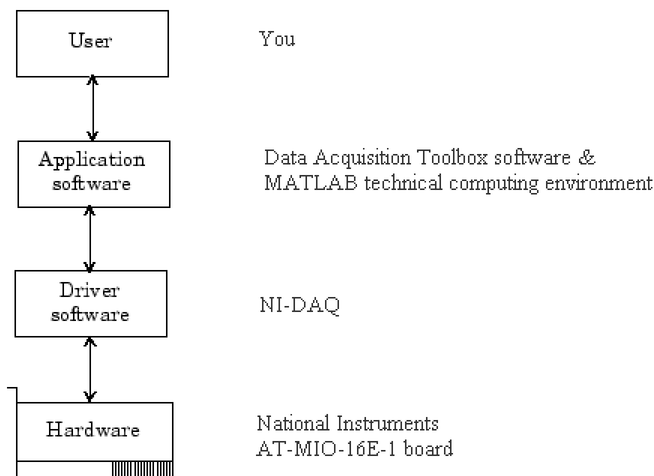
Software

Regardless of the hardware you are using, you must send information to the hardware and receive information from the hardware. You send configuration information to the hardware such as the sampling rate, and receive information from the hardware such as data, status messages, and error messages. You might also need to supply the hardware with information so that you can integrate it with other hardware and with computer resources. This information exchange is accomplished with software.

There are two kinds of software:

- Driver software
- Application software

For example, suppose you are using Data Acquisition Toolbox software with a National Instruments board and its associated driver. The following diagram shows the relationship between you, the driver software, and the application software.



The diagram illustrates that you supply information to the hardware, and you receive information from the hardware.

Driver Software

For a data acquisition device, there is associated driver software that you must use. Driver software allows you to access and control your hardware. Among other things, basic driver software allows you to

- Transfer data to and from the board
- Control the rate at which data is acquired
- Integrate the data acquisition hardware with computer resources such as processor interrupts, DMA, and memory
- Integrate the data acquisition hardware with signal conditioning hardware
- Access multiple subsystems on a given data acquisition board
- Access multiple data acquisition boards

Application Software

Application software provides a convenient front end to the driver software. Basic application software allows you to

- Report relevant information such as the number of samples acquired
- Generate events
- Manage the data stored in computer memory
- Condition a signal
- Plot acquired data

MATLAB and Data Acquisition Toolbox software provide you with these capabilities, and provide tools that let you perform analysis on the data.

Analog Input Subsystem

In this section...

"Function of the Analog Input Subsystem" on page 1-13
"Sampling" on page 1-13
"Quantization" on page 1-15
"Channel Configuration" on page 1-18
"Transferring Data from Hardware to System Memory" on page 1-20

Function of the Analog Input Subsystem

Many data acquisition hardware devices contain one or more subsystems that convert (digitize) real-world sensor signals into numbers your computer can read. Such devices are called analog input subsystems (AI subsystems, A/D converters, or ADCs). After the real-world signal is digitized, you can analyze it, store it in system memory, or store it to a disk file.

The function of the analog input subsystem is to *sample* and *quantize* the analog signal using one or more *channels*. You can think of a channel as a path through which the sensor signal travels. Typical analog input subsystems have eight or 16 input channels available to you. After data is sampled and quantized, it must be transferred to system memory.

Analog signals are continuous in time and in amplitude (within predefined limits). Sampling takes a "snapshot" of the signal at discrete times, while quantization divides the voltage (or current) value into discrete amplitudes.

Sampling

Sampling takes a snapshot of the sensor signal at discrete times. For most applications, the time interval between samples is kept constant (for example, sample every millisecond) unless externally clocked.

For most digital converters, sampling is performed by a sample and hold (S/H) circuit. An S/H circuit usually consists of a signal buffer followed by an electronic switch connected to a capacitor. The operation of an S/H circuit follows these steps:

1. At a given sampling instant, the switch connects the buffer and capacitor to an input.
2. The capacitor is charged to the input voltage.
3. The charge is held until the A/D converter digitizes the signal.
4. For multiple channels connected (multiplexed) to one A/D converter, the previous steps are repeated for each input channel.
5. The entire process is repeated for the next sampling instant.

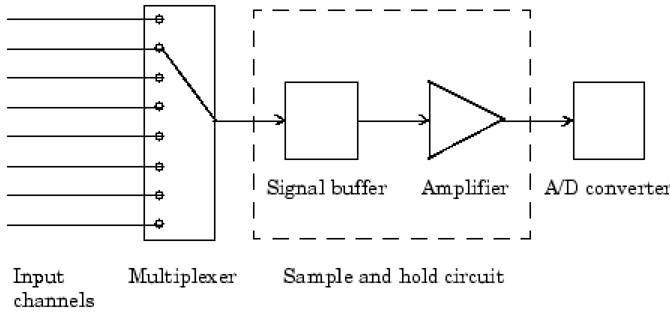
A multiplexer, S/H circuit, and A/D converter are illustrated in the next section.

Hardware can be divided into two main categories based on how signals are sampled: *scanning* hardware, which samples input signals sequentially, and *simultaneous sample and hold* (SS/H) hardware, which samples all signals at the same time. These two types of hardware are discussed below.

Scanning Hardware

Scanning hardware samples a single input signal, converts that signal to a digital value, and then repeats the process for every input channel used. In other words, each input channel is sampled sequentially. A *scan* occurs when each input in a group is sampled once.

As shown below, most data acquisition devices have one A/D converter that is multiplexed to multiple input channels.

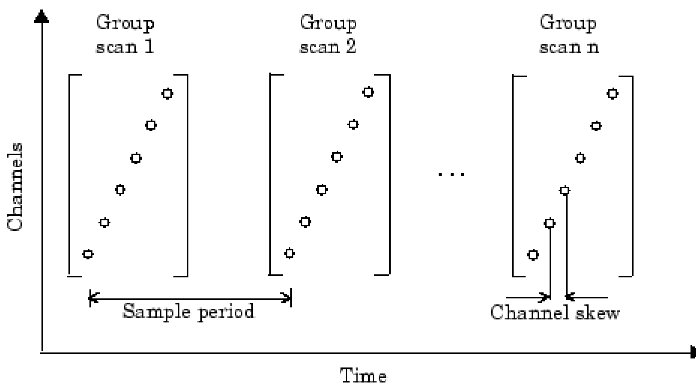


Therefore, if you use multiple channels, those channels cannot be sampled simultaneously and a time gap exists between consecutive sampled channels. This time gap is called the *channel skew*. You can think of the channel skew as the time it takes the analog input subsystem to sample a single channel.

Additionally, the maximum sampling rate your hardware is rated at typically applies for one channel. Therefore, the maximum sampling rate per channel is given by the formula:

$$maximumsamplingrateperchannel = \frac{maximumboardrate}{numberofchannelsscanned}$$

Typically, you can achieve this maximum rate only under ideal conditions. In practice, the sampling rate depends on several characteristics of the analog input subsystem including the settling time and the gain, as well as the channel skew. The following diagram shows the sample period and channel skew for a multichannel configuration using scanning hardware.



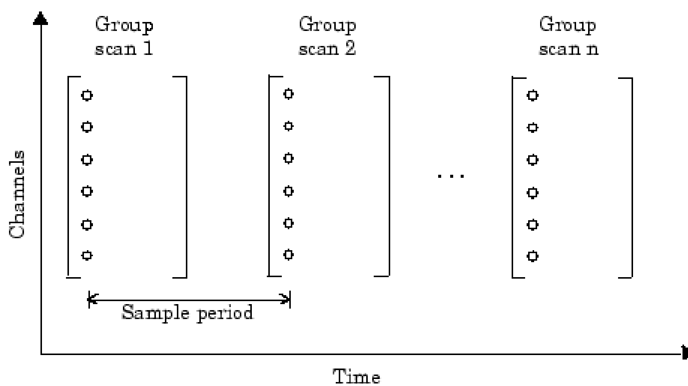
If you cannot tolerate channel skew in your application, you must use hardware that allows simultaneous sampling of all channels. Simultaneous sample and hold hardware is discussed in the next section.

Simultaneous Sample and Hold Hardware

Simultaneous sample and hold (SS/H) hardware samples all input signals at the same time and holds the values until the A/D converter digitizes all the signals. For high-end systems, there can be a separate A/D converter for each input channel.

For example, suppose you need to simultaneously measure the acceleration of multiple accelerometers to determine the vibration of some device under test. To do this, you must use SS/H hardware because it does not have a channel skew. In general, you might need to use SS/H hardware if your sensor signal changes significantly in a time that is less than the channel skew, or if you need to use a transfer function or perform a frequency domain correlation.

The following diagram shows sample period for a multichannel configuration using SS/H hardware. Note that there is no channel skew.

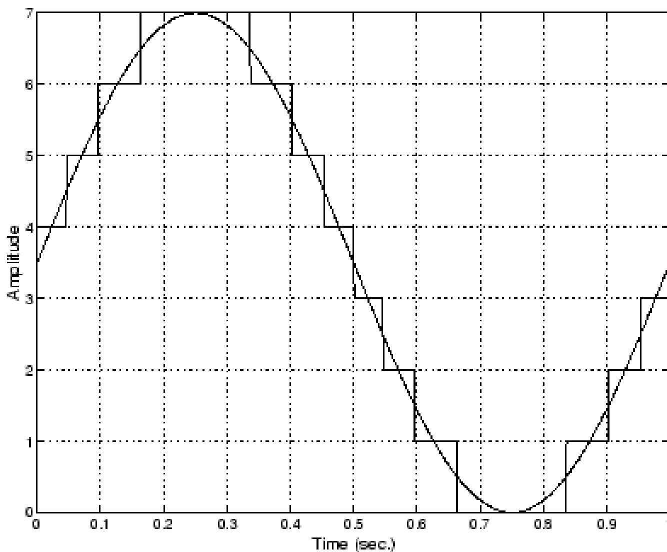


Quantization

As discussed in the previous section, sampling takes a snapshot of the input signal at an instant of time. When the snapshot is taken, the sampled analog signal must be converted from a voltage value to a binary number that the computer can read. The conversion from an infinitely precise amplitude to a binary number is called *quantization*.

During quantization, the A/D converter uses a finite number of evenly spaced values to represent the analog signal. The number of different values is determined by the number of bits used for the conversion. Most modern converters use 12 or 16 bits. Typically, the converter selects the digital value that is closest to the actual sampled value.

The figure below shows a 1 Hz sine wave quantized by a 3 bit A/D converter.

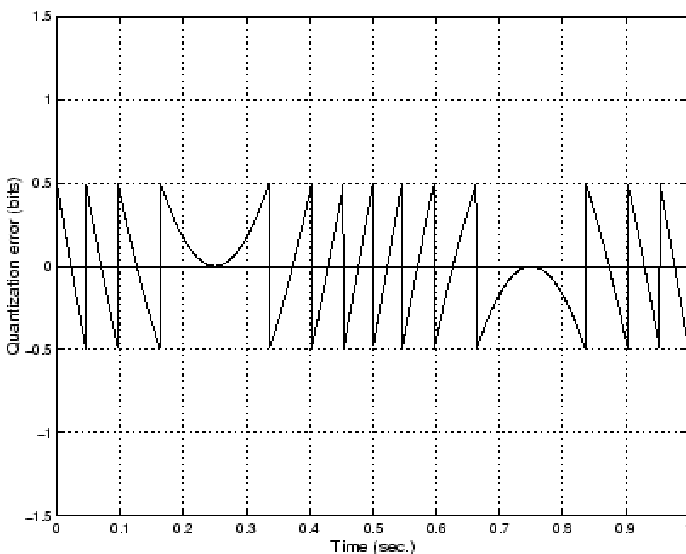


The number of quantized values is given by $2^3 = 8$, the largest representable value is given by 111 = $2^2 + 2^1 + 2^0 = 7.0$, and the smallest representable value is given by 000 = 0.0.

Quantization Error

There is always some error associated with the quantization of a continuous signal. Ideally, the maximum quantization error is ±0.5 least significant bits (LSBs), and over the full input range, the average quantization error is zero.

As shown below, the quantization error for the previous sine wave is calculated by subtracting the actual signal from the quantized signal.



Input Range and Polarity

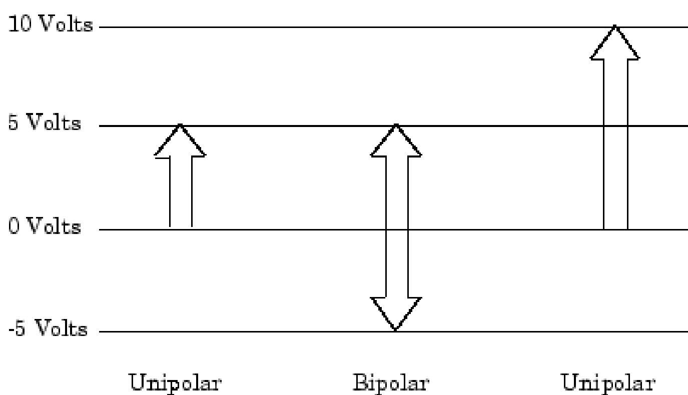
The *input range* of the analog input subsystem is the span of input values for which a conversion is valid. You can change the input range by selecting a different *gain* value. For example, National

Instruments' AT-MIO-16E-1 board has eight gain values ranging from 0.5 to 100. Many boards include a programmable gain amplifier that allows you to change the device gain through software.

When an input signal exceeds the valid input range of the converter, an *overrange* condition occurs. In this case, most devices saturate to the largest representable value, and the converted data is almost definitely incorrect. The gain setting affects the precision of your measurement — the higher (lower) the gain value, the lower (higher) the precision. Refer to How Are Range, Gain, and Measurement Precision Related? on page 1-24 for more information about how input range, gain, and precision are related to each other.

An analog input subsystem can typically convert both *unipolar* signals and *bipolar* signals. A unipolar signal contains only positive values and zero, while a bipolar signal contains positive values, negative values, and zero.

Unipolar and bipolar signals are depicted below. Refer to the figure in “Quantization” on page 1-15 for an example of a unipolar signal.



In many cases, the signal polarity is a fixed characteristic of the sensor and you must configure the input range to match this polarity.

As you can see, it is crucial to understand the range of signals expected from your sensor so that you can configure the input range of the analog input subsystem to maximize resolution and minimize the chance of an overrange condition.

How Are Acquired Samples Clocked?

Samples are acquired from an analog input subsystem at a specific rate by a clock. Like any timing system, data acquisition clocks are characterized their resolution and accuracy. Timing resolution is defined as the smallest time interval that you can accurately measure. The timing accuracy is affected by clock *jitter*. Jitter arises when a clock produces slightly different values for a given time interval.

For any data acquisition system, there are typically three clock sources that you can use: the onboard data acquisition clock, the computer clock, or an external clock. Data Acquisition Toolbox software supports all of these clock sources, depending on the requirements of your hardware.

Onboard Clock

The onboard clock is typically a timer chip on the hardware board that is programmed to generate a pulse stream at the desired rate. The onboard clock generally has high accuracy and low jitter compared to the computer clock. You should always use the onboard clock when the sampling rate is high, and when you require a fixed time interval between samples. The onboard clock is referred to as the *internal clock* in this guide.

Computer Clock

The computer (PC) clock is used for boards that do not possess an onboard clock. The computer clock is less accurate and has more jitter than the onboard clock, and is generally limited to sampling rates below 500 Hz. The computer clock is referred to as the *software clock* in this guide.

External Clock

An external clock is often used when the sampling rate is low and not constant. For example, an external clock source is often used in automotive applications where samples are acquired as a function of crank angle.

Channel Configuration

You can configure input channels in one of these two ways:

- Differential
- Single-ended

Your choice of input channel configuration might depend on whether the input signal is *floating* or *grounded*.

A floating signal uses an isolated ground reference and is not connected to the building ground. As a result, the input signal and hardware device are not connected to a common reference, which can cause the input signal to exceed the valid range of the hardware device. To circumvent this problem, you must connect the signal to the onboard ground of the device. Examples of floating signal sources include ungrounded thermocouples and battery devices.

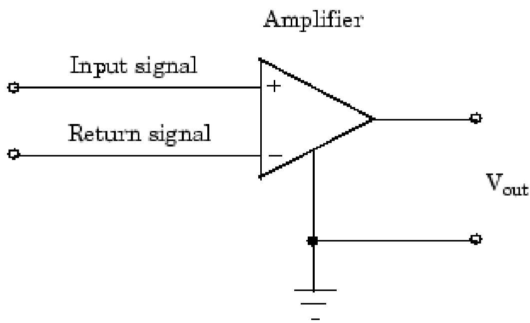
A grounded signal is connected to the building ground. As a result, the input signal and hardware device are connected to a common reference. Examples of grounded signal sources include nonisolated instrument outputs and devices that are connected to the building power system.

Note For more information about channel configuration, refer to your hardware documentation.

Differential Inputs

When you configure your hardware for differential input, there are two signal wires associated with each input signal — one for the input signal and one for the reference (return) signal. The measurement is the difference in voltage between the two wires, which helps reduce noise and any voltage that is common to both wires.

As shown below, the input signal is connected to the positive amplifier socket (labeled +) and the return signal is connected to the negative amplifier socket (labeled -). The amplifier has a third connector that allows these signals to be referenced to ground.



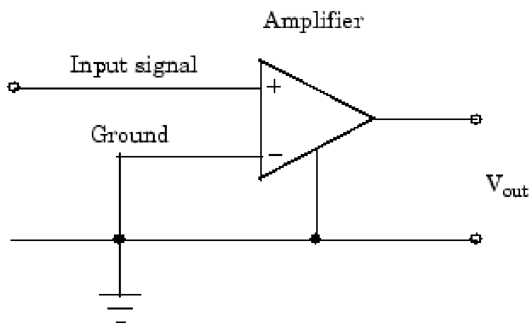
National Instruments recommends that you use differential inputs under any of these conditions:

- The input signal is low level (less than 1 volt).
- The leads connecting the signal are greater than 10 feet.
- The input signal requires a separate ground-reference point or return signal.
- The signal leads travel through a noisy environment.

Single-Ended Inputs

When you configure your hardware for single-ended input, there is one signal wire associated with each input signal, and each input signal is connected to the same ground. Single-ended measurements are more susceptible to noise than differential measurements because of differences in the signal paths.

As shown below, the input signal is connected to the positive amplifier socket (labeled +) and the ground is connected to the negative amplifier socket (labeled -).



National Instruments suggests that you can use single-ended inputs under any of these conditions:

- The input signal is high level (greater than 1 volt).
- The leads connecting the signal are less than 10 feet.
- The input signal can share a common reference point with other signals.

You should use differential input connectors for any input signal that does not meet the preceding conditions. You can configure many National Instruments boards for two different types of single-ended connections:

- Referenced single-ended (RSE) connection

  The RSE configuration is used for floating signal sources. In this case, the hardware device itself provides the reference ground for the input signal.

- Nonreferenced single-ended (NRSE) connection

  The NRSE input configuration is used for grounded signal sources. In this case, the input signal provides its own reference ground and the hardware device should not supply one.

Refer to your National Instruments hardware documentation for more information about RSE and NRSE connections.

Transferring Data from Hardware to System Memory

The transfer of acquired data from the hardware to system memory follows these steps:

1. Acquired data is stored in the hardware's first-in first-out (FIFO) buffer.
2. Data is transferred from the FIFO buffer to system memory using interrupts or DMA.

These steps happen automatically. Typically, all that's required from you is some initial configuration of the hardware device when it is installed.

FIFO Buffer

The FIFO buffer is used to temporarily store acquired data. The data is temporarily stored until it can be transferred to system memory. The process of transferring data into and out of an analog input FIFO buffer is given below:

1. The FIFO buffer stores newly acquired samples at a constant sampling rate.
2. Before the FIFO buffer is filled, the software starts removing the samples. For example, an interrupt is generated when the FIFO is half full, and signals the software to extract the samples as quickly as possible.
3. Because servicing interrupts or programming the DMA controller can take up to a few milliseconds, additional data is stored in the FIFO for future retrieval. For a larger FIFO buffer, longer latencies can be tolerated.
4. The samples are transferred to system memory via the system bus (for example, PCI bus or AT bus). After the samples are transferred, the software is free to perform other tasks until the next interrupt occurs. For example, the data can be processed or saved to a disk file. As long as the average rates of storing and extracting data are equal, acquired data will not be missed and your application should run smoothly.

Interrupts

The slowest but most common method to move acquired data to system memory is for the board to generate an interrupt request (IRQ) signal. This signal can be generated when one sample is acquired or when multiple samples are acquired. The process of transferring data to system memory via interrupts is given below:

1. When data is ready for transfer, the CPU stops whatever it is doing and runs a special interrupt handler routine that saves the current machine registers, and then sets them to access the board.
2. The data is extracted from the board and placed into system memory.
3. The saved machine registers are restored, and the CPU returns to the original interrupted process.

The actual data move is fairly quick, but there is a lot of overhead time spent saving, setting up, and restoring the register information. Therefore, depending on your specific system, transferring data by interrupts might not be a good choice when the sampling rate is greater than around 5 kHz.

DMA

Direct memory access (DMA) is a system whereby samples are automatically stored in system memory while the processor does something else. The process of transferring data via DMA is given below:

1. When data is ready for transfer, the board directs the system DMA controller to put it into in system memory as soon as possible.
2. As soon as the CPU is able (which is usually very quickly), it stops interacting with the data acquisition hardware and the DMA controller moves the data directly into memory.
3. The DMA controller gets ready for the next sample by pointing to the next open memory location.
4. The previous steps are repeated indefinitely, with data going to each open memory location in a continuously circulating buffer. No interaction between the CPU and the board is needed.

Your computer supports several different DMA channels. Depending on your application, you can use one or more of these channels, For example, simultaneous input and output with a sound card requires one DMA channel for the input and another DMA channel for the output.

Making Quality Measurements

In this section...
"What Do You Measure?" on page 1-22
"Accuracy and Precision" on page 1-22
"Noise" on page 1-25
"Matching the Sensor Range and A/D Converter Range" on page 1-25
"How Fast Should a Signal Be Sampled?" on page 1-26

What Do You Measure?

For most data acquisition applications, you need to measure the signal produced by a sensor at a specific rate.

In many cases, the sensor signal is a voltage level that is proportional to the physical phenomena of interest (for example, temperature, pressure, or acceleration). If you are measuring slowly changing (quasi-static) phenomena like temperature, a slow sampling rate usually suffices. If you are measuring rapidly changing (dynamic) phenomena like vibration or acoustic measurements, a fast sampling rate is required.

To make high-quality measurements, you should follow these rules:

- Maximize the precision and accuracy
- Minimize the noise
- Match the sensor range to the A/D range

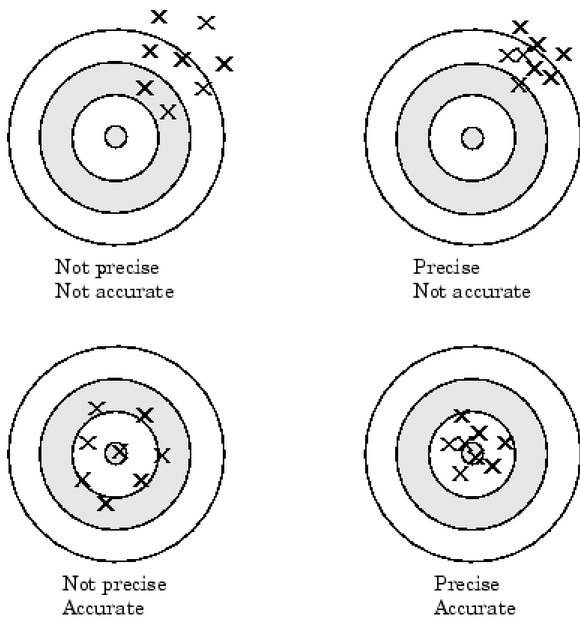
Accuracy and Precision

Whenever you acquire measured data, you should make every effort to maximize its accuracy and precision. The quality of your measurement depends on the accuracy and precision of the entire data acquisition system, and can be limited by such factors as board resolution or environmental noise.

In general terms, the *accuracy* of a measurement determines how close the measurement comes to the true value. Therefore, it indicates the correctness of the result. The *precision* of a measurement reflects how exactly the result is determined without reference to what the result means. The *relative precision* indicates the uncertainty in a measurement as a fraction of the result.

For example, suppose you measure a table top with a meter stick and find its length to be 1.502 meters. This number indicates that the meter stick (and your eyes) can resolve distances down to at least a millimeter. Under most circumstances, this is considered to be a fairly precise measurement with a relative precision of around 1/1500. However, suppose you perform the measurement again and obtain a result of 1.510 meters. After careful consideration, you discover that your initial technique for reading the meter stick was faulty because you did not read it from directly above. Therefore, the first measurement was not accurate.

Precision and accuracy are illustrated below.



For analog input subsystems, accuracy is usually limited by calibration errors while precision is usually limited by the A/D converter. Accuracy and precision are discussed in more detail below.

Accuracy

Accuracy is defined as the agreement between a measured quantity and the true value of that quantity. Every component that appears in the analog signal path affects system accuracy and performance. The overall system accuracy is given by the component with the worst accuracy.

For data acquisition hardware, accuracy is often expressed as a percent or a fraction of the least significant bit (LSB). Under ideal circumstances, board accuracy is typically ±0.5 LSB. Therefore, a 12 bit converter has only 11 usable bits.

Many boards include a programmable gain amplifier, which is located just before the converter input. To prevent system accuracy from being degraded, the accuracy and linearity of the gain must be better than that of the A/D converter. The specified accuracy of a board is also affected by the sampling rate and the *settling time* of the amplifier. The settling time is defined as the time required for the instrumentation amplifier to settle to a specified accuracy. To maintain full accuracy, the amplifier output must settle to a level given by the magnitude of 0.5 LSB before the next conversion, and is on the order of several tenths of a millisecond for most boards.

Settling time is a function of sampling rate and gain value. High rate, high gain configurations require longer settling times while low rate, low gain configurations require shorter settling times.

Precision

The number of bits used to represent an analog signal determines the precision (resolution) of the device. The more bits provided by your board, the more precise your measurement will be. A high precision, high resolution device divides the input range into more divisions thereby allowing a smaller detectable voltage value. A low precision, low resolution device divides the input range into fewer divisions thereby increasing the detectable voltage value.

The overall precision of your data acquisition system is usually determined by the A/D converter, and is specified by the number of bits used to represent the analog signal. Most boards use 12 or 16 bits. The precision of your measurement is given by:

$$\textit{precision} = \text{one part in } 2^{numberofbits}$$

The precision in volts is given by:

$$\textit{precision} = \frac{\textit{voltage range}}{2^{\textit{number of bits}}}$$

For example, if you are using a 12 bit A/D converter configured for a 10 volt range, then

$$\textit{precision} = \frac{10\ \textit{volts}}{2^{12}}$$

This means that the converter can detect voltage differences at the level of 0.00244 volts (2.44 mV).

How Are Range, Gain, and Measurement Precision Related?

When you configure the input range and gain of your analog input subsystem, the end result should maximize the measurement resolution and minimize the chance of an overrange condition. The actual input range is given by the formula:

$$\textit{actual input range} = \frac{\textit{input range}}{\textit{gain}}$$

The relationship between gain, actual input range, and precision for a unipolar and bipolar signal having an input range of 10 V is shown below.

Relationship Between Input Range, Gain, and Precision

Input Range	Gain	Actual Input Range	Precision (12 Bit A/D)
0 to 10 V	1.0	0 to 10 V	2.44 mV
	2.0	0 to 5 V	1.22 mV
	5.0	0 to 2 V	0.488 mV
	10.0	0 to 1 V	0.244 mV
-5 to 5 V	0.5	-10 to 10 V	4.88 mV
	1.0	-5 to 5 V	2.44 mV
	2.0	-2.5 to 2.5 V	1.22 mV
	5.0	-1.0 to 1.0 V	0.488 mV
	10.0	-0.5 to 0.5 V	0.244 mV

As shown in the table, the gain affects the precision of your measurement. If you select a gain that decreases the actual input range, then the precision increases. Conversely, if you select a gain that increases the actual input range, then the precision decreases. This is because the actual input range varies but the number of bits used by the A/D converter remains fixed.

Note With Data Acquisition Toolbox software, you do not have to specify the range and gain. Instead, you simply specify the actual input range desired.

Noise

Noise is considered to be any measurement that is not part of the phenomena of interest. Noise can be generated within the electrical components of the input amplifier (internal noise), or it can be added to the signal as it travels down the input wires to the amplifier (external noise). Techniques that you can use to reduce the effects of noise are described below.

Removing Internal Noise

Internal noise arises from thermal effects in the amplifier. Amplifiers typically generate a few microvolts of internal noise, which limits the resolution of the signal to this level. The amount of noise added to the signal depends on the bandwidth of the input amplifier.

To reduce internal noise, you should select an amplifier with a bandwidth that closely matches the bandwidth of the input signal.

Removing External Noise

External noise arises from many sources. For example, many data acquisition experiments are subject to 60 Hz noise generated by AC power circuits. This type of noise is referred to as *pick-up* or *hum*, and appears as a sinusoidal interference signal in the measurement circuit. Another common interference source is fluorescent lighting. These lights generate an arc at twice the power line frequency (120 Hz).

Noise is added to the acquisition circuit from these external sources because the signal leads act as aerials picking up environmental electrical activity. Much of this noise is common to both signal wires. To remove most of this common-mode voltage, you should

- Configure the input channels in differential mode. Refer to Channel Configuration on page 1-18 for more information about channel configuration.
- Use signal wires that are twisted together rather than separate.
- Keep the signal wires as short as possible.
- Keep the signal wires as far away as possible from environmental electrical activity.

Filtering

Filtering also reduces signal noise. For many data acquisition applications, a low-pass filter is beneficial. As the name suggests, a low-pass filter passes the lower frequency components but attenuates the higher frequency components. The cut-off frequency of the filter must be compatible with the frequencies present in the signal of interest and the sampling rate used for the A/D conversion.

A low-pass filter that's used to prevent higher frequencies from introducing distortion into the digitized signal is known as an antialiasing filter if the cut-off occurs at the Nyquist frequency. That is, the filter removes frequencies greater than one-half the sampling frequency. These filters generally have a sharper cut-off than the normal low-pass filter used to condition a signal. Antialiasing filters are specified according to the sampling rate of the system and there must be one filter per input signal.

Matching the Sensor Range and A/D Converter Range

When sensor data is digitized by an A/D converter, you must be aware of these two issues:

- The expected range of the data produced by your sensor. This range depends on the physical phenomena you are measuring and the output range of the sensor.
- The range of your A/D converter. For many devices, the hardware range is specified by the gain and polarity.

You should select the sensor and hardware ranges such that the maximum precision is obtained, and the full dynamic range of the input signal is covered.

For example, suppose you are using a microphone with a dynamic range of 20 dB to 140 dB and an output sensitivity of 50 mV/Pa. If you are measuring street noise in your application, then you might expect that the sound level never exceeds 80 dB, which corresponds to a sound pressure magnitude of 200 mPa and a voltage output from the microphone of 10 mV. Under these conditions, you should set the input range of your data acquisition card for a maximum signal amplitude of 10 mV, or a little more.

How Fast Should a Signal Be Sampled?

Whenever a continuous signal is sampled, some information is lost. The key objective is to sample at a rate such that the signal of interest is well characterized and the amount of information lost is minimized.

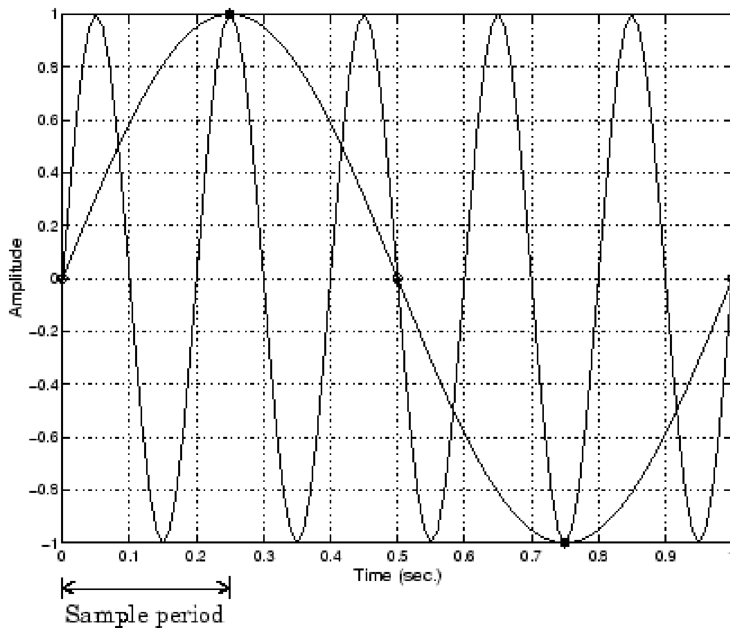
If you sample at a rate that is too slow, then signal aliasing can occur. Aliasing can occur for both rapidly varying signals and slowly varying signals. For example, suppose you are measuring temperature once a minute. If your acquisition system is picking up a 60-Hz hum from an AC power supply, then that hum will appear as constant noise level if you are sampling at 30 Hz.

Aliasing occurs when the sampled signal contains frequency components greater than one-half the sampling rate. The frequency components could originate from the signal of interest in which case you are undersampling and should increase the sampling rate. The frequency components could also originate from noise in which case you might need to condition the signal using a filter. The rule used to prevent aliasing is given by the *Nyquist theorem*, which states that

- An analog signal can be uniquely reconstructed, without error, from samples taken at equal time intervals.
- The sampling rate must be equal to or greater than twice the highest frequency component in the analog signal. A frequency of one-half the sampling rate is called the Nyquist frequency.

However, if your input signal is corrupted by noise, then aliasing can still occur.

For example, suppose you configure your A/D converter to sample at a rate of 4 samples per second (4 S/s or 4 Hz), and the signal of interest is a 1 Hz sine wave. Because the signal frequency is one-fourth the sampling rate, then according to the Nyquist theorem, it should be completely characterized. However, if a 5 Hz sine wave is also present, then these two signals cannot be distinguished. In other words, the 1 Hz sine wave produces the same samples as the 5 Hz sine wave when the sampling rate is 4 S/s. The following diagram illustrates this condition.



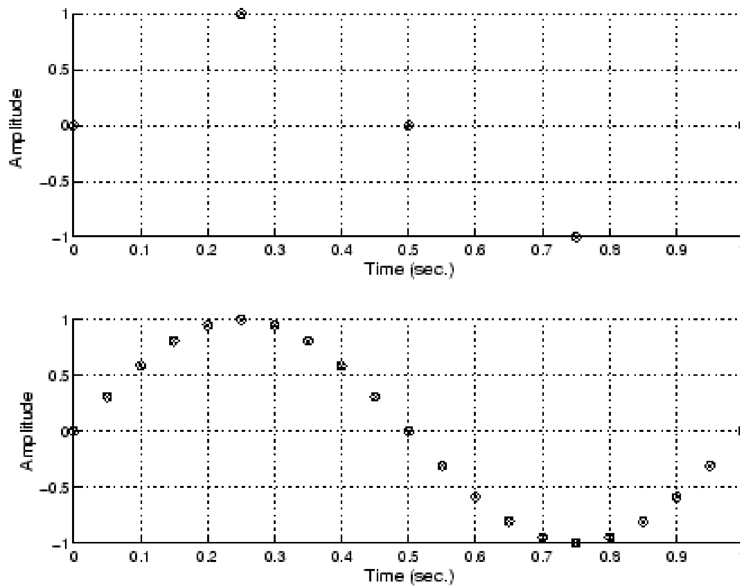
In a real-world data acquisition environment, you might need to condition the signal by filtering out the high frequency components.

Even though the samples appear to represent a sine wave with a frequency of one-fourth the sampling rate, the actual signal could be any sine wave with a frequency of:

$(n \pm 0.25) \times (\textit{sampling rate})$

where n is zero or any positive integer. For this example, the actual signal could be at a frequency of 3 Hz, 5 Hz, 7 Hz, 9 Hz, and so on. The relationship 0.25 × (*sampling rate*) is called the *alias* of a signal that might be at another frequency. In other words, aliasing occurs when one frequency assumes the identity of another frequency.

If you sample the input signal at least twice as fast as the highest frequency component, then that signal might be uniquely characterized, but this rate would not mimic the waveform very closely. As shown below, to get an accurate picture of the waveform, you need a sampling rate of roughly 10 to 20 times the highest frequency.



As shown in the top figure, the low sampling rate produces a sampled signal that appears to be a triangular waveform. As shown in the bottom figure, a higher fidelity sampled signal is produced when the sampling rate is higher. In the latter case, the sampled signal actually looks like a sine wave.

How Can Aliasing Be Eliminated?

The primary considerations involved in antialiasing are the sampling rate of the A/D converter and the frequencies present in the sampled data. To eliminate aliasing, you must

- Establish the useful bandwidth of the measurement.
- Select a sensor with sufficient bandwidth.
- Select a low-pass antialiasing analog filter that can eliminate all frequencies exceeding this bandwidth.
- Sample the data at a rate at least twice that of the filter's upper cutoff frequency.

Selected Bibliography

[1] *Transducer Interfacing Handbook — A Guide to Analog Signal Conditioning*, edited by Daniel H. Sheingold; Analog Devices Inc., Norwood, MA, 1980.

[2] Bentley, John P., *Principles of Measurement Systems, Second Edition*; Longman Scientific and Technical, Harlow, Essex, UK, 1988.

[3] Bevington, Philip R., *Data Reduction and Error Analysis for the Physical Sciences*; McGraw-Hill, New York, NY, 1969.

[4] *Carr, Joseph J., Sensors*; Prompt Publications, Indianapolis, IN, 1997.

[5] *The Measurement, Instrumentation, and Sensors Handbook*, edited by John G. Webster; CRC Press, Boca Raton, FL, 1999.

[6] *PCI-MIO E Series User Manual, January 1997 Edition*; Part Number 320945B-01, National Instruments, Austin, TX, 1997.

Using Data Acquisition Toolbox Software

This topic provides the information you need to get started with Data Acquisition Toolbox software. The sections are as follows.

- “Installation Information” on page 2-2
- “Access Your Hardware” on page 2-3

Installation Information

In this section...
“Prerequisites” on page 2-2
“Toolbox Installation” on page 2-2
“Hardware and Driver Installation” on page 2-2

Prerequisites

To acquire live, measured data or generate signals between the MATLAB workspace, you must install these components:

- MATLAB, and optionally Simulink
- Data Acquisition Toolbox
- The support package for your data acquisition device vendor
- A supported data acquisition device (see `https://www.mathworks.com/hardware-support/data-acquistion-software.html`)

Toolbox Installation

To determine if Data Acquisition Toolbox software is installed on your system, type

```
ver
```

at the MATLAB prompt. The Command Window lists information about the software versions you are running, including installed add-on products and their version numbers. Check the list to see if Data Acquisition Toolbox appears. For information about installing the toolbox, see the MATLAB Installation documentation.

If you experience installation difficulties and have Internet access, look for the license manager and installation information at the MathWorks website (`https://www.mathworks.com`).

Hardware and Driver Installation

Device drivers and other vendor-specific software are available as Support Packages from the Add-Ons menu. See “Install Hardware Support Package for Vendor Support” on page 5-2.

See Also

Related Examples

- “Set Up Your System for Device Detection” on page A-10

Access Your Hardware

In this section...

"Connect to Your Hardware" on page 2-3
"Examine Your Hardware Resources" on page 2-3
"Acquire Audio Data" on page 2-4
"Generate Audio Data" on page 2-4
"Acquire and Generate Digital Data" on page 2-5

Connect to Your Hardware

Perhaps the most effective way to get started with Data Acquisition Toolbox software is to connect to your hardware, and input or output data.

Each example in this topic illustrates a typical data acquisition workflow. A workflow comprises all the steps you are likely to take when acquiring or outputting data using a supported hardware device. You should keep these steps in mind when constructing your own data acquisition applications.

Note that the analog input and analog output examples use a sound card, while the digital I/O example uses a National Instruments board. If you are using a different supported hardware device, you should modify the vendor name and the device ID as needed.

If you want detailed information about any functions that are used, refer to the list of functions.

Note If you are connecting to a CompactDAQ devices or a counter/timer device, see "Counter and Timer Input and Output".

Examine Your Hardware Resources

You can examine the data acquisition hardware resources visible to the toolbox with the `daqvendorlist` and `daqlist` functions. Hardware resources include installed boards, hardware drivers, and adaptors.

For example, to view the available audio devices, type:

```
daqlist("directsound")
```

To view available National Instruments devices, type:

```
daqlist("ni")
```

To view all available devices, type:

```
daqlist
```

To view the operational status of hardware vendors, type:

```
daqvendorlist
```

Acquire Audio Data

If you have a sound card installed, you can run the following example, which acquires 1 second of data an audio input hardware channels, and then plots the acquired data.

You should modify this example to suit your specific application needs.

1. **Create a DataAcquisition object** — Create the DataAcquisition object d for a sound card.

```
d = daq('directsound');
```

2. Identify the system devices and their IDs for audio input and output.

```
daqlist("directsound")

  7×4 table

    DeviceID                          Description                                                    Model                             
    ________    __________________________________________________________    _______________________________________________    __
                                                                                                                                          
    "Audio0"    "DirectSound Primary Sound Capture Driver"                    "Primary Sound Capture Driver"                     [1:
    "Audio1"    "DirectSound Headset Microphone (Plantronics BT600)"          "Headset Microphone (Plantronics BT600)"           [1:
    "Audio2"    "DirectSound Primary Sound Driver"                            "Primary Sound Driver"                             [1:
    "Audio3"    "DirectSound Headset Earphone (Plantronics BT600)"            "Headset Earphone (Plantronics BT600)"             [1:
    "Audio4"    "DirectSound Speakers (2- Realtek High Definition Audio)"     "Speakers (2- Realtek High Definition Audio)"      [1:
    "Audio5"    "DirectSound Speakers (Realtek High Definition Audio)"        "Speakers (Realtek High Definition Audio)"         [1:
    "Audio6"    "DirectSound LEN LT2452pwC (NVIDIA High Definition Audio)"    "LEN LT2452pwC (NVIDIA High Definition Audio)"     [1:
```

3. **Add channel** — Add an audio input channel to d for the microphone device.

```
addinput(d,"Audio1","1","Audio");
```

To display a summary of the DataAcquisition channels, type:

```
d.Channels

ans =

    Index     Type      Device     Channel    Measurement Type        Range         Name
    _____    ______    ________    _______    ________________    ______________    ____

      1      "audi"    "Audio1"      "1"          "Audio"         "-1.0 to +1.0"     ""
```

4. **Acquire data** — Start the acquisition. When all the data is acquired, it is assigned to `data`.

```
data = read(d,seconds(1));
plot(data)
```

5. **Clean up** — When you no longer need d, you should remove it from memory.

```
delete(d)
clear d
```

Generate Audio Data

If you have a sound card installed, you can run the following example, which outputs 1 second of data to two analog output hardware channels.

You should modify this example to suit your specific application needs.

1. **Create a DataAcquisition object** — Create the DataAcquisition object d for a sound card.

```
d = daq('directsound');
```

2 **Add channel** — Add an audio output channel to DataAcquisition `d`. This example uses the device ID `Audio4` for the speakers.

```
addoutput(d,"Audio4',"1","Audio");
```

To display a summary of the DataAcquisition and its channels, type:

```
d,d.Channels
```

3 **Output data** — Create 1 second of output data, and queue the data for output from the device. You queue a matrix with one column of data for each hardware channel.

```
data = sin(linspace(0,2*pi*500,44100)');
preload(d,data)
```

Start the output. When all the data is output, `d` stops generating.

```
start(d)
```

4 **Clean up** — When you no longer need `d`, you should remove it from memory and from the MATLAB workspace.

```
delete(d)
clear d
```

Acquire and Generate Digital Data

If you have a supported National Instruments board with at least two digital I/O ports, you can run the following example, which writes and reads digital values.

You should modify this example to suit your specific application needs. Adjust the example if the ports on your device do not support the input/output directions specified here.

1 **Create a DataAcquisition object** — Create the DataAcquisition interface `d` for a National Instruments board with hardware device ID `cDAQ1Mod1`.

```
s = daq("ni");
```

2 **Add digital input channels** — Add two lines from port 0 to `d`, and configure them for input.

```
addinput(d,"cDAQ1Mod1","Port0/Line0:1","Digital");
```

3 **Add digital output lines** — Add two lines from port 0 to `s`, and configure them for output.

```
addoutput(d,"cDAQ1Mod1","Port0/Line2:3","Digital");
```

To display a summary of the channels, type:

```
d.Channels
```

```
ans =

    Index    Type     Device         Channel        Measurement Type    Range          Name
    _____    _____    ___________    ____________    ________________    _____    __________________

      1      "dio"    "cDAQ1Mod1"    "port0/line0"    "InputOnly"        "n/a"    "Dev3_port0/line0"
      2      "dio"    "cDAQ1Mod1"    "port0/line1"    "InputOnly"        "n/a"    "Dev3_port0/line1"
      3      "dio"    "cDAQ1Mod1"    "port0/line2"    "OutputOnly"       "n/a"    "Dev3_port0/line2"
      4      "dio"    "cDAQ1Mod1"    "port0/line3"    "OutputOnly"       "n/a"    "Dev3_port0/line3"
```

4 **Add clock and trigger** — To synchronize operations, add a clock and trigger connection.

```
addclock(d,"ScanClock","External","cDAQ1/PFI0");
addtrigger(d,"Digital","StartTrigger","External","cDAQ1/PFI1");
d.Clocks,d.DigitalTriggers

ans = 

  Clock with properties:

         Source: 'External'
    Destination: 'cDAQ1/PFI0'
           Type: ScanClock


ans = 

  DigitalTrigger with properties:

         Source: 'External'
    Destination: 'cDAQ1/PFI1'
           Type: StartTrigger
      Condition: 'RisingEdge'
```

Note Digital line values are usually not transferred at a specific rate. Although some specialized boards support clocked I/O.

5 **Queue output data and start device** — Create an array of output values, and queue the values. Note that reading and writing digital I/O line values typically does not require that you configure specific property values.

```
preload(d,round(rand(4000,2)));
gval = start(d);
```

6 **Display input** — To read only the input lines, type:

```
gval
```

7 **Clean up** — When you no longer need `d`, you should remove it from memory and from the MATLAB workspace.

```
delete(d)
clear d
```

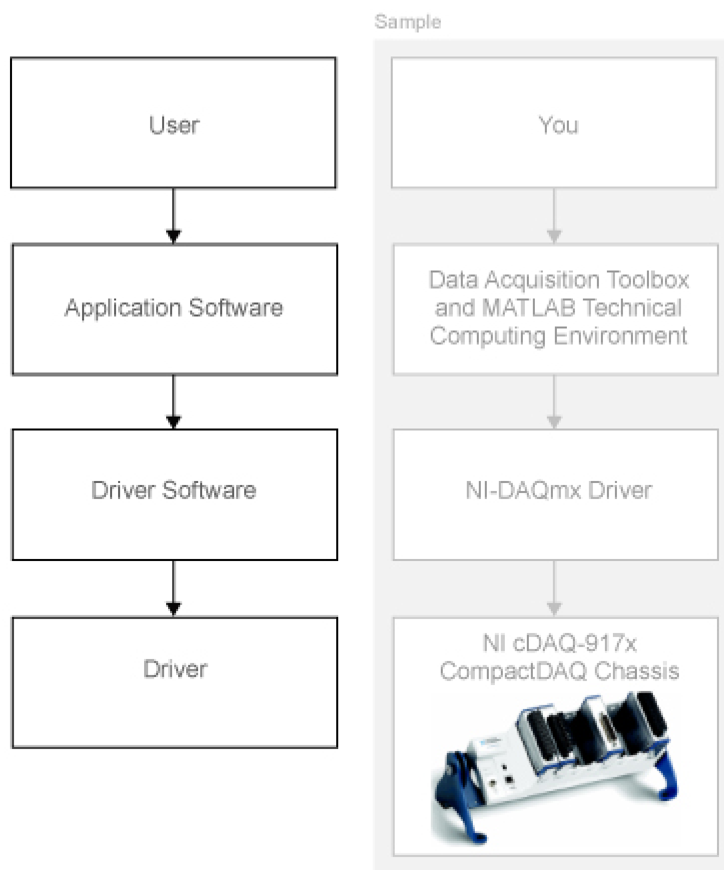
Introduction to the DataAcquisition Interface

- “The DataAcquisition Object” on page 3-2
- “Get Command-Line Help” on page 3-3

The DataAcquisition Object

The toolbox interface uses a DataAcquisition object that allows you to communicate easily with devices from National Instruments, Measurement Computing, Analog Devices, Microsoft Windows sound cards, and Digilent. You create a DataAcquisition using the `daq` function. A DataAcquisition represents one or more channels that you specify on data acquisition devices. You configure a DataAcquisition to acquire or generate data at a specific rate, based on the specified number of scans or the duration of the operation.

For an explanation of how this communication works, see Data Acquisition System on page 1-5. The relationship between you, the application software, the driver software, the chassis, and the devices is shown here.



For more information about creating a DataAcquisition, see "Create a DataAcquisition Interface" on page 4-5.

See Also

More About

- "Limitations by Vendor" on page B-2

Get Command-Line Help

To access command-line help for Data Acquisition Toolbox, type:

```
help daq
```

or

```
daqhelp
```

The Command Window displays links for the functions of the DataAcquisition interface.

To access command-line help for a particular function, type:

```
daqhelp function_name
```

For example,

```
daqhelp readwrite
```

You can get help on individual properties of the toolbox objects. For example, to see help on the `Channels` property of a `DataAcquisition` object, type:

```
help daq.interfaces.DataAcquisition.Channels
```

It can be easier to get function and property help if the object exists in the workspace. For example,

```
d = daq("ni");
help d.Rate
help d.addinput
```

Using the DataAcquisition Interface

- “Interface Workflow” on page 4-2
- “Digital Input and Output” on page 4-3
- “Discover Hardware Devices” on page 4-4
- “Create a DataAcquisition Interface” on page 4-5
- “Channel Properties” on page 4-7

Interface Workflow

In this section...
“Working a DataAcquisition” on page 4-2
“DataAcquisition Interface and Data Acquisition Toolbox” on page 4-2

Working a DataAcquisition

Use the DataAcquisition object to communicate with data acquisition devices, such as National Instruments devices including a CompactDAQ chassis.

Use the `daq` function to create a DataAcquisition interface.

You can also synchronize operations within the DataAcquisition. See “Synchronization” on page 13-2 for more information.

DataAcquisition Interface and Data Acquisition Toolbox

Data Acquisition Toolbox and the MATLAB technical computing environment use the DataAcquisition interface to communicate with devices of various vendors, such as National Instruments, including a CompactDAQ chassis. You can operate in the foreground, where the operation blocks MATLAB until complete, or in the background, where MATLAB continues to run additional MATLAB commands while the hardware operation proceeds.

You can create a DataAcquisition with both analog input and analog output channels and configure acquisition and generation simultaneously. See “Acquire Data and Generate Signals Simultaneously” on page 6-16 for more information.

See Also

More About

- “Transition Your Code from Session to DataAcquisition Interface” on page 14-2

Digital Input and Output

Digital subsystems transfer digital or logical values in bits via digital lines. You can perform clocked and non-clocked digital operations using the DataAcquisition interface in the Data Acquisition Toolbox.

For more information see “Digital Channels” on page 9-2.

Discover Hardware Devices

Discover the supported data acquisition devices on your system.

Step 1. Discover hardware devices.

```
dev  = daqlist
```

```
dev =

  4×5 table

      VendorID       DeviceID                 Description                              Model                      DeviceInfo
    _____________    ________    ____________________________________________    ______________________________    ____________________

    "ni"             "Dev2"      "National Instruments(TM) USB-6509"             "USB-6509"                        [1×1 daq.DeviceInfo]
    "ni"             "Dev3"      "National Instruments(TM) USB-6211"             "USB-6211"                        [1×1 daq.DeviceInfo]
    "directsound"    "Audio0"    "DirectSound Primary Sound Capture Driver"      "Primary Sound Capture Driver"    [1×1 daq.DeviceInfo]
    "directsound"    "Audio1"    "DirectSound Primary Sound Driver"              "Primary Sound Driver"            [1×1 daq.DeviceInfo]
```

Step 2. Get detailed device information.

View the `DeviceInfo` details for the `Dev3` device:

```
dev.DeviceInfo(2)
```

```
ans =

ni: National Instruments(TM) USB-6211 (Device ID: 'Dev3')
   Analog input supports:
      4 ranges supported
      Rates from 0.1 to 250000.0 scans/sec
      16 channels ('ai0' - 'ai15')
      'Voltage' measurement type

   Analog output supports:
      -10 to +10 Volts range
      Rates from 0.1 to 250000.0 scans/sec
      2 channels ('ao0','ao1')
      'Voltage' measurement type

   Digital IO supports:
      8 channels ('port0/line0' - 'port1/line3')
      'InputOnly','OutputOnly' measurement types

   Counter input supports:
      Rates from 0.1 to 80000000.0 scans/sec
      2 channels ('ctr0','ctr1')
      'EdgeCount','PulseWidth','Frequency','Position' measurement types

   Counter output supports:
      Rates from 0.1 to 80000000.0 scans/sec
      2 channels ('ctr0','ctr1')
      'PulseGeneration' measurement type
```

Create a DataAcquisition Interface

This example shows how to create a DataAcquisition interface and add channels to acquire and generate data. You can also configure DataAcquisition and channel properties needed for your operation.

Step 1. Find Devices for the Vendor.

```
daqlist("ni")
```

```
  2×4 table

    DeviceID               Description                  Model             DeviceInfo
    ________    __________________________________    __________    ______________________

     "Dev2"     "National Instruments(TM) USB-6509"    "USB-6509"    [1×1 daq.ni.DeviceInfo]
     "Dev3"     "National Instruments(TM) USB-6211"    "USB-6211"    [1×1 daq.ni.DeviceInfo]
```

Step 2. Create a DataAcquisition Object.

```
 d = daq("ni")
```

```
DataAcquisition using National Instruments(TM) hardware:

                     Running: 0
                        Rate: 1000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []
```

After you create a DataAcquisition object, add channels using the `addinput` and `addoutput` functions.

Step 3. Add Channels to the DataAcquisition.

Add an analog input channel, and view the DataAcquisition channel list:

```
addinput(d,"Dev3","ai0","Voltage")
d.Channels
```

```
    Index    Type    Device    Channel    Measurement Type          Range              Name
    _____    ____    ______    _______    ________________    _________________    __________

      1      "ai"    "Dev3"     "ai0"     "Voltage (Diff)"    "-10 to +10 Volts"    "Dev3_ai0"
```

Step 4. Change Channel Properties.

Change the channel `TerminalConfig` property to `'SingleEnded'`, and view the updated configuration:

```
d.Channels.TerminalConfig = "SingleEnded";
d.Channels
```

```
    Index    Type    Device    Channel       Measurement Type             Range              Name
    _____    ____    ______    _______    _____________________    _________________    __________
```

```
    1      "ai"    "Dev3"     "ai0"     "Voltage (SingleEnd)"    "-10 to +10 Volts"    "Dev3_ai0"
```

See Also

Related Examples

- “Acquire Counter Input Data” on page 8-3
- “Generate Pulse Data on a Counter Channel” on page 8-6

More About

- “Analog Input and Output”
- “Transition Your Code from Session to DataAcquisition Interface” on page 14-2

Channel Properties

Get Property Information

You can use the `get`, `set`, and `properties` functions to get information on channel object properties. For example, create a DataAcquisition and add a voltage measurement input channel, then view the channel properties:

```
d = daq("ni");
ch = addinput(d,"Dev1",1,"Voltage");
get(ch)
```

```
       Coupling: DC
  TerminalConfig: Differential
          Range: -10 to +10 Volts
           Name: 'Dev1_ai1'
             ID: 'ai1'
         Device: [1x1 daq.ni.DeviceInfo]
MeasurementType: 'Voltage'
```

View the channel settable properties and their acceptable values:

```
set(ch)
```

```
      Coupling: [ DC | AC ]
TerminalConfig: [ Differential | SingleEnded | SingleEndedNonReferenced | PseudoDifferential ]
         Range: -10 to +10 Volts
          Name: {}
```

Change the channel terminal configuration:

```
ch.TerminalConfig = "SingleEnded"
```

```
ch =

    Index    Type    Device    Channel      Measurement Type          Range              Name
    _____    ____    ______    _______    ____________________    _________________    __________

      1      "ai"    "Dev1"     "ai1"     "Voltage (SingleEnd)"   "-10 to +10 Volts"    "Dev1_ai1"
```

You can also request help in the Command Window on a specific property, providing either the object and property, or the class name and property. For example:

```
help ch.TerminalConfig
```

or

```
help ("daq.AnalogInputVoltageChannel.TerminalConfig")
```

```
 TerminalConfig The current input type (single ended/differential)
```

All Channels

All channel objects have these properties in common for all measurement types:

Property	Description	Values
Name	Channel name	character vector constructed of device ID and channel ID

Property	Description	Values

`ID`	Channel identifier corresponding to device terminal ID	character vector, for example:
		
		`'ai0'`
		`'ao3'`
		`'Port2/Line1'`
		`'ctr0'`

`Device`	`DeviceInfo` object for device with this channel	`DeviceInfo` object handle

`MeasurementType`	Type of measurement	character vector, for example:
		
		`'Voltage'`
		`'Current'`
		`'Thermocouple'`
		`'Digital'`
		`'Audio'`

Analog Input and Output Channels

- “Voltage Measurement” on page 4-8
- “Current Measurement” on page 4-8

Voltage Measurement

Input voltage measurement channel objects also include these properties:

Property	Description	Values
`Range`	Input value range	double values depending on measurement type and device support

`Coupling`	Coupling mode of the channel	character vector of:
		
		`'AC'`
		`'DC'`

`TerminalConfig`	Channel terminal configuration as described in “Channel Configuration” on page 1-18	character vector of:
		
		`'Differential'`
		`'SingleEnded'`
		`'SingleEndedNonReferenced'`
		`'PseudoDifferential'`

Current Measurement

Current measurement channel objects also include these properties:

Property	Description	Values
Current Input		
`Range`	Input value range	double values depending on measurement type and device support

Property	Description	Values

`Coupling`	Coupling mode of the channel	character vector of:
		
		`'AC'`
		`'DC'`

`TerminalConfig`	Channel terminal configuration as described in “Channel Configuration” on page 1-18	character vector of:
		
		`'Differential'`
		`'SingleEnded'`
		`'SingleEndedNonReferenced'`
		`'PseudoDifferential'`

Current Input and Output		

`ShuntLocation`	(Only some vendors) Indicates if the shunt resistor is located internally on the device or externally	character vector of:
		
		`'Internal'`
		`'External'`

`ShuntResistance`	(Only some vendors) Indicates shunt resistance in ohms	double

Other Analog Measurements

- “Thermocouple Measurement” on page 4-9
- “Accelerometer Measurement” on page 4-10
- “RTD Measurement” on page 4-10
- “Bridge Measurement” on page 4-11
- “Microphone Measurement” on page 4-12
- “IEPE Measurement” on page 4-13

Thermocouple Measurement

Thermocouple measurement input channel objects also include these properties:

Property	Description	Values

`ThermocoupleType`	Type of thermocouple based on temperature range and sensitivity, according to the NIST Thermocouple Types Definitions.	character vector of:
		
		`'J'`
		`'K'`
		`'N'`
		`'R'`
		`'S'`
		`'T'`
		`'B'`
		`'E'`

`Units`	Temperature units	character vector of:
		
		`'Celsius' (default)`
		`'Fahrenheit'`
		`'Kelvin'`
		`'Rankine'`

Property	Description	Values
Range	Input value range	double values depending on measurement type and device support

Accelerometer Measurement

Accelerometer measurement input channel objects also include these properties:

Property	Description	Values
Sensitivity	Sensitivity of accelerometer channel expressed as volts per g-force, V/*g*	double
ExcitationCurrent	Current to excite an IEPE accelerometer, IEPE microphone, generic IEPE sensor, or RTD, specified in amperes.	double
ExcitationSource	Indicates source of excitation for IEPE sensor or RTD	character vector of: 'Internal' 'External' 'None' 'Unknown'
Coupling	Coupling mode of the channel	character vector of: 'AC' 'DC'
TerminalConfig	Channel input configuration as described in “Channel Configuration” on page 1-18	character vector of: 'Differential' 'SingleEnded' 'SingleEndedNonReferenced' 'PseudoDifferential'
Range	Input value range	double values depending on measurement type and device support

RTD Measurement

RTD measurement input channel objects also include these properties:

Property	Description	Values
Units	Temperature units	character vector of: 'Celsius' (default) 'Fahrenheit' 'Kelvin' 'Rankine'

Property	Description	Values
RTDType	Specify the sensitivity of a standard RTD 100-ohm platinum sensor	character vector of: 'Pt3750' 'Pt3851' 'Pt3911' 'Pt3916' 'Pt3920' 'Pt3928'
RTDConfiguration	Specify the wiring configuration for measuring resistance	character vector of: 'TwoWire' 'ThreeWire' 'FourWire'
R0	Specify the resistance of this device to a reference temperature	double
ExcitationCurrent	Current to excite an IEPE accelerometer, IEPE microphone, generic IEPE sensor, or RTD, specified in amperes	double
ExcitationSource	Indicates source of excitation for IEPE sensor	character vector of: 'Internal' 'External' 'None' 'Unknown'
Coupling	Coupling mode of the channel	character vector of: 'AC' 'DC'
TerminalConfig	Channel input configuration as described in “Channel Configuration” on page 1-18	character vector of: 'Differential' 'SingleEnded' 'SingleEndedNonReferenced' 'PseudoDifferential'
Range	Input value range	double values depending on measurement type and device support

Bridge Measurement

Bridge measurement input channel objects also include these properties:

Property	Description	Values
`BridgeMode`	Bridge mode representing the active gauge of the analog input channel	character vector of: `'Full'` — All four gauges are active. `'Half'`— Only two bridges are active. `'Quarter'`— Only one bridge is active.

`ExcitationSource`	Indicates source of excitation voltage	character vector of:
		
		`'Internal'`
		`'External'`
		`'None'`
		`'Unknown'`

`ExcitationVoltage`	Indicates the excitation voltage value to apply to bridge measurements	double
`NominalBridgeResistance`	Resistance of a bridge-based sensor in ohms	double
`Range`	Range of input values	double values depending on measurement type and device support

Microphone Measurement

Microphone measurement input channel objects also include these properties:

Property	Description	Values
`Sensitivity`	Microphone channel sensitivity in volts per pascal, V/Pa	double
`MaxSoundPressureLevel`	Maximum sound pressure of the microphone channel in decibels	double
`ExcitationCurrent`	Current to excite an IEPE accelerometer, IEPE microphone, generic IEPE sensor, or RTD, specified in amperes.	double

`ExcitationSource`	Indicates source of excitation for IEPE sensor	character vector of:
		
		`'Internal'`
		`'External'`
		`'None'`
		`'Unknown'`

`Coupling`	Coupling mode of the channel	character vector of:
		
		`'AC'`
		`'DC'`

Property	Description	Values
TerminalConfig	Channel input configuration as described in “Channel Configuration” on page 1-18	character vector of: 'Differential' 'SingleEnded' 'SingleEndedNonReferenced' 'PseudoDifferential'
Range	Input value range	double values depending on measurement type and device support

IEPE Measurement

IEPE measurement input channel objects also include these properties:

Property	Description	Values
ExcitationCurrent	Current to excite an IEPE accelerometer, IEPE microphone, generic IEPE sensor, or RTD, specified in amperes.	double
ExcitationSource	Indicates source of excitation for IEPE sensor	character vector of: 'Internal' 'External' 'None' 'Unknown'
Coupling	Coupling mode of the channel	character vector of: 'AC' 'DC'
TerminalConfig	Channel input configuration as described in “Channel Configuration” on page 1-18	character vector of: 'Differential' 'SingleEnded' 'SingleEndedNonReferenced' 'PseudoDifferential'
Range	Input value range	double values depending on measurement type and device support

Digital Channels

Digital channel objects include the following properties:

Property	Description	Values
Direction	Direction of data flow, changeable only for bidirectional channels	character vector of: 'Input' 'Output'

Counter Channels

- “All Counter Channels” on page 4-14
- “Edge Count” on page 4-14
- “Frequency” on page 4-14
- “Position” on page 4-14
- “Pulse Width” on page 4-15
- “Pulse Generation” on page 4-15

All Counter Channels

Counter input and output channel objects also include these properties:

Edge Count

Counter input edge count channels also include the following properties:

Property	Description	Values

`ActiveEdge`	Indicates rising or falling edge of edge count signal	character vector of:
		
		`'Rising'`
		`'Falling'`

`CountDirection`	Indicates counting up or down	character vector of:
		
		`'Increment'`
		`'Decrement'`

`InitialCount`	Value to count from	uint32
`Terminal`	Terminal on device	character vector, for example `'PFI2'`

Frequency

Counter input frequency measurement channels also include the following properties:

Property	Description	Values

`ActiveEdge`	Indicates rising or falling edge of edge count signal	character vector of:
		
		`'Rising'`
		`'Falling'`

`Terminal`	Terminal on device	character vector, for example: `'PFI2'`

Position

For an overview of position measurement, including signals, encoding types, and Z-indexing, see National Instruments Encoder Measurements: How-To Guide. See also “Measure Angular Position with an Incremental Rotary Encoder” on page 18-110.

Counter input position measurement channels also include the following properties:

Property	Description	Values
EncoderType	Specify the encoding type of the counter input	character vector of: 'X1' 'X2' 'X4' 'TwoPulse'
ZResetEnable	Allow the Z-indexing to be reset	logical
ZResetValue	Specify the reset value for Z-indexing on a counter input	numeric
ZResetCondition	Specify reset conditions for Z-indexing of counter	character vector of: 'AHigh' 'BHigh' 'BothLow' 'BothHigh'
InitialCount	Specify the point from which the device starts the counter	double value, typically 0
TerminalA	External terminal on device	character vector, for example: 'PFI0'
TerminalB	External terminal on device	character vector, for example: 'PFI1'
TerminalZ	External index terminal on device for zero or reference signal	character vector, for example: 'PFI2'

Pulse Width

Counter input pulse width measurement channels also include the following properties:

Property	Description	Values
ActivePulse	Indicates active level	character vector of: 'High' 'Low'
Terminal	External terminal on device	character vector, for example: 'PFI2'

Pulse Generation

Counter output pulse generation channels also include the following properties:

Property	Description	Values
IdleState	Indicate the default state of the counter output channel when not running	character vector of: 'High' 'Low'

Property	Description	Values
InitialDelay	Specify an initial delay on the counter output channel before pulse generation	double value in seconds
Frequency	Specify the pulse repetition rate of a counter output channel	double value in Hz
DutyCycle	Specify the fraction of time that the generated pulse is in active state, as a portion of 1.0. A square wave has a duty cycle of 0.5.	double
Terminal	External terminal on device	character vector, for example, 'PFI2'

Audio Channels

Audio input and output channel objects also include these properties:

Property	Description	Values
Range	Input/output value range	-1.0 to +1.0

Function Generator Channels

Function generator channel objects also include these properties:

Property	Description	Values
Range	Output value range	double values depend on device support
TerminalConfig	Channel input configuration as described in “Channel Configuration” on page 1-18	character vector of: 'Differential' 'SingleEnded' 'SingleEndedNonReferenced' 'PseudoDifferential'
Gain	Specify amplification of scan data for channel output.	double value between –5 and 5. Ensure that Gain x data + Offset falls within the valid Range of device output.
Offset	Specify offset of scan data for channel output.	double value between –5 and 5. Ensure that Gain x data + Offset falls within the valid Range of device output.
Frequency	Specify waveform frequency	double value in Hz, within FrequencyLimit value
Phase	Specify waveform phase shift in degrees	double, from 0 to 360

Property	Description	Values
WaveformType	Specify waveform shape	character vector of: 'Sine' 'Square' 'Triangle' 'RampUp' 'RampDown' 'DC' 'Arbitrary'
FrequencyLimit	Minimum and maximum rates that the function generation channel supports	double

See Also

Functions
addinput | addoutput | addbidirectional

Support Package Installer

Install Hardware Support Package for Vendor Support

In this section...
"Install Support Packages" on page 5-2
"Update or Uninstall Support Packages" on page 5-2

To communicate with a data acquisition device, you need to install the required support package on your system for the device vendor. Data Acquisition Toolbox support packages are available for the following vendors:

- Analog Devices (ADALM1000)
- Digilent (Analog Discovery)
- Measurement Computing
- Microsoft (Windows Sound cards)
- National Instruments (NI-DAQmx)

Install Support Packages

To install the required support package for a specific vendor and device:

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
2. In the left pane of the Add-On Explorer, scroll to **Filter by Type** and check **Hardware Support Packages**.
3. Under **Filter by Vendor** check the vendor of your device. The Add-On Explorer displays support packages for that vendor. Click the support package for your vendor and device.
4. Click **Install > Install**. Sign in to your MathWorks® account if necessary, and proceed.

Update or Uninstall Support Packages

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

See Also

More About

- "Get and Manage Add-Ons"

Analog Input and Output

- “Acquire Data in the Foreground” on page 6-2
- “Acquire Data from Multiple Channels” on page 6-3
- “Acquire Data in the Background with Live Plot Updates” on page 6-4
- “Acquire Bridge Measurements” on page 6-5
- “Acquire Sound Pressure Data” on page 6-7
- “Acquire IEPE Data” on page 6-9
- “Generate Signals in the Foreground” on page 6-11
- “Generate Signals on Multiple Channels” on page 6-12
- “Generate Signals in the Background” on page 6-13
- “Generate Signals in the Background Continuously” on page 6-14
- “Acquire Data and Generate Signals Simultaneously” on page 6-16
- “Acquire Data with Analog Input Recorder” on page 6-17
- “Generate Signals with Analog Output Generator” on page 6-21

Acquire Data in the Foreground

This example shows how to acquire voltage data from an NI 9205 device with ID `cDAQ1Mod1`.

Create a DataAcquisition object and save it to the variable, `d`:

```
d = daq("ni")
```

```
d =

DataAcquisition using National Instruments(TM) hardware:

                     Running: 0
                        Rate: 1000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []
```

By default, the acquisition is configured to acquire at the rate of 1000 scans per second.

Add an analog input channel for voltage measurement, using the device channel `ai0`:

```
addinput(d,"cDAQ1Mod1","ai0","Voltage");
```

Acquire data for 2 seconds and store it in the variable, `data`, then plot it:

```
data = read(d,seconds(2),"OutputFormat","Matrix");
plot(data)
```

Specify an acquisition of 4096 scans of data. Changing the number of scans changes the duration of the acquisition to 4.096 seconds at the default rate of 1000 scans per second.

Acquire the data and store it in the variable `data`, and then plot it:

```
data = read(d,4096,"OutputFormat","Matrix");
plot(data)
```

See Also

Related Examples

- “Acquire Data in the Background with Live Plot Updates” on page 6-4

Acquire Data from Multiple Channels

This example shows how to acquire data from multiple channels, and from multiple devices on the same chassis. In this example, you acquire voltage data from an NI 9201 device with ID `cDAQ1Mod4` and an NI 9205 device with ID `cDAQ1Mod1`.

Create a DataAcquisition object and add two analog input voltage channels for `cDAQ1Mod1` with channel IDs 0 and 1:

```
d = daq("ni");
addinput(d,"cDAQ1Mod1",0:1,"Voltage")
```

```
ch =

    Index    Type      Device      Channel    Measurement Type          Range              Name
    _____    ____    ___________    _______    ________________    _________________    ______________

      1      "ai"    "cDAQ1Mod1"    "ai0"      "Voltage (Diff)"    "-10 to +10 Volts"    "cDAQ1Mod1_ai0"
      2      "ai"    "cDAQ1Mod1"    "ai1"      "Voltage (Diff)"    "-10 to +10 Volts"    "cDAQ1Mod1_ai1"
```

Add an additional channel for a separate device, `cDAQ1Mod6` with channel ID 0. For NI devices, use either a terminal name, like `ai0`, or a numeric equivalent like `0`. Then view all channels on the DataAcquisition.

```
ch = addinput(d,"cDAQ1Mod6","ai0","Voltage");
d.Channels
```

```
    Index    Type      Device      Channel    Measurement Type          Range              Name
    _____    ____    ___________    _______    ________________    _________________    ______________

      1      "ai"    "cDAQ1Mod1"    "ai0"      "Voltage (Diff)"    "-10 to +10 Volts"    "cDAQ1Mod1_ai0"
      2      "ai"    "cDAQ1Mod1"    "ai1"      "Voltage (Diff)"    "-10 to +10 Volts"    "cDAQ1Mod1_ai1"
      3      "ai"    "cDAQ1Mod6"    "ai0"      "Voltage (Diff)"    "-10 to +10 Volts"    "cDAQ1Mod6_ai0"
```

Acquire one second of data and store it in the variable `data`, and then plot it:

```
data = read(d,seconds(1),"OutputFormat","Matrix");
plot(data)
```

Change the properties of the channel `ai0` on `cDAQ1Mod6` and display `ch`:

```
ch.TerminalConfig ="SingleEnded";
ch.Name = "Velocity sensor";
ch
```

```
ch =

 Index    Type      Device      Channel      Measurement Type           Range               Name
 _____    ____    ___________    _______    ____________________    _________________    ________________

   1      "ai"    "cDAQ1Mod6"    "ai0"      "Voltage (SingleEnd)"    "-10 to +10 Volts"    "Velocity sensor"
```

Acquire the data and store it in the variable, `data`, and plot it:

```
data = read(d,seconds(1),"OutputFormat","Matrix");
plot(data)
```

See Also

Related Examples

- “Acquire Data in the Foreground” on page 6-2
- “Acquire Data in the Background with Live Plot Updates” on page 6-4

Acquire Data in the Background with Live Plot Updates

This example shows how to acquire data in the background using callbacks while MATLAB continues to run.

A background acquisition uses callbacks to allow your code to access data as the hardware acquires it or to react to any errors as they occur. In this example, you acquire data from a microphone with ID `Audio1` using the `ScansAvailableFcnCount` property to trigger the function call defined by the `ScansAvailableFcn` property. Using a callback allows a plot to be updated in real time while acquisition continues.

Get a list of devices so you can identify the microphone you want to use. The partial listing here indicates the device ID.

```
daqlist
```

```
     VendorID        DeviceID                    Description
    _____________    ________    _____________________________________________________

    "directsound"    "Audio1"    "DirectSound Headset Microphone (Plantronics BT600)"
```

Create a `directsound` DataAcquisition object with a microphone input channel on `Audio1`. You might have to use a different device.

```
d = daq("directsound");
ch = addinput(d,"Audio1",1,"audio");
```

Create a simple callback function to plot the acquired data and save it as `plotMyData.m` in your current working folder. Enter the following code in the file.

```
function plotMyData(obj,evt)
% obj is the DataAcquisition object passed in. evt is not used.
     data = read(obj,obj.ScansAvailableFcnCount,"OutputFormat","Matrix");
     plot(data)
 end
```

Set the callback function property of the DataAcquisition object to use your function.

```
d.ScansAvailableFcn = @plotMyData;
```

Start the acquisition to run for 5 seconds in the background.

```
start(d,"Duration",seconds(5))
```

Speak into the microphone and watch the plot. It updates 10 times per second.

See Also

Functions
`daqlist` | `daq` | `addinput` | `start`

Related Examples

- “Acquire Data in the Foreground” on page 6-2

Acquire Bridge Measurements

This example shows how to acquire and plot data from an NI USB-9219 device. The device ID is `cDAQ1Mod7`.

Create a DataAcquisition object assigned to the variable `d`:

```
d = daq("ni");
```

Add an analog input channel for `Bridge` measurement type, assigned to the variable `ch`:

```
ch = addinput(d,"cDAQ1Mod7","ai1","Bridge");
```

You might see this warning:

```
Warning: The Rate property was reduced to 2 due to the default ADCTimingMode of this device,
which is 'HighResolution'.
To increase rate, change ADCTimingMode on this channel to 'HighSpeed'.
```

To allow a higher acquisition rate, change the channel `ADCTimingMode` to `'HighSpeed'`:

```
ch.ADCTimingMode = "HighSpeed"
```

You might see this warning:

```
Warning: This property must be the same for all channels on this device.  All channels
associated with this device were updated.
```

Change the acquisition rate to `10` scans per second.

```
d.Rate = 10;
```

Set the channel `BridgeMode` to `'Full'`, which uses all four resistors in the device to acquire the voltage values:

```
ch.BridgeMode = "Full"
```

```
ch =

Data acquisition analog input channel 'ai1' on device 'cDAQ1Mod7':

             BridgeMode: Full
       ExcitationSource: Internal
      ExcitationVoltage: 2.5
NominalBridgeResistance: 'Unknown'
                  Range: -0.063 to +0.063 VoltsPerVolt
                   Name: empty
                     ID: 'ai1'
                 Device: [1x1 daq.ni.CompactDAQModule]
        MeasurementType: 'Bridge'
          ADCTimingMode: HighSpeed
```

Set the resistance of the bridge device to 350 ohms:

```
ch.NominalBridgeResistance = 350
```

```
ch =

Data acquisition analog input channel 'ai1' on device 'cDAQ1Mod7':

             BridgeMode: Full
       ExcitationSource: Internal
      ExcitationVoltage: 2.5
NominalBridgeResistance: 350
                  Range: -0.063 to +0.063 VoltsPerVolt
```

```
                  Name: empty
                    ID: 'ai1'
                Device: [1x1 daq.ni.CompactDAQModule]
       MeasurementType: 'Bridge'
         ADCTimingMode: HighSpeed
```

Save the acquired data to a variable and start the acquisition:

```
data = read(d,seconds(1),"OutputFormat","Matrix")
```

Plot the acquired data:

```
plot(data)
```

See Also

Related Examples

- “Acquire Data in the Foreground” on page 6-2
- “Acquire Data in the Background with Live Plot Updates” on page 6-4

Acquire Sound Pressure Data

This example shows how to acquire sound data from an NI 9234. The device is in an NI cDAQ-9178 chassis, in slot 3 with ID `cDAQ1Mod3`.

Create a DataAcquisition object, and add an analog input channel with `Microphone` measurement type:

```
d = daq('ni');
ch = addAnalogInputChannel(d,"cDAQ1Mod3",0,"Microphone");
```

Set the channel sensitivity to 0.037 v/pa.

```
ch.Sensitivity = 0.037
```

```
ch =

Data acquisition analog input microphone channel 'ai0' on device 'cDAQ1Mod3':

          Sensitivity: 0.037
MaxSoundPressureLevel: 136
    ExcitationCurrent: 0.002
     ExcitationSource: Internal
             Coupling: AC
       TerminalConfig: PseudoDifferential
                Range: -135 to +135 Pascals
                 Name: ''
                   ID: 'ai0'
               Device: [1x1 daq.ni.CompactDAQModule]
      MeasurementType: 'Microphone'
        ADCTimingMode: ''
```

Change the maximum sound pressure level to 100 dB.

```
ch.MaxSoundPressureLevel = 100
```

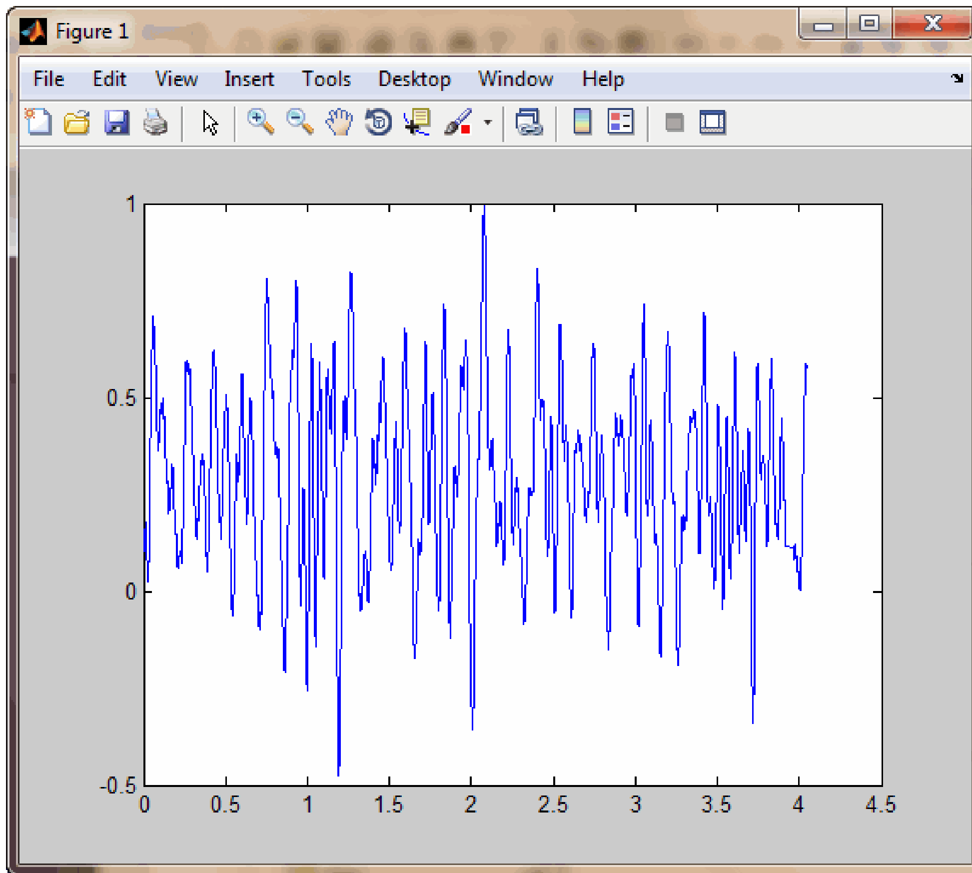
```
ch =

Data acquisition analog input microphone channel 'ai0' on device 'cDAQ1Mod3':

          Sensitivity: 0.037
MaxSoundPressureLevel: 100
    ExcitationCurrent: 0.002
     ExcitationSource: Internal
             Coupling: AC
       TerminalConfig: PseudoDifferential
                Range: -135 to +135 Pascals
                 Name: ''
                   ID: 'ai0'
               Device: [1x1 daq.ni.CompactDAQModule]
      MeasurementType: 'Microphone'
        ADCTimingMode: ''
```

Acquire 4 seconds of data and save it in a variable.

```
[data,time] = read(d,seconds(4),"OutputFormat","Matrix");
```

Plot the data.

```
plot(time,data)
```



See Also

Related Examples

- “Acquire Data in the Foreground” on page 6-2
- “Acquire Data in the Background with Live Plot Updates” on page 6-4

Acquire IEPE Data

This example shows how to acquire IEPE data using an NI 9234. The device is in an NI cDAQ-9178 chassis in slot 3 with ID `cDAQ1Mod3`.

Create a `DataAcquisition` object, and add an analog input channel with IEPE measurement type.

```
d = daq("ni");
ch = addinput(d,"cDAQ1Mod3",0,"IEPE");
```

Change the channel `ExcitationCurrent` property value to 0.004 amperes.

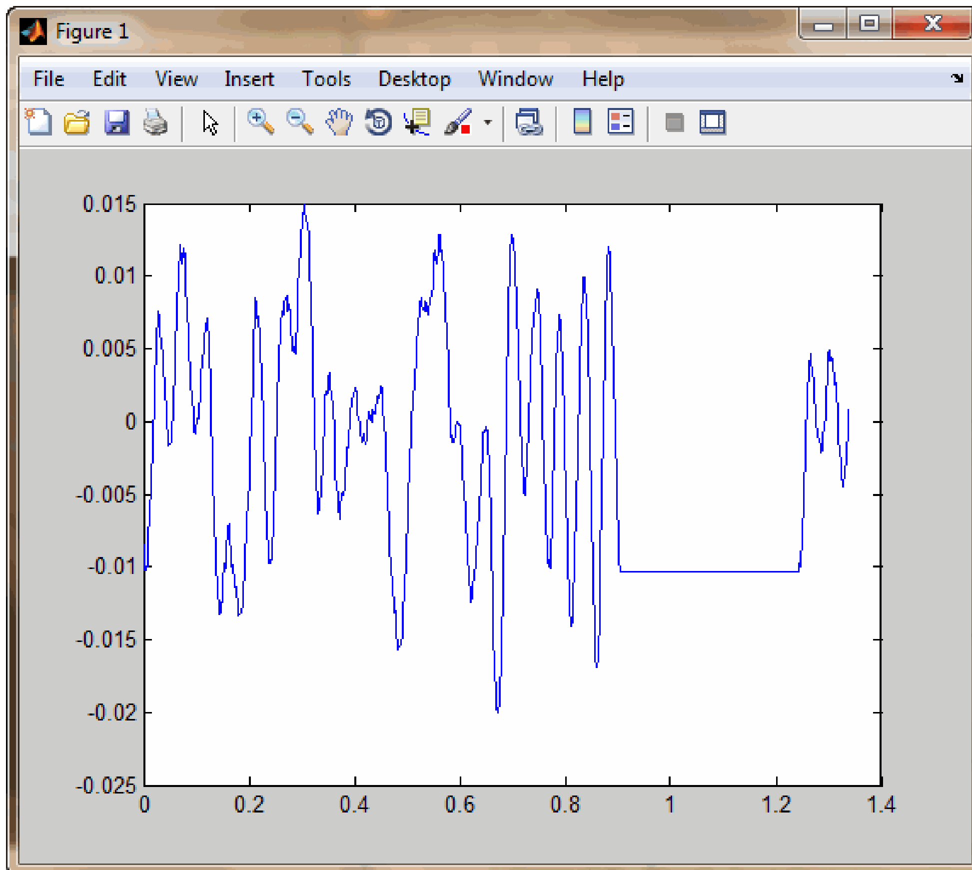
```
ch.ExcitationCurrent = .004;
```

Acquire the data and the corresponding sample times, storing them in two vectors.

```
[data,time] = read(d,seconds(1.35),"OutputFormat","Matrix");
```

Plot the data.

```
plot(time,data)
```



See Also

Related Examples

- "Acquire Data in the Foreground" on page 6-2
- "Acquire Data in the Background with Live Plot Updates" on page 6-4

Generate Signals in the Foreground

This example shows how to generate data using an NI 9263 device with ID `cDAQ1Mod2`.

Create a DataAcquisition object assigned to the variable `d`:

```
d = daq("ni");
```

Change the scan rate of the DataAcquisition to generate 10,000 scans per second:

```
d.Rate = 10000
```

```
d =

DataAcquisition using National Instruments(TM) hardware:

                     Running: 0
                        Rate: 10000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []
```

Add an analog output Voltage channel:

```
ch = addoutput(d,"cDAQ1Mod2",0,"Voltage");
```

You can specify the channel ID on NI devices using a terminal name, like `'ao1'`, or a numeric equivalent like `1`.

Create the data to define the output signal being generated. The output scans of one channel are defined by a column vector.

```
outputData = linspace(-1, 1, 2200)';
```

Generate the output signal. The output signal will have a duration of 0.22 seconds, based on the length of the queued data and the specified scan rate. MATLAB waits for this foreground generation, and returns when the generation is complete.

```
write(d,outputData)
```

See Also

Related Examples

- “Generate Signals in the Background” on page 6-13

Generate Signals on Multiple Channels

This example shows how to generate data from multiple channels and multiple devices. The example generates data using channels from an NI 9263 voltage device with ID `cDAQ1Mod2`, and an NI 9265 current device with ID `cDAQ1Mod8`.

Create an NI DAtaAcquisition object and add two analog output voltage channels from `cDAQ1Mod2`:

```
d = daq("ni");
addoutput(d, "cDAQ1Mod2', 2:3, "Voltage");
```

Add one output current channel from `cDAQ1Mod8`:

```
addoutput(d, "cDAQ1Mod8", "ao2", "Current");
```

Specify the channel ID on NI devices using a terminal name, like `ao1`, or a numeric equivalent like `1`.

Create a set of 1000 scans of data to output for all channels. Each channel output data is defined by a column in the 1000-by-3 data matrix.

```
outputData(:,1) = linspace(-1,1,1000)';
outputData(:,2) = linspace(-2,2,1000)';
outputData(:,3) = linspace(0,0.02,1000)';
```

Generate the output signals from the data matrix.

```
write(d,outputData);
```

See Also

Related Examples

- “Generate Signals in the Foreground” on page 6-11
- “Generate Signals in the Background” on page 6-13

Generate Signals in the Background

This example shows how to generate signals in the background while MATLAB continues to run.

Create an NI DataAcquisition object and add an analog output voltage channel from `cDAQ1Mod2`:

```
d = daq("ni");
addoutput(d,"cDAQ1Mod2","ao0","Voltage");
```

Specify the channel ID on NI devices using a terminal name, like `'ao1'`, or a numeric equivalent like `1`.

Create the data to output:

```
outputData = (linspace(-1,1,5000)');
```

In this case, 5000 scans will run for 5 seconds.

Queue the output data:

```
preload(d,outputData);
```

Start signal output generation:

```
start(d);
```

You can execute other MATLAB commands while the generation is in progress. In this example, call `pause`, which causes the MATLAB command line to wait for you to press any key.

```
pause
```

See Also

Related Examples

- “Generate Signals in the Foreground” on page 6-11

Generate Signals in the Background Continuously

This example shows how to continuously generate signals. A continuous background generation depends on callbacks to enable continuous queuing of data and to react to any errors as they occur. In this example, you generate from an NI 9263 device with ID `cDAQ1Mod2`.

A callback function is configured to run when a certain number of scans are required.

Create an NI DataAcquisition object and add an analog output voltage channel on `cDAQ1Mod2`:

```
d = daq("ni");
addoutput(d,"cDAQ1Mod2","ao0","Voltage");
```

Specify the channel ID on NI devices using a terminal name, like `'ao1'`, or a numeric equivalent like `1`.

Queue a column of output data.

```
preload(d,linspace(1,10,1000)');
```

Create a simple callback function to load data 1000 samples at a time. Save the function file as `loadMoreData.m` in your working folder:

```
function loadMoreData(obj,evt)
    % obj is the DataAcquisition object passed in. evt is not used.
    write(obj,linspace(1,10,1000)');
end
```

Define the `ScansRequiredFcn` to call your function `loadMoreData`:

```
d.ScansRequiredFcn = @loadMoreData;
```

This callback is executed whenever the number of queued scans falls below the threshold defined by the property `ScansRequiredFcnCount`. The default threshold is defined at 0.5 seconds of data at the default scan rate. In other words, with a default `Rate` at 1000 scans per second, the default `ScansRequiredFcnCount` value is 500. As your device generates an output signal, when the queued data falls below 500 scans, it triggers the `ScansRequiredFcn`.

```
d.ScansRequiredFcnCount
```

```
ans =

    500
```

Generate the continuous output signal:

```
start(d,"Continuous")
```

You can execute other MATLAB commands while the generation is in progress. In this example, issue a `pause`, which causes the MATLAB command line to wait for you to press any key.

```
pause
```

Tip If you want to continuously generate a repeating or periodic output, preload the waveform data, and use

```
start(d,"RepeatOutput")
```

See Also

Related Examples

- "Generate Signals in the Background" on page 6-13

Acquire Data and Generate Signals Simultaneously

This example shows how to acquire data with an NI 9205 device of ID `cDAQ1Mod1`, while generating signals from an NI 9263 device with ID `cDAQ1Mod2`.

You can acquire data and generate signals at the same time, on devices on the same chassis. When the DataAcquisition contains output channels, the duration of a finite generation and acquisition depends on the number of scans and the scan rate.

Create an NI DataAcquisition object and add one analog input channel on `cDAQ1Mod1` and one analog output channel on `cDAQ1Mod2`:

```
d = daq("ni");
addinput(d,"cDAQ1Mod1","ai0","Voltage");
addoutput(d,"cDAQ1Mod2","ao0","Voltage");
d.Channels
```

```
ans =

  Index    Type      Device       Channel      Measurement Type            Range               Name
  _____    ____    ___________    _______    _____________________    ________________    _____________

    1      "ai"    "cDAQ1Mod1"     "ai0"     "Voltage (SingleEnd)"    "-10 to +10 Volts"    "cDAQ1Mod1_ai0"
    1      "ao"    "cDAQ1Mod2"     "ao0"     "Voltage (SingleEnd)"    "-10 to +10 Volts"    "cDAQ1Mod2_ao0"
```

Define the output signal data for 2500 scans:

```
outData = linspace(-1,10,2500)';
```

The generated output signal of 2500 scans will run for 2.5 seconds at a scan rate of 1000 samples per second.

Generate the output signal and acquire the input data:

```
inData = readwrite(d,outData,"OutputFormat","Matrix");
plot(inData)
```

See Also

Related Examples

- “Generate Signals in the Foreground” on page 6-11
- “Generate Signals in the Background” on page 6-13
- “Acquire Data in the Foreground” on page 6-2
- “Acquire Data in the Background with Live Plot Updates” on page 6-4

Acquire Data with Analog Input Recorder

This topic shows how to use the **Analog Input Recorder** app to view and record data from an NI USB-6211 device.

To open the **Analog Input Recorder**, on the MATLAB Toolstrip, on the **Apps** tab, in the **Test and Measurement** section, click **Analog Input Recorder**.

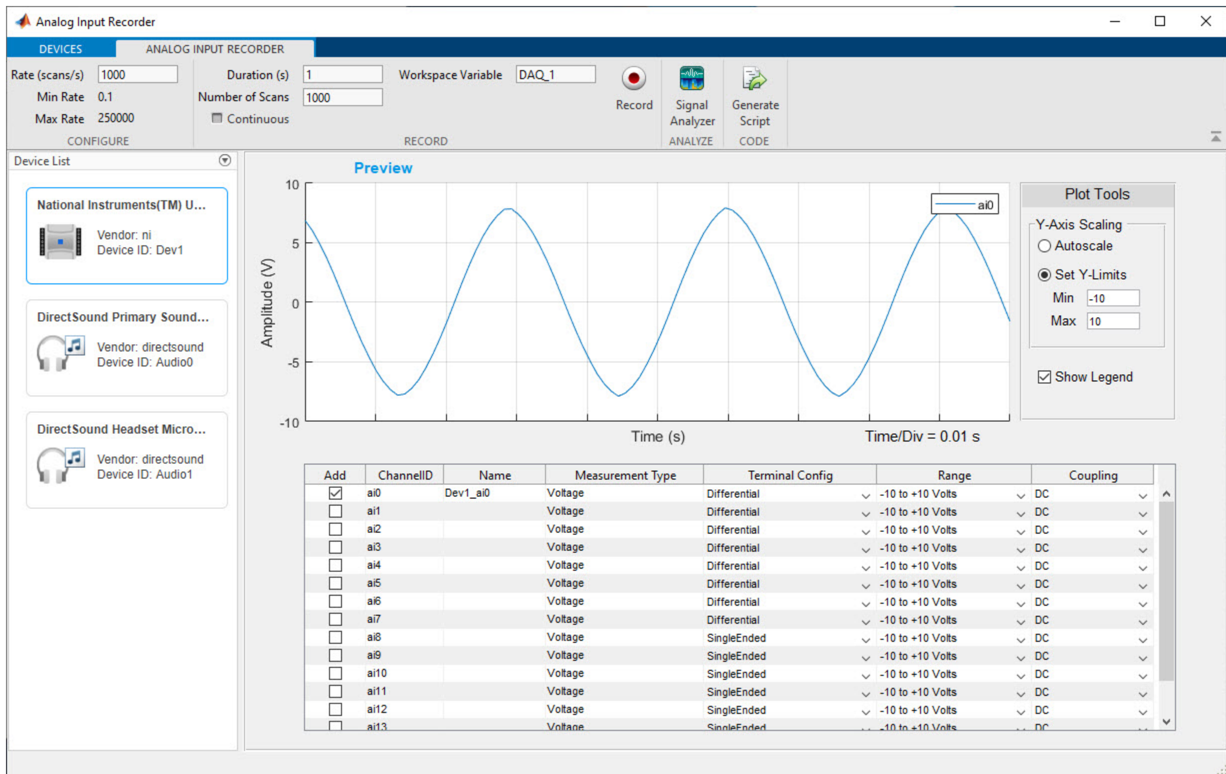
Upon opening, the **Analog Input Recorder** attempts to find all your attached analog and audio input devices.

Note Opening the **Analog Input Recorder** deletes all your existing DataAcquisition interfaces in MATLAB.

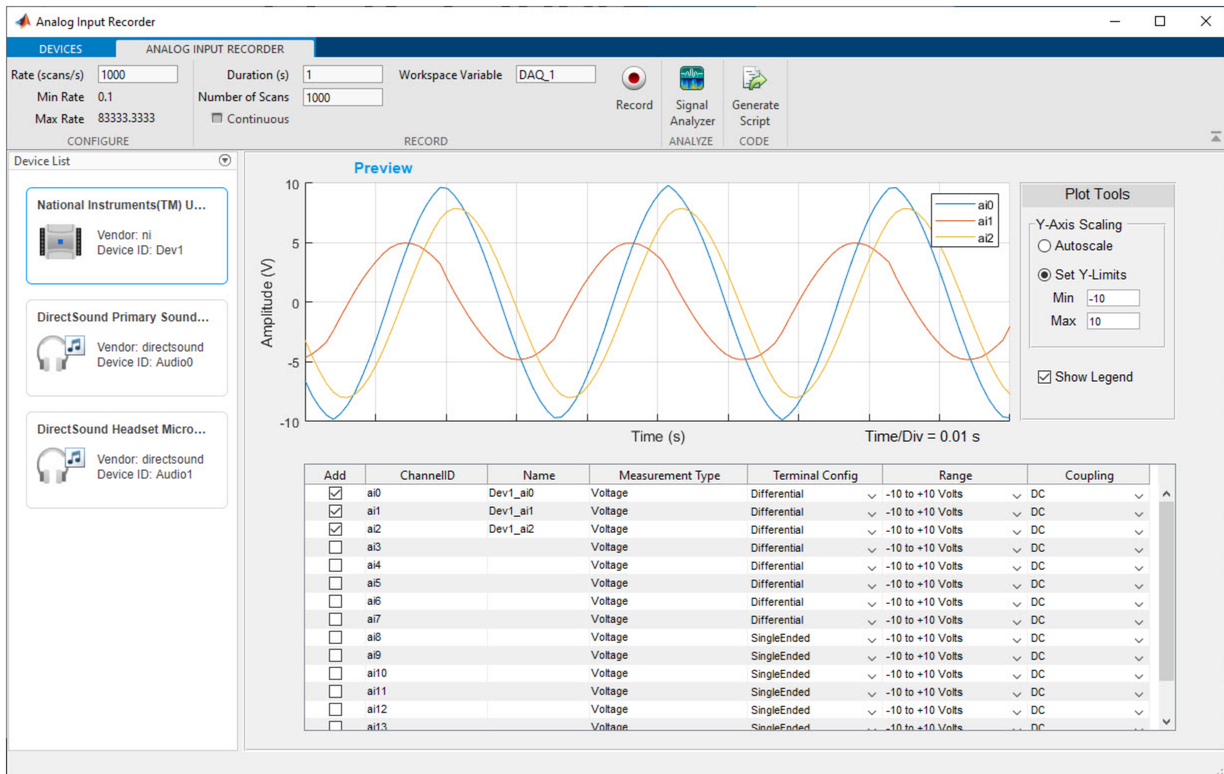
The DataAcquisition interface created by the **Analog Input Recorder** is not accessible from the MATLAB command line.

If you plug in a device while the app is open, you must refresh the listing for access to that device. On the **Devices** tab, click **Refresh**. Use the same procedure to remove a device from the listing after unplugging it.

Select the device you want to use in the **Device List**. The app immediately starts a preview of the analog input using default settings.



Modify any scan and channel settings for your specific needs. The following image shows the app displaying three channels of the device. Notice that the **Max Rate** value has changed with the number of channels; this relationship depends on the device.



Set values for **Number of Scans** or **Duration**, and **Rate**.

Check **Continuous** if you want to override the duration or number of scans. In this mode, recording continues until you explicitly stop it.

When you are ready to start recording data, click **Record**.

When recording is complete, either because the specified number of scans is recorded or you click **Stop**, the recorded data is assigned to the indicated MATLAB Workspace variable. By default, the variable starts as `DAQ_1`, and its name is incremented with every recording, but you can specify any valid MATLAB variable name not already in use. The variable is assigned an M-by-N timetable, where M table rows is the number of scans and N columns is the number of channels.

The following commands show the beginning of the acquired timetable for a multiple channel recording.

```
whos
```

```
  Name           Size              Bytes  Class        Attributes

  DAQ_1       1000x3               33315  timetable
```

View the first four rows of the timetable.

```
DAQ_1(1:4,:)
```

```
ans =
  4×3 timetable

      Time        Dev1_ai0     Dev1_ai1     Dev1_ai2
```

```
    _________    ________    ________    ________

    0 sec          4.0578     -1.9676      5.1516
    0.001 sec      2.8081     -2.5671      4.3738
    0.002 sec      1.4604     -3.0992      3.4339
    0.003 sec    0.029896     -3.5211      2.3651
```

The timestamp elements of the table are relative to the first scan. The absolute time of the first scan is available in the timetable `TriggerTime` custom property. For example,

```
DAQ_1.Properties.CustomProperties.TriggerTime
```

```
  datetime

   19-Nov-2019 15:21:01.239
```

In the **Analog Input Recorder**, click **Generate Script** for the app to open the MATLAB editor and display the equivalent code for recording data. The following code is generated for the finite (non-continuous) 3-channel recording of this example.

Create DataAcquisition Object

Create a DataAcquisition object for the specified vendor.

```
d = daq("ni");
```

Add Channels

Add channels and set channel properties, if any.

```
addinput(d,"Dev1","ai0","Voltage");

addinput(d,"Dev1","ai1","Voltage");

addinput(d,"Dev1","ai2","Voltage");
```

Read Data

Read the data in timetable format.

```
DAQ_1 = read(d,seconds(1))
```

Plot Data

Plot the read data on labeled axes.

```
plot(DAQ_1.Time, DAQ_1.Variables)
xlabel("Time")
ylabel("Amplitude (V)")
legend(DAQ_1.Properties.VariableNames)
```

Clean Up

Clear all DataAcquisition and channel objects.

```
clear d
```

See Also

Apps
Analog Input Recorder | Analog Output Generator

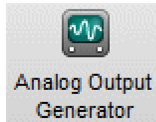
More About

- "Generate Signals with Analog Output Generator" on page 6-21
- "Timetables"

Generate Signals with Analog Output Generator

This topic shows how to use the **Analog Output Generator** app to define and generate signals from an audio device.

To open the **Analog Output Generator**, on the MATLAB Toolstrip, on the **Apps** tab, in the **Test and Measurement** section, click **Analog Output Generator**.



Upon opening, the **Analog Output Generator** attempts to find all your attached analog and audio output devices.

Note Opening the **Analog Output Generator** deletes all your existing DataAcquisition interfaces in MATLAB.

The DataAcquisition interface created by the **Analog Output Generator** is not accessible from the MATLAB command line.

If you plug in a device while the app is open, you must refresh the listing for access to that device. On the **Devices** tab, click **Refresh**. Use the same procedure to remove a device from the listing after unplugging it.

Select the device you want to use in the **Device List**. By default, the app immediately displays a preview of a test signal.

Use the following steps to produce an audio output of the "Hallelujah" chorus from Handel's *Messiah*.

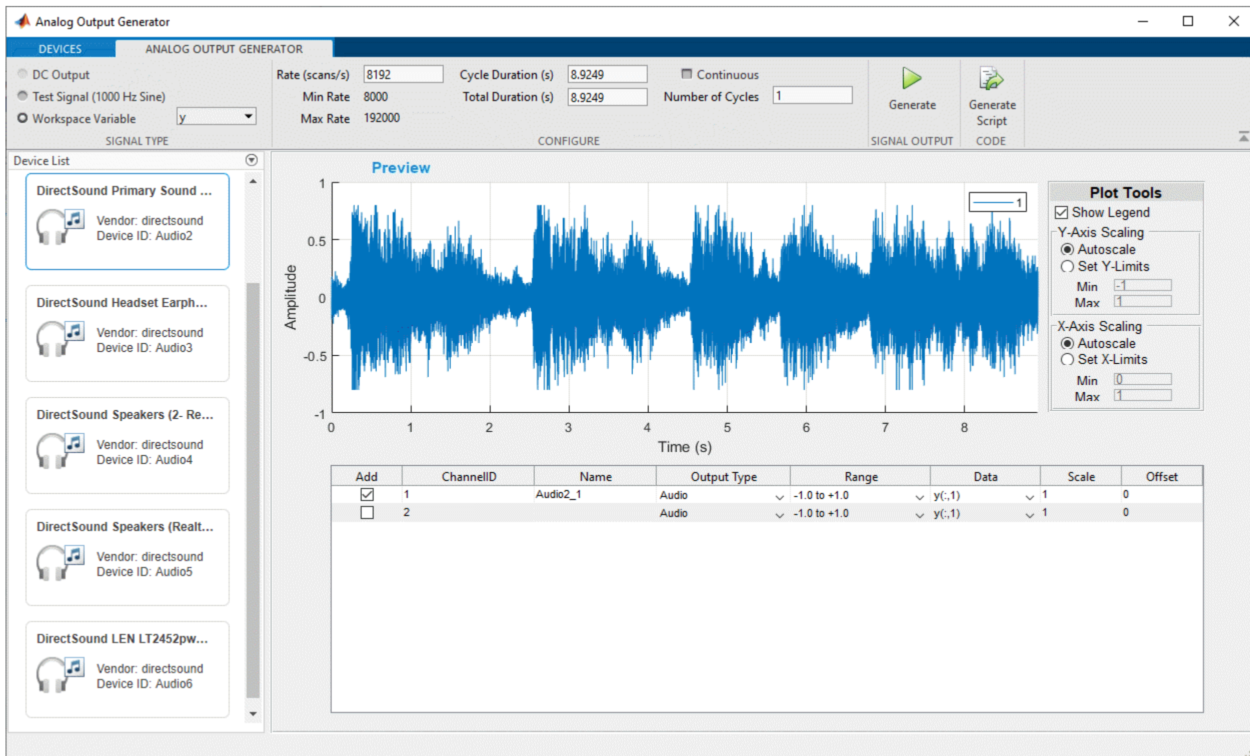
1. Select the device for your output. This might be the primary sound driver, speakers, or a headset.
2. Load the sound data into the workspace with the following command in MATLAB:

   ```
   load handel
   ```

   This loads two variables into your workspace. The sound data is contained in array named `y`. The sampling rate is contained in the variable `Fs`. You will need to know the sampling rate, so display its value.

   ```
   Fs
   ```

   ```
   8192
   ```
3. In the Signal Type section of the **Analog Output Generator** toolstrip, select **Workspace Variable**. In the adjacent selection list, choose `y`. This indicates the source of the data for the generator to output.
4. Enter the `Fs` value of `8192` in the **Rate** text box in the **Analog Output Generator**. This indicates the sampling rate. The app should now look something like this.



5 Click **Generate** to produce the sound output.

If you were successful in producing a sound output, try experimenting with some of the settings in the app. For example, modify the **Rate** value or the **Number of Cycles**.

Tip If you could not hear any sound, use the **Test Signal** option to generate a constant tone. Check all your hardware connections and different devices in the app until you hear the tone.

In the **Analog Output Generator**, click **Generate Script** for the app to open the MATLAB Editor and display the code for producing the signal. The code is generated for the finite (non-continuous) output of this example.

Create DataAcquisition Object
Create a DataAcquisition object for the specified vendor.

```
1   d = daq("directsound");
```

Add Channels
Add channels and set channel properties, if any.

```
2   addoutput(d,"Audio2","1","Audio");
```

Set DataAcquisition Rate
Set scan rate.

```
3   d.Rate = 192000;
```

Define Output Signal
Apply the specified scale and offset on the selected variable.

```
4   outputSignal = [];
5   outputSignal(:,1) = y(:,1) * 1 + 0;
```

Generate Signal
Write the signal data.

```
6   write(d,outputSignal);
```

Clean Up
Clear all DataAcquisition and channel objects.

```
7   clear d outputSignal
```

See Also

Apps
Analog Input Recorder | Analog Output Generator

More About

- “Acquire Data with Analog Input Recorder” on page 6-17

Analog Devices Active Learning Module

Analog Devices ADALM1000 Hardware

Data Acquisition Toolbox supports the Analog Devices ADALM1000 active learning module. ADALM1000 is an inexpensive evaluation platform designed for learning the fundamentals of electrical engineering. You can download associated teaching materials, reference designs, and lab projects from the Analog Devices website.

The support package lets you perform the following tasks in MATLAB with the ADALM1000:

- Generate voltages and waveforms, 0 to +5 V
- Sink or source current, -200 ma to +200 ma
- Simultaneously source voltage and measure current on the same channel
- Simultaneously source current and measure voltage on the same channel

See Also

More About

- “Generate and Measure Signals with Analog Devices ADALM1000” on page 7-3
- “Analog Devices ADALM1000 Limitations” on page B-6

External Websites

- ADALM1000 Overview

Generate and Measure Signals with Analog Devices ADALM1000

In this section...

"Updated Function Syntax" on page 7-3
"Source Voltage and Measure Current" on page 7-3
"Generate a Pulse" on page 7-4
"Generate Waveforms" on page 7-5

Updated Function Syntax

To accommodate the ADALM1000, the following Data Acquisition Toolbox functions allow vendor-specific argument options:

- `daq` and `daqlist` accept the vendor argument `"adi"`.
- `addinput` and `addoutput` accept the device name argument `'SMU1'` (source-measurement unit), and the channel ID arguments `'A'` and `'B'` to correspond with the channel labels on the ADALM1000 module.

Source Voltage and Measure Current

This example shows how to source a voltage while measuring current on the same channel, to calculate load resistance. First program the ADALM1000 to provide a constant 5 V supply to the load, and then measure the current on the same device channel.

Discover your ADALM device and view its information.

```
dev = daqlist("adi")
```

```
dev =

  1×4 table

    DeviceID            Description                Model              DeviceInfo
    ________    ____________________________    ___________    _______________________

     "SMU1"     "Analog Devices Inc. ADALM1000"    "ADALM1000"    [1×1 daq.adi.DeviceInfo]
```

```
dev{1,"DeviceInfo"}
```

```
adi: Analog Devices Inc. ADALM1000 (Device ID: 'SMU1')
   Analog input supports:
      0 to +5.0 Volts,-0.20 to +0.20 A ranges
      Rates from 100000.0 to 100000.0 scans/sec
      2 channels ('A','B')
      'Voltage','Current' measurement types

   Analog output supports:
      0 to +5.0 Volts,-0.20 to +0.20 A ranges
      Rates from 100000.0 to 100000.0 scans/sec
      2 channels ('A','B')
      'Voltage','Current' measurement types
```

Set up a Data Acquisition Toolbox DataAcquisition to operate the ADALM100.

```
d = daq("adi")
```

```
d =

DataAcquisition using Analog Devices Inc. hardware:

                     Running: 0
                        Rate: 100000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: [100000 100000]
```

Add an analog output channel to source voltage from device channel A.

```
addoutput(d,"SMU1","A","Voltage");
```

Add an analog input channel to measure current on the same device channel A.

```
addinput(d,"SMU1","A","Current");
```

View the channel configuration.

```
d.Channels
```

```
ans =

    Index    Type    Device    Channel      Measurement Type           Range              Name
    _____    ____    ______    _______    ____________________    _________________    __________

      1      "ao"    "SMU1"      "A"      "Voltage (SingleEnd)"   "0 to +5.0 Volts"    "SMU1_A"
      2      "ai"    "SMU1"      "A"      "Current"               "-0.20 to +0.20 A"   "SMU1_A_1"
```

Generate an output voltage, and measure the current.

```
V_load = 5;
write(d,V_load);
I_load = read(d,"OutputFormat","Matrix");
write(d,0);  % Reset device output.
R_load = V_load/I_load
```

```
R_load =

   50.3005
```

Tip The ADALM1000 continues to generate the last value programmed until you release the hardware. When you are finished with your signals, reset the device to output 0 volts.

Generate a Pulse

This example shows how to generate a 1-millisecond, 5-volt pulse, surrounded on either side by 10 milliseconds at 0 volts.

```
pdata = zeros(2100,1); % Column vector of 2100 samples.
pdata (1001:1100) = 5; % Pulse in middle of vector.

d = daq("adi");
addoutput(d,"SMU1","B","Voltage");
```

```
write(d,pdata)
```

Generate Waveforms

This example shows how to simultaneously generate a 1-kHz square wave on channel A, and a 100 Hz sine wave on channel B. Each output lasts for 5 seconds.

The example requires two DataAcquisition channels for device channels A and B, both as output channels for voltage.

```
d = daq("adi");
addoutput(d,"SMU1","A","Voltage");
addoutput(d,"SMU1","B","Voltage");
```

Define the two waveforms.

```
Sq = zeros(500000,1); % Column vectors of 500k scans.
Sw = zeros(500000,1);

% Define square wave:
for r = 1:100:499900;
    Sq(r:r+49) = 5;  % Set first 50 of each 100 samples to 5 v.
end

% Define sine wave:
Sw = sin(linspace(1,500000,500000)'*2*pi/1000);
Sw = Sw + 1;  % Shift for positive voltage output
```

View channel configuration.

```
d.Channels
```

```
ans =

    Index    Type    Device    Channel      Measurement Type          Range           Name
    _____    ____    ______    _______    ___________________    ________________    ________

      1      "ai"    "SMU1"      "A"      "Voltage (SingleEnd)"  "0 to +5.0 Volts"   "SMU1_A"
      2      "ai"    "SMU1"      "B"      "Voltage (SingleEnd)"  "0 to +5.0 Volts"   "SMU1_B"
```

Start the output signal generation. The 500000 scans at 100000 scans per second lasts for 5 seconds.

```
write(d,[Sq Sw])
```

See Also

Functions
daq | addinput | addoutput | read | write

More About

- “Analog Devices ADALM1000 Hardware” on page 7-2
- “Analog Devices ADALM1000 Limitations” on page B-6

External Websites

- ADALM1000 Overview

Counter Input and Output

- “Analog and Digital Counters” on page 8-2
- “Acquire Counter Input Data” on page 8-3
- “Generate Pulse Data on a Counter Channel” on page 8-6

Analog and Digital Counters

Use digital and analog counters to count clock ticks and external events. Counters output a pulse train, count rising or falling edges, and measure other quantities including:

- Frequency
- Edges
- PWM
- Position
- Pulse generation

Counters enable timed acquisition and synchronization.

See Also

Related Examples

- "Acquire Counter Input Data" on page 8-3
- "Generate Pulse Data on a Counter Channel" on page 8-6

Acquire Counter Input Data

In this section...
"Add Counter Input Channel" on page 8-3
"Acquire a Single Count" on page 8-3
"Acquire a Single Frequency Count" on page 8-4
"Acquire Counter Input Data in the Foreground" on page 8-4

Add Counter Input Channel

Use `addinput` to add a channel that acquires edge counts from a device. You can acquire a single input data or an array by acquiring in the foreground. For more information, see "Interface Workflow" on page 4-2.

Acquire a Single Count

This example shows how to acquire a single count of falling edges from an NI 9402 with device ID `cDAQ1Mod5`. The example assumes that some external source is providing an input to the counter channel, and that the count is accumulating over time. You can read the accumulated count at one point in time, then reset the counter and read it again at a later time.

Step 1. Create a DataAcquisition object assigned to the variable `d`.

```
d = daq("ni");
```

Step 2. Add a counter input channel with an edge count measurement type.

```
ch = addinput(d,"cDAQ1Mod5","ctr0","EdgeCount")
```

```
ch =

    Index    Type      Device       Channel     Measurement Type    Range          Name
    _____    ____    ___________    _______    ________________    _____    ________________

      1      "ci"    "cDAQ1Mod5"    "ctr0"       "EdgeCount"       "n/a"    "cDAQ1Mod5_ctr0"
```

Step 3. Change the channel `ActiveEdge` property to `'Falling'` and view the channel properties to see the change.

```
ch.ActiveEdge = 'Falling';
get(ch)

      ActiveEdge: Falling
  CountDirection: Increment
    InitialCount: 0
        Terminal: 'PFI0'
SampleTimingType: 10388
            Name: 'cDAQ1Mod5_ctr0'
              ID: 'ctr0'
          Device: [1x1 daq.ni.DeviceInfo]
 MeasurementType: 'EdgeCount'
```

Step 4. Acquire a single scan reading of the counter buffer.

```
count = read(d)
```

```
count =

   133
```

Step 5. Reset counters from the initial count and acquire an updated count value. This value is the number of detections since resetting the counter.

```
resetcounters(d);
count = read(d)
```

```
count =

   71
```

Acquire a Single Frequency Count

This example shows how to acquire a single scan of frequency measurement from an NI 9402 with device ID `cDAQ1Mod5`.

Step 1. Create a DataAcquisition object.

```
d = daq("ni");
```

Step 2. Add a counter channel with a frequency measurement type.

```
addinput(d,"cDAQ1Mod5","ctr0","Frequency")
```

Index	Type	Device	Channel	Measurement Type	Range	Name
1	"ci"	"cDAQ1Mod5"	"ctr0"	"Frequency"	"n/a"	"cDAQ1Mod5_ctr0"

Step 3. Acquire a single scan of counter data.

```
f = read(d,"OutputFormat","Matrix")
```

```
f =

  9.5877e+003
```

Acquire Counter Input Data in the Foreground

This example shows how to acquire rising edge data from an NI 9402 with device ID `cDAQ1Mod5`, and plot the acquired data.

Step 1. Create a DataAcquisition object.

```
d = daq("ni");
```

Step 2. Add a counter input channel with an edge count measurement type.

```
addinput(d,"cDAQ1Mod5","ctr0","EdgeCount")
```

Index	Type	Device	Channel	Measurement Type	Range	Name
1	"ci"	"cDAQ1Mod5"	"ctr0"	"EdgeCount"	"n/a"	"cDAQ1Mod5_ctr0"

Step 3. Add an analog input channel for a voltage measurement type.

The counter input channel requires an external clock to perform a foreground acquisition. If you do not have an external clock, add an analog input channel from a clocked device on the same CompactDAQ chassis to the DataAcquisition. This example uses an NI 9205 device on the same chassis with the device ID `cDAQ1Mod1`. Alternatively, the analog input channel could be on the same device as the counter channel.

```
addinput(d,"cDAQ1Mod1","ai1","Voltage");
```

Step 4. Acquire the data and assign it to the variable `data`, and plot the results.

```
data = read(d,seconds(1),"OutputFormat","Matrix");
plot(data)
```

The plot displays the results from both channels in the DataAcquisition:

- Edge count measurement
- Analog input data

Generate Pulse Data on a Counter Channel

Add Counter Output Channels

Use `addoutput` to add a channel that generates pulses on a counter/timer subsystem. You can generate on one channel or on multiple channels on the same device.

Generate Pulses on a Counter Output Channel

This example shows how to generate pulse data on an NI 9402 with device ID `cDAQ1Mod5`.

Step 1. Create a DataAcquisition object assigned to the variable `d`:

```
d = daq("ni");
```

Step 2. Add a counter output channel for pulse generation:

```
ch = addoutput(d,"cDAQ1Mod5",0,"PulseGeneration")

    Index    Type      Device       Channel    Measurement Type     Range        Name
    _____    ____    ___________    _______    _________________    _____    _________________

      1      "co"    "cDAQ1Mod5"    "ctr0"     "PulseGeneration"    "n/a"    "cDAQ1Mod5_ctr0"
```

Step 3. Configure the output counter channel properties for signal frequency and duty cycle.

```
ch.Frequency = 50000;
ch.DutyCycle = 0.25;
```

Step 4. Generate pulses in the background while MATLAB continues:

```
start(d,"Continuous")
```

Step 5. When finished, stop the DataAcquisition output.

```
stop(d)
```

See Also

More About

- “Synchronize Counter Outputs from Multiple Devices” on page 13-8

Digital Operations

- “Digital Channels” on page 9-2
- “Acquire Non-Clocked Digital Data” on page 9-4
- “Acquire Digital Data Using a Shared Clock” on page 9-5
- “Acquire Digital Data Using an External Clock” on page 9-6
- “Acquire Digital Data Using a Counter Output Channel as External Clock” on page 9-8
- “Acquire Digital Data Using an External Clock via Chassis PFI Terminal” on page 9-11
- “Acquire Digital Data in Hexadecimal Values” on page 9-12
- “Generate Non-Clocked Digital Data” on page 9-13
- “Generate Digital Output Using Decimal Data Across Multiple Lines” on page 9-14
- “Generate and Acquire Data on Bidirectional Channels” on page 9-15
- “Generate Signals on Both Analog and Digital Channels” on page 9-16

Digital Channels

Digital subsystems transfer digital or logical values in bits via digital lines. You can perform clocked and non-clocked digital operations using the DataAcquisition interface in the Data Acquisition Toolbox.

Add lines of the digital subsystem as channels to your DataAcquisition using `addinput`, `addoutput`, or `addbidirectional`. Digital channels can be:

- **InputOnly**: Allows you to read digital data.
- **OutputOnly**: Allows you to write digital data.
- **Bidirectional**: Allows you to change the direction of the channel to read or write data. You can set the direction to `Input` or `Output`. By default the direction is `Input`.

Digital Clocked Operations

With clocked operations, you can acquire or generate clocked signals at a specified scan rate for a specified duration or number of scans. These operations use hardware timing to acquire or generate at specific times. The operation is controlled by events tied to subsystem clocks. In a clocked acquisition, data is transferred from the device to your system memory and displays when the event calls for the data. In signal generation, data generated from the device is stored in memory until the configured event occurs. When an event occurs, data is sent via the digital channels to the specified devices.

Your device might or might not have an onboard clock. However, Data Acquisition Toolbox does not support direct access to device onboard clocks for clocked sampling when using only digital input/output channels with a DataAcquisition object. You can enable clocked operation by adding a clock in one of these ways:

- Import a clock from an external source. See “Acquire Digital Data Using an External Clock” on page 9-6 for more information.
- Generate a clock from a Counter Output subsystem in your DataAcquisition and import that clock. See “Acquire Digital Data Using a Counter Output Channel as External Clock” on page 9-8 for more information.
- Share a clock from the analog input subsystem. See “Acquire Digital Data Using a Shared Clock” on page 9-5 for more information.

Access Digital Subsystem Information

This example shows how to access the device digital subsystem information and find line and port information using `daqlist`.

Find devices connected to your system and find the NI model USB-6509 device.

```
dev = daqlist("ni")
```

```
dev =

  2×4 table

    DeviceID              Description                Model          DeviceInfo
    ________    ____________________________    _________    ___________________
```

```
    "Dev2"     "National Instruments(TM) USB-6509"    "USB-6509"    [1×1 daq.ni.DeviceInfo]
    "Dev3"     "National Instruments(TM) USB-6211"    "USB-6211"    [1×1 daq.ni.DeviceInfo]
```

View the subsystem information in the `DeviceInfo` for `Dev2` using index `1`.

```
DevInf = dev.DeviceInfo(1)
```

```
DevInf =

ni: National Instruments(TM) USB-6509 (Device ID: 'Dev2')
   Digital IO supports:
      96 channels ('port0/line0' - 'port9/line7')
      'InputOnly','OutputOnly','Bidirectional' measurement types
```

See Also

More About

- “Acquire Digital Data Using a Shared Clock” on page 9-5
- “Acquire Digital Data Using an External Clock” on page 9-6

Acquire Non-Clocked Digital Data

This example shows how to read digital data using two channels on an NI USB-6255

Discover NI devices connected to your system and find the ID for the NI 6255:

```
dev = daqlist("ni")
```

```
dev = 

  3×4 table

    DeviceID              Description                  Model              DeviceInfo       
    ________    __________________________________    __________    _______________________

     "Dev1"     "National Instruments(TM) USB-6255"    "USB-6255"    [1×1 daq.ni.DeviceInfo]
     "Dev2"     "National Instruments(TM) USB-6509"    "USB-6509"    [1×1 daq.ni.DeviceInfo]
     "Dev3"     "National Instruments(TM) USB-6211"    "USB-6211"    [1×1 daq.ni.DeviceInfo]
```

Create a DataAcquisition object and add two input lines from port `0` on `Dev1`:

```
d = daq("ni");
ch = addinput(d,"Dev1","Port0/Line0:1","Digital")
```

```
ch = 

    Index    Type     Device      Channel       Measurement Type    Range          Name        
    _____    _____    ______    ____________    ________________    _____    __________________

      1      "dio"    "Dev1"    "port0/line0"      "InputOnly"      "n/a"    "Dev1_port0/line0"
      2      "dio"    "Dev1"    "port0/line1"      "InputOnly"      "n/a"    "Dev1_port0/line1"
```

Acquire a single scan of digital data from both channels:

```
data = read(d,"OutputFormat","Matrix")
```

```
data = 

     1     0
```

Acquire Digital Data Using a Shared Clock

This example shows how to share the clock with the analog input subsystem on your device with the digital subsystem to acquire clocked data that is automatically synchronized. You do not need any physical connections to share the clock. For more information, see "Automatic Synchronization" on page 13-4.

Create a DataAcquisition object and add a digital input line from port `0` line `0` on `Dev1`.

```
d = daq("ni");
addinput(d,"Dev1","Port0/Line0","Digital")
```

Note Not all devices support clocked digital I/O operations with hardware timing. For these devices you can use software timed operations with single scan calls to `read` and `write`.

Devices that support clocked digital I/O operations might not support them on all ports. Check your device specifications.

Add an analog input channel to your DataAcquisition.

```
addinput(d,"Dev1",0,"Voltage");
d.Channels
```

```
ans =

    Index    Type     Device      Channel       Measurement Type         Range               Name
    _____    _____    ______    _____________    ________________    __________________    _________________

      1      "dio"    "Dev1"    "port0/line0"    "InputOnly"         "n/a"                 "Dev1_port0/line0"
      2      "ai"     "Dev1"    "ai0"            "Voltage (Diff)"    "-10 to +10 Volts"    "Dev1_ai0"
```

Read and plot the acquired digital data. The device acquires digital data at the scan rate determined by its analog subsystem.

```
dataIn = read(d,seconds(1),"OutputFormat","Matrix");
plot(dataIn(1:100,1)) % Column 1 is data from the first channel.
```

See Also

Related Examples

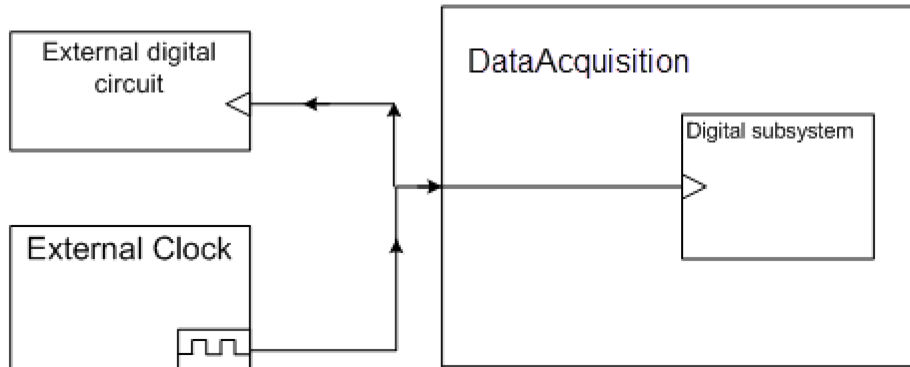
- "Acquire Digital Data Using an External Clock" on page 9-6
- "Acquire Digital Data Using a Counter Output Channel as External Clock" on page 9-8

More About

- "Synchronization" on page 13-2

Acquire Digital Data Using an External Clock

This example shows how to acquire digital data in the foreground by using an external scan clock.



You can use a function generator or the on-board clock from a digital circuit. Here, a function generator is physically wired to terminal PFI9 on device NI 6255.

Create a DataAcquisition object and add a output line at port `0` line `2` on `Dev1`.

```
d = daq("ni");
ch = addinput(d,"Dev1","Port0/Line2","Digital")
```

```
ch = 

    Index    Type     Device     Channel          Measurement Type    Range          Name        
    _____    _____    ______    _____________    ________________    _____    __________________

      1      "dio"    "Dev1"    "port0/line2"      "InputOnly"       "n/a"    "Dev1_port0/line2"
```

Note Not all devices support clocked digital I/O operations with hardware timing. For these devices you can use software timed operations with single scan calls to `read` and `write`.

Devices that support clocked digital I/O operations might not support them on all ports. Check your device specifications.

Set the rate of your DataAcquisition to the expected rate of your external scan clock.

```
d.Rate = 1000;
```

Note Importing an external clock does not automatically set the scan rate of your DataAcquisition. Manually set the DataAcquisition `Rate` property value to match the expected external clock frequency.

Programmatically add a scan clock to your DataAcquisition, indicating the source as external and the target as device terminal `PFI9`.

```
clk = addclock(d,"ScanClock","External","Dev1/PFI9")
```

```
clk = 

  Clock with properties:
```

```
         Source: 'External'
    Destination: 'Dev1/PFI9'
           Type: ScanClock
```

Acquire clocked digital data and plot it.

```
dataIn = read(d,seconds(1),"OutputFormat","Matrix");
plot(dataIn(1:100,1))
```

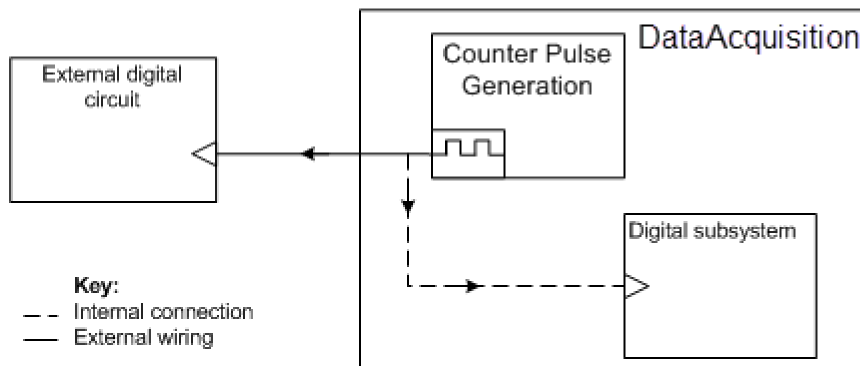
See Also

Related Examples

- “Acquire Digital Data Using a Shared Clock” on page 9-5
- “Acquire Digital Data Using a Counter Output Channel as External Clock” on page 9-8

Acquire Digital Data Using a Counter Output Channel as External Clock

This example shows how to use a device counter output channel to generate pulses to use as an external clock for acquiring digital data.



In this example, you generate a clock in one DataAcquisition using a counter output channel and export the clock to another DataAcquisition that acquires digital data. The counter output and the digital subsystem can be on the same device or on different devices. If using multiple devices not in the same chassis, you must wire a physical connection between the counter output of one device to the digital clock input of the other.

Note Importing an external clock does not automatically set the scan rate of your DataAcquisition. Manually set the DataAcquisition `Rate` property value to match the expected external clock frequency.

Generate a Clock Using a Counter Output Channel

Create a clocked DataAcquisition with a counter output channel that continuously generates frequency pulses in the background. You can use this channel as an external clock for a clocked digital acquisition.

Define the clock frequency to be used for synchronizing the scan rate of your counter output as well as the rate of your digital acquisition.

```
clockFreq = 100;
```

Create a DataAcquisition object and add a counter output channel for `PulseGeneration` measurement type.

```
daqClk = daq("ni");
ch1 = addoutput(daqClk,"Dev1","ctr0","PulseGeneration");
```

Tip Make sure the counter channel you add is not being used in a different DataAcquisition, otherwise a terminal conflict error occurs.

Save the counter output terminal ID to a variable so you can use it later to specify the external clock that synchronizes your digital clocked operations.

```
clkTerminal = ch1.Terminal;
```

Set the frequency of your counter channel to the clock frequency.

```
ch1.Frequency = clockFreq;
```

Use Counter Clock to Acquire Clocked Digital Data

Create a DataAcquisition for digital input and import the external clock from the clock DataAcquisition.

Create a DataAcquisition and add a digital input line from port `0` line `2` on `Dev1`.

```
daqDgt = daq("ni");
addinput(daqDgt,"Dev1","Port0/Line2","Digital")
```

Note Not all devices support clocked digital I/O operations with hardware timing. For these devices you can use software timed operations with single scan calls to `read` and `write`.

Devices that support clocked digital I/O operations might not support them on all ports. Check your device specifications.

Tip PFI terminal resources might be shared. Check your device routing in the NI MAX app.

Set the DataAcquisition scan rate to the same value as the rate of the counter output channel.

```
daqDgt.Rate = clockFreq;
```

Import the clock from your clock DataAcquisition to synchronize your acquisition.

```
addclock(daqDgt,"ScanClock","External",clkTerminal)
```

Start the counter output channel to run continuously in the background.

```
start(daqClk,"Continuous")
```

Pulse generation begins immediately on the counter output. It does not need data.

Acquire and plot digital input data.

```
dataIn = read(daqDgt,seconds(1),"OutputFormat","Matrix");
plot(dataIn(1:100,1))
```

Stop the clock DataAcquisition.

```
stop(daqClk)
```

See Also

Related Examples

- “Acquire Digital Data Using a Shared Clock” on page 9-5
- “Acquire Digital Data Using an External Clock” on page 9-6

Acquire Digital Data Using an External Clock via Chassis PFI Terminal

This example shows how to acquire clocked digital data using an external clock provided at the CompactDAQ chassis PFI terminal. It uses a cDAQ 9178 chassis and NI 9402 module with ID cDAQ2Mod3. A digital signal is connected to the module PFI0 terminal to provide a scan clock.

Create a DataAcquisition object and add the digital input line.

```
d = daq("ni");
addinput(d,"cDAQ2Mod3","Port0/Line0","Digital");
```

Add a clock specifying source and destination. Then set the DataAcquisition scan rate to match the external clock frequency.

```
addclock(d,"ScanClock","External","cDAQ2/PFI0");
d.Rate = 100e3;
```

Acquire and plot the digital input data.

```
[data,timestamps] = read(d,seconds(1),"OutputFormat","Matrix");
plot(timestamps,data(1:100,1))
```

Acquire Digital Data in Hexadecimal Values

This example shows how to acquire digital data using four channels on an NI 6255.

Discover devices connected to your system and find the ID for the NI 6255.

```
dev = daqlist
```

```
dev =

  3×4 table

    DeviceID               Description                 Model            DeviceInfo
    ________    __________________________________    __________    ______________________

     "Dev1"     "National Instruments(TM) USB-6255"    "USB-6255"    [1×1 daq.ni.DeviceInfo]
     "Dev2"     "National Instruments(TM) USB-6363"    "USB-6363"    [1×1 daq.ni.DeviceInfo]
```

Create a DataAcquisition and add four digital input lines from port `0` on `Dev1`.

```
d = daq("ni");
addinput(d,"Dev1","Port0/Line0:3","Digital");
d.Channels
```

```
ans =

    Index    Type     Device      Channel       Measurement Type    Range          Name
    _____    _____    ______    ____________    ________________    _____    _________________

      1      "dio"    "Dev1"    "port0/line0"     "InputOnly"       "n/a"    "Dev1_port0/line0"
      2      "dio"    "Dev1"    "port0/line1"     "InputOnly"       "n/a"    "Dev1_port0/line1"
      3      "dio"    "Dev1"    "port0/line2"     "InputOnly"       "n/a"    "Dev1_port0/line2"
      4      "dio"    "Dev1"    "port0/line3"     "InputOnly"       "n/a"    "Dev1_port0/line3"
```

Acquire digital data in hexadecimal values.

```
hData = binaryVectorToHex(read(d,"OutputFormat","Matrix"))
```

```
hData =

    'C'
```

Generate Non-Clocked Digital Data

This example shows how to write data to two lines on an NI 6255.

Discover NI devices connected to your system and find the ID for the NI 6255.

```
d = daqlist("ni")
```

```
dev =

  3×4 table

    DeviceID              Description                Model             DeviceInfo
    ________    __________________________________    __________    ____________________

     "Dev1"     "National Instruments(TM) USB-6255"    "USB-6255"    [1×1 daq.ni.DeviceInfo]
     "Dev2"     "National Instruments(TM) USB-6363"    "USB-6363"    [1×1 daq.ni.DeviceInfo]
```

Create a DataAcquisition object and add two digital output lines from port `0` on `Dev1`.

```
d = daq("ni");
addoutput(d,"Dev1","Port0/Line0:1","Digital");
d.Channels
```

```
ans =

    Index    Type     Device      Channel       Measurement Type    Range          Name
    _____    _____    ______    _____________    ________________    _____    __________________

      1      "dio"    "Dev1"    "port0/line0"      "OutputOnly"      "n/a"    "Dev1_port0/line0"
      2      "dio"    "Dev1"    "port0/line1"      "OutputOnly"      "n/a"    "Dev1_port0/line1"
```

Generate digital output.

```
write(d,[1 0])
```

Generate Digital Output Using Decimal Data Across Multiple Lines

This example shows how to convert decimal data and output to two lines on an NI 6255.

Discover NI devices connected to your system and find the ID for the NI 6255.

```
d = daqlist("ni")
```

```
dev = 

  3×4 table

    DeviceID              Description                  Model              DeviceInfo       
    ________    __________________________________    __________    ______________________

     "Dev1"     "National Instruments(TM) USB-6255"    "USB-6255"    [1×1 daq.ni.DeviceInfo]
     "Dev2"     "National Instruments(TM) USB-6363"    "USB-6363"    [1×1 daq.ni.DeviceInfo]
```

Create a DataAcquisition and add two digital output lines from port `0` on `Dev1`.

```
d = daq("ni");
addoutput(d,"Dev1","Port0/Line0:1","Digital");
d.Channels
```

```
ans =

    Index    Type     Device       Channel       Measurement Type    Range          Name        
    _____    _____    ______    _____________    ________________    _____    __________________

      1      "dio"    "Dev1"    "port0/line0"      "OutputOnly"      "n/a"    "Dev1_port0/line0"
      2      "dio"    "Dev1"    "port0/line1"      "OutputOnly"      "n/a"    "Dev1_port0/line1"
```

Convert the decimal number 2 to a binary vector, and generate that digital output value on the two lines.

```
write(d,decimalToBinaryVector(2))
```

Generate and Acquire Data on Bidirectional Channels

This example shows how to use a bidirectional channel and read and write data using the same two lines on an NI 6255.

Discover NI devices connected to your system and find the ID for the NI 6255.

```
d = daqlist("ni")
```

```
dev =

  3×4 table

    DeviceID              Description                  Model            DeviceInfo
    ________    ________________________________    __________    ____________________

     "Dev1"     "National Instruments(TM) USB-6255"    "USB-6255"    [1×1 daq.ni.DeviceInfo]
     "Dev2"     "National Instruments(TM) USB-6363"    "USB-6363"    [1×1 daq.ni.DeviceInfo]
```

Create a DataAcquisition and add two lines from port 0 and 2 lines from port 1 on Dev1.

```
d = daq("ni");
addbidirectional(d,"Dev1","Port0/Line0:1","Digital");
addbidirectional(d,"Dev1","Port1/Line0:1","Digital");
d.Channels
```

```
ans =

    Index    Type     Device      Channel         Measurement Type        Range          Name
    _____    _____    ______    ____________    _____________________    _____    ________________

      1      "dio"    "Dev1"    "port0/line0"    "Bidirectional (Input)"    "n/a"    "Dev1_port0/line0"
      2      "dio"    "Dev1"    "port0/line1"    "Bidirectional (Input)"    "n/a"    "Dev1_port0/line1"
      3      "dio"    "Dev1"    "port1/line0"    "Bidirectional (Input)"    "n/a"    "Dev1_port1/line0"
      4      "dio"    "Dev1"    "port1/line1"    "Bidirectional (Input)"    "n/a"    "Dev1_port1/line1"
```

Set the direction on all channels to output data.

```
set(d.Channels,"Direction","Output");
```

Generate the digital output data.

```
write(d,[1 0 1 0])
```

Change the direction on all channels to input data

```
set(d.Channels,"Direction","Input");
```

Acquire the digital data.

```
read(d,"OutputFormat","Matrix")
```

```
ans =

     1     0     1     0
```

Generate Signals on Both Analog and Digital Channels

This example shows how to generate signals when the DataAcquisition contains both analog and digital channels.

Discover NI devices connected to your system and find the ID for the NI 6255.

```
d = daqlist("ni")
```

```
dev =

  3×4 table

    DeviceID               Description                  Model             DeviceInfo
    ________    __________________________________    __________    ______________________

     "Dev1"     "National Instruments(TM) USB-6255"    "USB-6255"    [1×1 daq.ni.DeviceInfo]
     "Dev2"     "National Instruments(TM) USB-6363"    "USB-6363"    [1×1 daq.ni.DeviceInfo]
```

Create a DataAcquisition and add two digital output lines from port `0` on `Dev1`.

```
d = daq("ni");
addoutput(d,"Dev1","Port0/Line0:1","Digital")
```

Add an analog output channel from `Dev1`, then view all channels.

```
addoutput(d,'Dev1',0,'Voltage')
d.Channels
```

```
ans =

    Index    Type     Device      Channel       Measurement Type             Range                 Name
    _____    _____    ______    _____________   ____________________    __________________    __________________

      1      "dio"    "Dev1"    "port0/line0"   "OutputOnly"            "n/a"                 "Dev1_port0/line0"
      2      "dio"    "Dev1"    "port0/line1"   "OutputOnly"            "n/a"                 "Dev1_port0/line1"
      3      "ao"     "Dev1"    "ao0"           "Voltage (SingleEnd)"   "-10 to +10 Volts"    "Dev1_ao0"
```

Output a single scan of data on both the digital and analog channels.

```
write(d, [decimalToBinaryVector(2), 1.23])
```

Multichannel Audio

Audio Input and Output

You can acquire and generate audio signals using one or more available channels of a supported audio device. You can also simultaneously operate channels on multiple supported audio devices. Data Acquisition Toolbox supports audio channels for devices that work with the DirectSound interface. You can:

- Acquire and generate audio signals either in sequence or as separate operations.
- Acquire and generate signals simultaneously where the signals may share their start time.
- Acquire audio data in the background and filter or process the input data simultaneously. You can generate data immediately in response to the processed input data. In this case, both the acquisition and generation operations start and stop together.

Data Acquisition Toolbox does not read directly from or write directly to audio files using the multichannel audio feature. Instead, use the MATLAB functions `audioread` and `audiowrite`.

Multichannel Audio Scan Rate

The `Rate` of an audio DataAcquisition is the scan rate at which it samples audio data. All channels in a DataAcquisition have the same scan rate. The default DataAcquisition rate for an audio DataAcquisition is 44100 Hz. If you have multiple devices in the DataAcquisition, make sure that they can all operate at a common scan rate.

Audio Measurement Range

Data you acquire or generate using audio channels contains double-precision values. These values are normalized to the range of `-1` to `+1`. The DataAcquisition represents data acquired or generated in amplitude without units.

Acquire Audio Data

This example shows how to acquire audio data for seven seconds and plot the result.

Discover DirectSound audio devices installed on your system and create a DataAcquisition for these devices.

```
dev = daqlist;
```

```
dev =

  4×4 table

    DeviceID                      Description                                     Model                             DeviceInfo
    ________    ______________________________________________    ________________________________________    ____________________

    "Audio0"    "DirectSound Primary Sound Capture Driver"        "Primary Sound Capture Driver"              [1×1 daq.audio.Devic
    "Audio1"    "DirectSound Headset Microphone (Plantronics BT600)"    "Headset Microphone (Plantronics BT600)"    [1×1 daq.audio.Devic
    "Audio2"    "DirectSound Primary Sound Driver"                "Primary Sound Driver"                      [1×1 daq.audio.Devic
    "Audio3"    "DirectSound Headset Earphone (Plantronics BT600)"      "Headset Earphone (Plantronics BT600)"      [1×1 daq.audio.Devic
```

```
d = daq("directsound")
```

```
d =

DataAcquisition using DirectSound hardware:
```

```
                    Running: 0
                       Rate: 44100
          NumScansAvailable: 0
           NumScansAcquired: 0
             NumScansQueued: 0
    NumScansOutputByHardware: 0
                  RateLimit: []
```

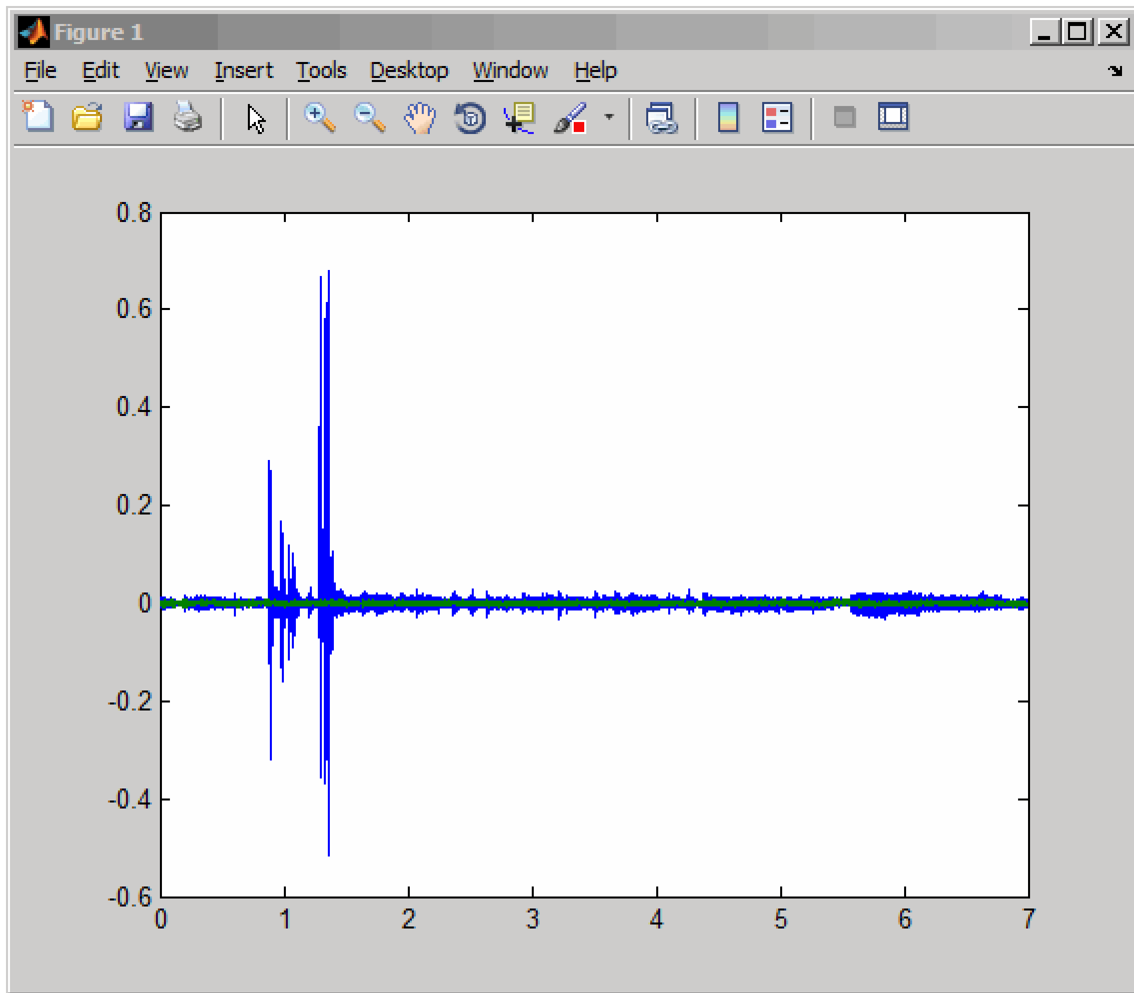
Add an audio input channel for the microphone with id `Audio1`. The measurement type is `Audio`.

```
addinput(d,"Audio1",1,"Audio");
```

```
ch =

    Index    Type      Device     Channel    Measurement Type        Range           Name
    _____    ______    ________    _______    ________________    _____________    __________

      1      "audi"    "Audio1"     "1"           "Audio"        "-1.0 to +1.0"    "Audio1_1"
```

Acquire 7 seconds of data in the foreground and plot the data versus time.

```
[data,t] = read(d, seconds(7), "OutputFormat","Matrix");
plot(t,data)
```



See Also

Related Examples

- “Generate Audio Signals” on page 18-86

Waveform Function Generation

- “Digilent Analog Discovery Devices” on page 11-2
- “Digilent Function Waveform Generator Channels” on page 11-3
- “Waveform Types” on page 11-5
- “Generate a Standard Waveform Using Function Waveform Generator Channels” on page 11-8

Digilent Analog Discovery Devices

MATLAB supports the Digilent Analog Discovery design kit, a low-cost, portable USB DAQ device. The kit enables project-based learning for analog circuit design. For professors and course instructors, the kit comes with downloadable teaching materials, reference designs, and lab projects.

The Data Acquisition Toolbox Support Package for Digilent Analog Discovery hardware lets you perform the following tasks in MATLAB:

- Read data from oscilloscope channels.
- Control and generate data from waveform generators.
- Characterize ICs and measure behavior of the circuit and IC components.
- Configure the sampling rate of the Analog Discovery device.
- Trigger the start of your data acquisition.
- Find and display Digilent Analog Discovery device settings.

See Also

Related Examples

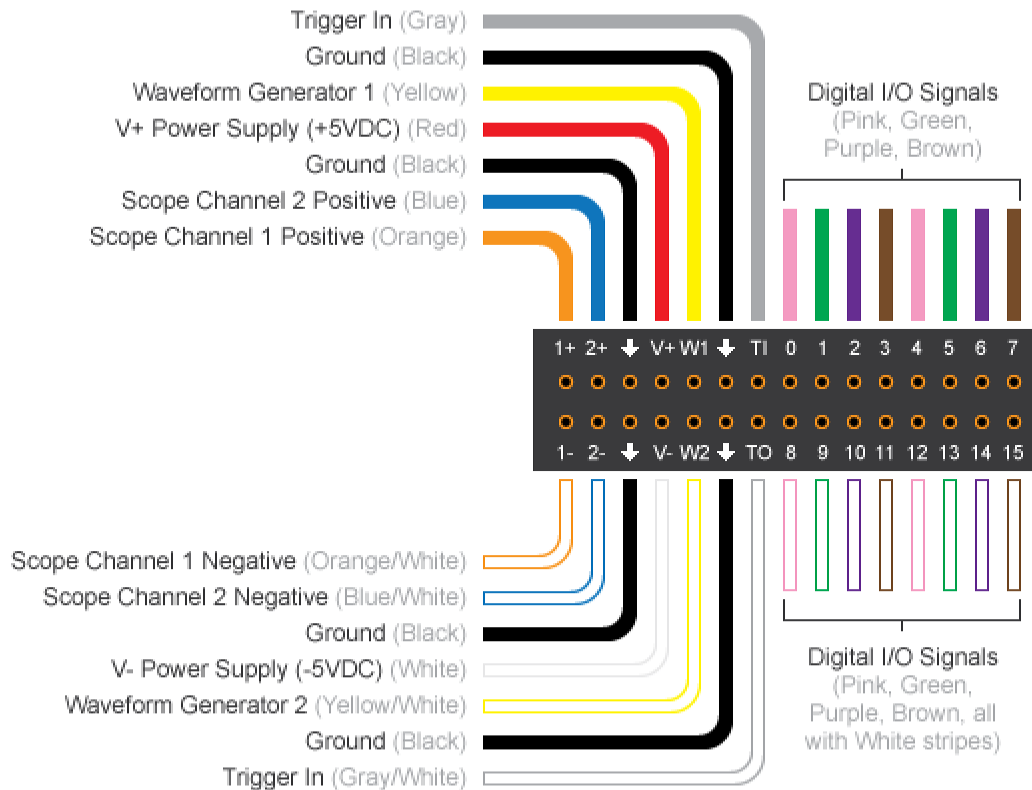
- “Getting Started Acquiring Data with Digilent Analog Discovery” on page 18-68
- “Getting Started Generating Data with Digilent Analog Discovery” on page 18-71

More About

- “Install Hardware Support Package for Vendor Support” on page 5-2

Digilent Function Waveform Generator Channels

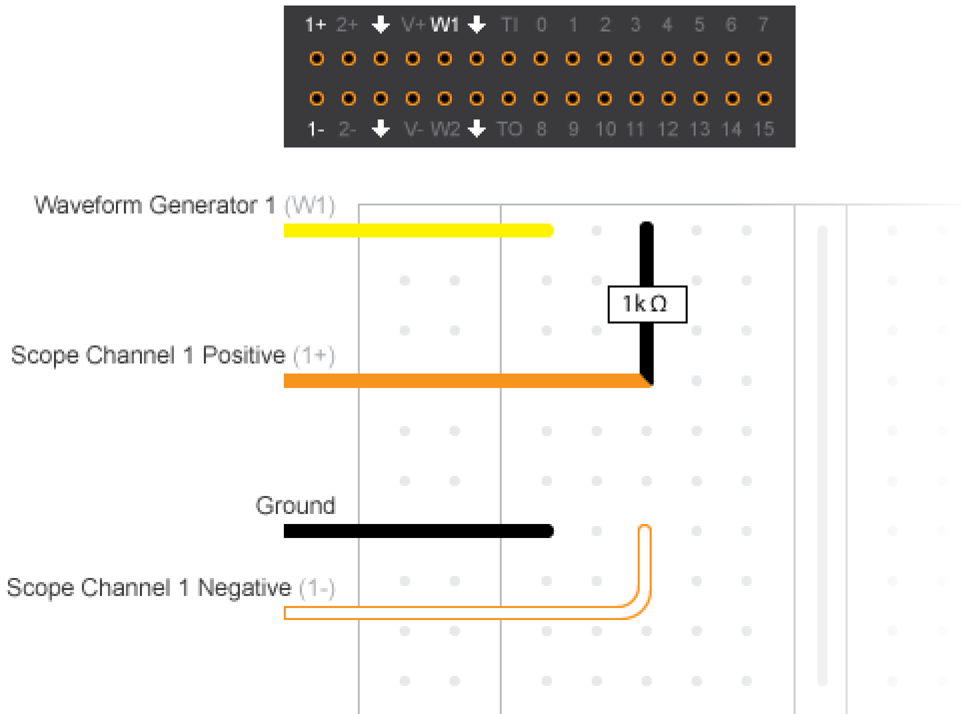
Waveform function generator channels on a Digilent device can generate both standard and arbitrary function waveforms. For more information, see “Waveform Types” on page 11-5. This diagram shows the pin configuration on a typical Digilent Analog Discovery device. The yellow and the yellow/white lines represent the waveform generator channels, marked as `W1` and `W2` on the device.



To test the Analog Discovery device, create the following connection to acquire the generated waveform, and use it with the corresponding code:

- `1+` (scope channel 1 positive) to `WI` through a `1K` resistor.
- `1–` (scope channel 1 negative) `W2` to `GND`.

This diagram depicts these connections on a breadboard.



Unlike analog input channels, the waveform generator channels control their own frequency. If your DataAcquisition contains both waveform generator channels and any other type of acquisition channels, the waveform generator channels will have their own frequency and all other channels will inherit the DataAcquisition scan rate. If you have analog input channels in the DataAcquisition with waveform generator channels, the analog input channels start first and act as a trigger for the waveform generator channels.

See Also

Related Examples

- “Generate Standard Periodic Waveforms Using Digilent Analog Discovery” on page 18-76
- “Generate Arbitrary Periodic Waveforms Using Digilent Analog Discovery” on page 18-79

More About

- “Waveform Types” on page 11-5

Waveform Types

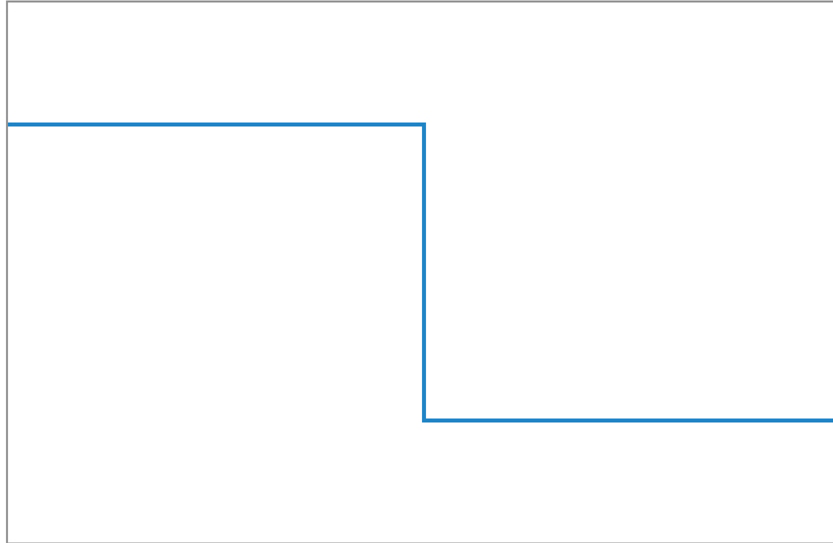
Digilent Analog Discovery devices support generation of arbitrary waveforms, standard waveforms, or both. If your device supports standard waveforms, you can set the gain, offset, and frequency to control the output. Standard waveform types include:

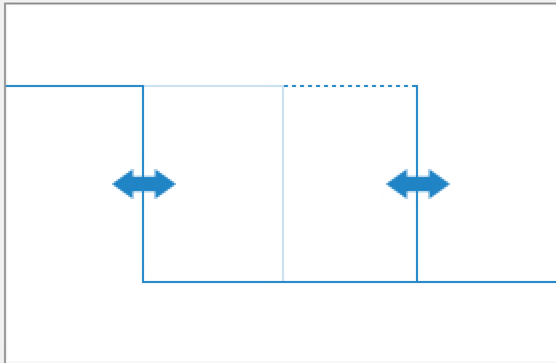
- Sine
- Square
- Triangle
- RampUp
- RampDown
- DC

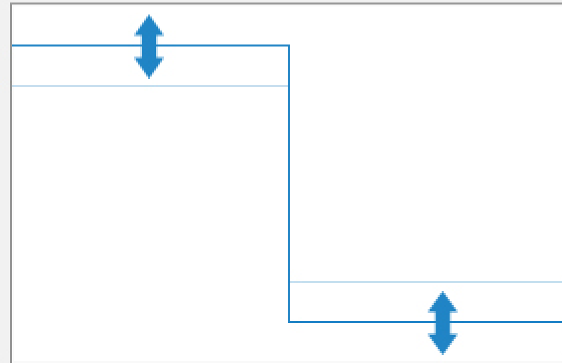
You can control the behavior of different waveform types using the associated properties. This table shows you which properties work with the supported waveform types for Digilent devices.

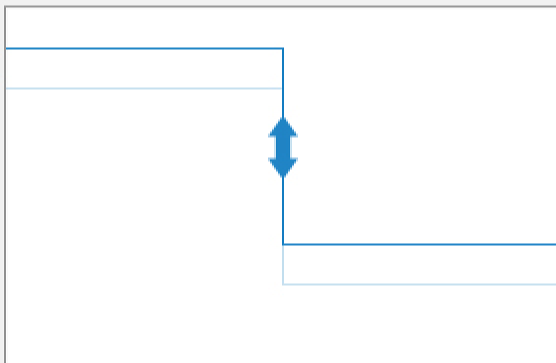
	Frequency	Gain	Offset	Phase	DutyCycle
Sine	✓	✓	✓	✓	
Square	✓	✓	✓	✓	✓
Triangle	✓	✓	✓	✓	✓
RampUp	✓	✓	✓	✓	✓
RampDown	✓	✓	✓	✓	✓
DC			✓		
Arbitrary	✓				

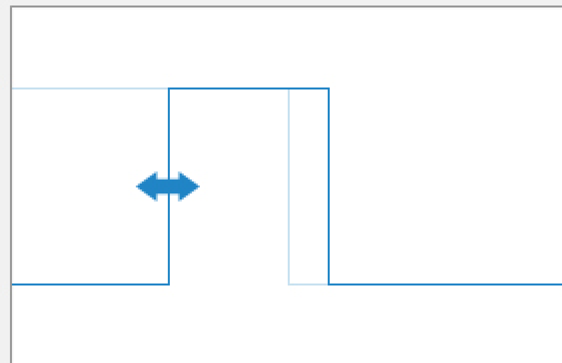
This diagram illustrates how these properties affect a standard square waveform.


Original


DutyCycle


Gain


Offset


Phase

Standard waveforms cannot be clipped. You must keep gain and offset values so that the waveform amplitude remains within voltage range. You cannot change gain and offset of arbitrary waveforms.

See Also

Related Examples

- “Generate Standard Periodic Waveforms Using Digilent Analog Discovery” on page 18-76
- “Generate Arbitrary Periodic Waveforms Using Digilent Analog Discovery” on page 18-79

More About

- “Digilent Function Waveform Generator Channels” on page 11-3

Generate a Standard Waveform Using Function Waveform Generator Channels

This example shows how to use the function generator channel in a DataAcquisition to generate a sine function waveform at a frequency of 100 kHz. The signal output voltage range is specified as `-5.0` to `+5.0` volts

Discover available Digilent devices.

```
dev = daqlist("digilent")

dev =

  1×4 table

    DeviceID                   Description                          Model                 DeviceInfo
    ________    _____________________________________________    __________________    ______________________

     "AD1"      "Digilent Inc. Analog Discovery 2 Kit Rev. C"    "Analog Discovery 2"    [1×1 daq.di.DeviceInfo]
```

Create a DataAcquisition object for Digilent devices.

```
d = daq("digilent")

d = 

DataAcquisition using Digilent Inc. hardware:

                     Running: 0
                        Rate: 10000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []
```

Add a waveform function generator channel for device `AD1` with a `Sine` waveform type.

```
fgenCh = addoutput(d,"AD1",1,"Sine")

fgenCh = 

    Index     Type      Device    Channel    Measurement Type           Range                 Name
    _____    ______    ______    _______    ________________    ___________________    ____________

      1      "fgen"    "AD1"       "1"           "Sine"         "-5.0 to +5.0 Volts"    "AD1_1_fgen"
```

Set the channel amplitude to 5 v using the `Gain` property and the channel frequency to 100 kHz.

```
fgenCh.Gain = 5;
fgenCh.Frequency = 100e3;
```

Specify the output duration to run for `5` seconds and start the generation.

```
write(d,seconds(5))
```

See Also

Related Examples

- "Generate Standard Periodic Waveforms Using Digilent Analog Discovery" on page 18-76
- "Generate Arbitrary Periodic Waveforms Using Digilent Analog Discovery" on page 18-79

Triggers and Clocks

- “Trigger Connections” on page 12-2
- “Acquire Voltage Data Using a Digital Trigger” on page 12-4
- “Clock Connections” on page 12-5

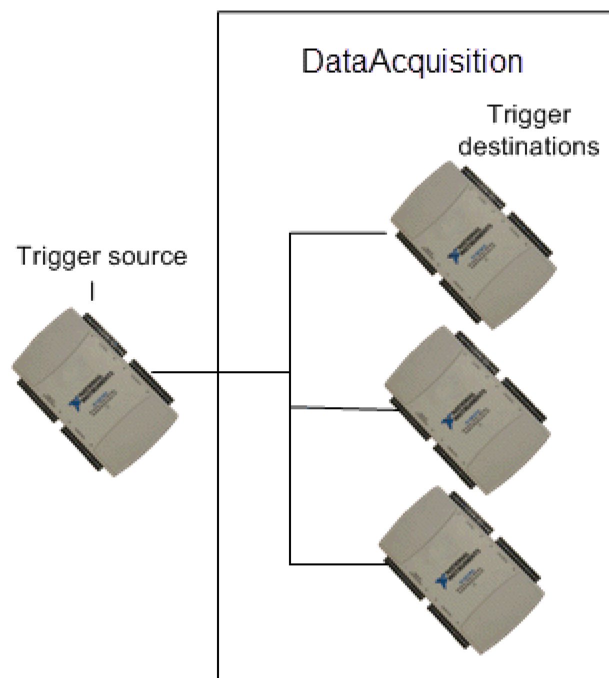
Trigger Connections

In this section...
"When to Use Triggers" on page 12-2
"External Triggering" on page 12-2

When to Use Triggers

Use triggers to simultaneously start all devices in the DataAcquisition. You connect a trigger source to a trigger destination. A trigger source can be either external, where the trigger comes from a source outside a DataAcquisition, or on a device and terminal pair within a DataAcquisition. Trigger destination devices can be external, where the signals are received outside the DataAcquisition, or devices within the DataAcquisition. For more information, see "Source and Destination Devices" on page 13-3.

Note You can have multiple destinations for your trigger, but only one source.



Note You cannot use trigger and clock connections with audio channels.

External Triggering

You can configure devices in a DataAcquisition to receive an external trigger. To use an external trigger source, your connection parameters must correctly specify the exact device and terminal pairs to which the external source is connected. Two circumstances of externally clocked and triggered synchronization are:

- An external hardware event that controls the operation of one or more devices in a DataAcquisition object. For example, opening and closing a switch starts a background acquisition on a DataAcquisition.
- An external hardware event synchronizes multiple devices in a DataAcquisition. For example, opening and closing of a switch starts a background operation across multiple devices or CompactDAQ chassis in a DataAcquisition.

See Also

Related Examples

- “Multiple-Device Synchronization Using USB or PXI Devices” on page 13-7
- “Multiple-Chassis Synchronization with CompactDAQ Devices” on page 13-12
- “Acquire Voltage Data Using a Digital Trigger” on page 12-4

More About

- “Synchronization” on page 13-2

Acquire Voltage Data Using a Digital Trigger

This example shows how to use a falling edge digital trigger, which occurs when a switch closes on an external source. The trigger is connected to terminal `PFI0` on device `Dev1` and starts acquiring sensor voltage data.

Create a DataAcquisition object for NI devices.

```
d = daq("ni");
```

Add a voltage input channel from NI USB-6211 with device ID `Dev1`.

```
addinput(d,"Dev1",0,"Voltage")
```

Physically connect the switch to terminal `PFI0` on NI USB-6211. The trigger comes from the switch, which is an external source. Programmatically add the trigger to the DataAcquisition, indicating source, destination, and device PFI terminal.

```
t = addtrigger(d,"Digital","StartTrigger","External","Dev1/PFI0")
```

```
t = 

  DigitalTrigger with properties:

         Source: 'External'
    Destination: 'Dev1/PFI0'
           Type: StartTrigger
      Condition: 'RisingEdge'
```

Set the trigger `Condition` property to `'FallingEdge'`.

```
t.Condition = 'FallingEdge';
```

Acquire data and store it in `dataIn`. The DataAcquisition waits for the trigger to occur, and starts acquiring data when the switch closes.

```
dataIn = read(d,seconds(1),"OutputFormat","Matrix");
```

See Also

Related Examples

- "Multiple-Device Synchronization Using USB or PXI Devices" on page 13-7
- "Multiple-Chassis Synchronization with CompactDAQ Devices" on page 13-12

More About

- "Synchronization" on page 13-2
- "Trigger Connections" on page 12-2

Clock Connections

In this section...

"When to Use Clocks" on page 12-5
"Import Scan Clock from External Source" on page 12-5
"Export Scan Clock to External System" on page 12-5

When to Use Clocks

Use clocks to synchronize operations on all connected devices in the DataAcquisition. You connect a clock source to a clock destination. A clock source can be either external, where the clock signal comes from a source outside a DataAcquisition, or on a device and terminal pair within a DataAcquisition. Destination devices can be external, where the signals are received outside the DataAcquisition, or devices within the DataAcquisition. For more information, see "Source and Destination Devices" on page 13-3.

Note You cannot use trigger and clock connections with audio channels.

Import Scan Clock from External Source

To import a scan clock from an external source, you must connect the external clock to a terminal and device pair on a device in your DataAcquisition. Two circumstances of externally clocked synchronization include:

- Synchronizing operations on all devices within a DataAcquisition by sharing the clock on a device within the DataAcquisition or an external clock
- Synchronizing operations on all devices within a DataAcquisition and some external devices, by sharing an external clock

Note Importing an external clock does not automatically set the scan rate of your DataAcquisition. Manually set the DataAcquisition `Rate` property value to match the expected external clock frequency.

Export Scan Clock to External System

This example shows how to add a scan clock to a device and output the clock to a device outside your DataAcquisition, which is connected to an oscilloscope. The scan clock controls the operations on the external device.

Create a DataAcquisition and add a voltage input channel from an NI USB-6211 with device ID `Dev1`.

```
d = daq("ni");
addinput(d,"Dev1",0,"Voltage")
```

Add a clock to the DataAcquisition, to export an external scan clock sourced at terminal `PFI6` on `Dev1`, and physically connect it to an external destination.

```
c = addclock(d,"ScanClock","Dev1/PFI6","External")
```

```
c =

  Clock with properties:

         Source: 'Dev1/PFI6'
    Destination: 'External'
           Type: ScanClock
```

Acquire data and store it in `dataIn`.

```
dataIn = read(d,seconds(1),"OutputFormat","Matrix");
```

See Also

Related Examples

- “Multiple-Device Synchronization Using USB or PXI Devices” on page 13-7
- “Multiple-Chassis Synchronization with CompactDAQ Devices” on page 13-12

More About

- “Synchronization” on page 13-2

Synchronization

- “Synchronization” on page 13-2
- “Multiple-Device Synchronization Using USB or PXI Devices” on page 13-7
- “Synchronize with PFI on CompactDAQ Chassis Without Terminals” on page 13-11
- “Multiple-Chassis Synchronization with CompactDAQ Devices” on page 13-12
- “Synchronize DSA Devices” on page 13-13

Synchronization

Synchronization of data acquisition operations between multiple channels or devices has two aspects:

- Start trigger: The signal to initiate all operations
- Scan clock: The timing for repeated generation or acquisition of signals at a clocked rate

Synchronization can involve the coordination of triggering, clocking, or both. To synchronize the start of operations on multiple channels or devices, they must use a shared start trigger. To synchronize the clocked scanning operations on multiple channels or devices, they must use a shared scan clock.

The following definitions summarize some concepts of synchronization:

Type of Synchronization	Description
Start trigger synchronization	Channels or devices are configured to simultaneously start their operations from a shared start trigger.
Scan clock synchronization	Channels or devices use a shared scan clock to generate or measure signals.
Perfect synchronization	Channels or devices use both a shared start trigger and a shared scan clock. This does not imply a specific skew or latency performance between devices or between channels on a device.
Automatic synchronization	The default start trigger synchronization and scan clock synchronization supported by a DataAcquisition, the driver, and the hardware. This is the extent of synchronization provided by a DataAcquisition without any explicit synchronization configuration. When a DataAcquisition starts, it sends a start trigger signal to all connected channels in the DataAcquisition. The driver and device might support synchronization from that moment forward. For example, in some devices all channels use the same internal scan clock and a shared start trigger, so they are automatically synchronized without further configuration of the DataAcquisition.

Shared Triggers and Shared Scan Clocks

Typical data acquisition devices provide synchronization between their channels of the same subsystem. For example, all the analog input channels on one card use a shared scan clock. A DataAcquisition can configure start trigger and scan clock connections for wider synchronization needs. Use shared start triggers and shared scan clocks to synchronize data between:

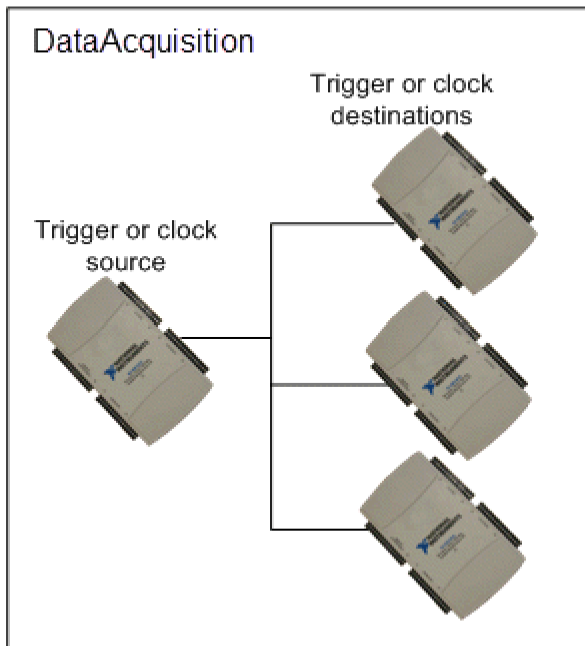
- Multiple subsystems in a device (analog input, analog output, counter input, etc.)
- Multiple devices
- Multiple CompactDAQ or PXI chassis

Note Counter output channels run independently and are unaffected by synchronization connections.

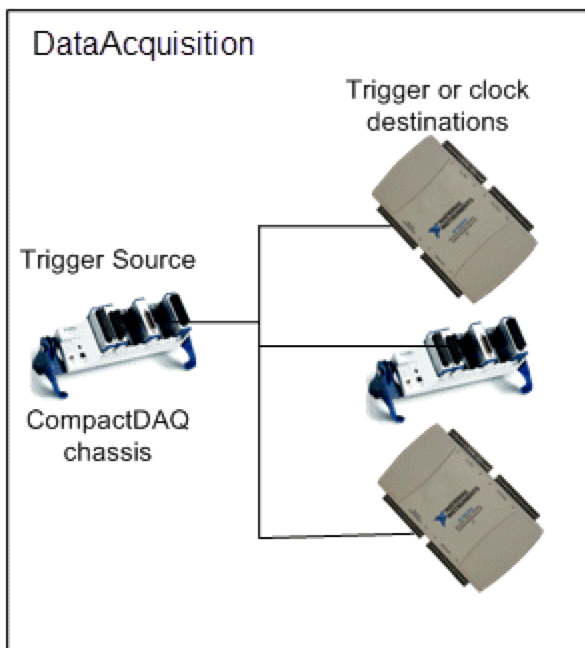
Source and Destination Devices

You can share start triggers and scan clock connections to synchronize operations within a DataAcquisition. Synchronization connections can be:

- Devices in a DataAcquisition connected to a start trigger or scan clock source on another device in the DataAcquisition



- Devices and chassis in a DataAcquisition connected to a start trigger or scan clock source on another device in the DataAcquisition



A source device and terminal pair generates the synchronization signal and is connected to the destination device and terminal pairs. You must physically connect the source and destination terminals, unless they are internally connected. Check your device specifications for more information. Synchronization connections are added from the source device to one or more destination devices.

- The source device provides the start trigger or scan clock signals.
- The destination device receives a start trigger or scan clock signal.

For example, if you determine that a terminal on `Dev1` will provide a start trigger and a terminal on `Dev2` will receive that trigger, then `Dev1` becomes your source device and `Dev2` your destination device. You can have multiple destinations for your trigger and clock connections, but only one source.

Use `addtrigger` to add start trigger connections, and `addclock` to add a scan clock connection to your DataAcquisition.

Automatic Synchronization

In most cases, a DataAcquisition automatically starts all its devices at the same time when you start an operation. You must configure them to start synchronously when devices are not on a single chassis and do not share a clock. If you have not configured synchronization on such devices, the start operation reduces the latency between devices, running them very close together to achieve near-simultaneous signals. However, devices are automatically and perfectly synchronized in the DataAcquisition if they are:

- Subsystems on a single device in the DataAcquisition. This synchronizes your analog input, analog output, and counter input channels.

  Note Counter output channels run independently and are unaffected by synchronization connections.
- Modules on a single CompactDAQ chassis in the DataAcquisition.
- PXI modules synchronized with a reference clock on a PXI chassis. For perfect synchronization, you must share a trigger as well. See “Acquire Synchronized Data Using PXI Devices” on page 13-9 for more information.

Synchronization Scenarios

You must employ different techniques for synchronization, depending on the configurations of your channels, devices, and chassis. The following sections describe these different scenarios.

Multiple Channels on the Same Device or Module

In this topic, hardware that performs the signal conversion when not plugged into a chassis is referred to as a *device*; this includes USB devices. When the conversion hardware is a card plugged into a chassis, it is usually referred to as a *module*.

Data Acquisition Toolbox DataAcquisition software is based on the assumption that all channels of the same acquisition device or module use the same internal scan clock and start trigger. As such, these channels meet the requirements for perfect synchronization. For most vendors, this includes digital channels, analog channels, and counter input channels, but does not include counter output channels.

The following topics illustrate this scenario, providing automatic synchronization between multiple channels.

- “Acquire Data from Multiple Channels using an MCC Device” on page 18-22
- “Acquiring and Generating Data at the Same Time with Digilent Analog Discovery” on page 18-73

Exceptions: Some devices do not support setting the source of the start trigger or do not internally route start trigger signals between subsystems. These include National Instruments myDAQ and USB-6002. In such devices, only channels of the same subsystem support start trigger synchronization by default.

Multiple Modules in the Same CompactDAQ Chassis

Modules in the same CompactDAQ chassis use the chassis scan clock and start trigger. The Data Acquisition Toolbox DataAcquisition interface configures the chassis scan clock rate and issues the start trigger signal. The chassis in turn provides synchronized signals to its modules.

The following examples illustrate this scenario, providing synchronization between multiple modules in the same chassis without external connections or extra programming.

- “Simultaneously Acquire Data and Generate Signals” on page 18-60
- “Count Pulses on a Digital Signal Using NI Devices” on page 18-101
- “Measure Frequency Using NI Devices” on page 18-104
- “Measure Pulse Width Using NI Devices” on page 18-106

Exceptions: Some CompactDAQ modules have their own onboard clocks, for example, DSA modules.

Multiple Modules in the Same PXI Chassis

Modules in a PXI chassis share a common scan clock, but a Data Acquisition Toolbox DataAcquisition does not synchronize the start trigger for multiple modules in the chassis by default. The start triggers of multiple DSA modules can be synchronized using the `AutoSyncDSA` property, while other PXI modules require an external trigger connection for start trigger synchronization.

The following topics illustrate these scenarios, showing how to synchronize start triggers on multiple modules.

- “Synchronize DSA Devices” on page 13-13
- “Synchronize DSA PXI Devices Using AutoSyncDSA” on page 13-8
- “Acquire Synchronized Data Using PXI Devices” on page 13-9

Multiple Devices Without Chassis or in Different Chassis

This scenario represents multiple devices or modules in their most independent configuration. The configuration could be multiple USB devices, for example, or modules in separate chassis. Neither the start triggers nor the scan clocks of these devices are synchronized by default.

The following topics illustrates these scenarios, showing how to synchronize start triggers and scan clocks on multiple devices without chassis or in different chassis, by way of an external connection.

- “Acquire Synchronized Data Using USB Devices” on page 13-7
- “Multiple-Chassis Synchronization with CompactDAQ Devices” on page 13-12

- “Synchronize Counter Outputs from Multiple Devices” on page 13-8
- “Acquire Data from Two Devices at Different Rates” on page 18-136

See Also

More About

- “Multiple-Device Synchronization Using USB or PXI Devices” on page 13-7
- “Synchronize DSA Devices” on page 13-13

Multiple-Device Synchronization Using USB or PXI Devices

You can synchronize multiple devices in a DataAcquisition using a shared scan clock and shared start trigger. You can synchronize devices using either PFI or RTSI lines.

Requirement You must register your RTSI cable using the National Instruments Measurement & Automation Explorer.

Acquire Synchronized Data Using USB Devices

This example shows how to acquire synchronized voltage data from multiple devices using a shared start trigger and a shared scan clock. Analog input channels on all three devices are connected to the same function generator.

Create a DataAcquisition and add one voltage input channel from each device:

- NI USB-6211 with device ID `Dev1`
- NI USB 6218 with device ID `Dev2`
- NI USB 6255 with device ID `Dev3`

```
d = daq("ni");
addinput(d,"Dev1",0,"Voltage")
addinput(d,"Dev2",0,"Voltage")
addinput(d,"Dev3",0,"Voltage")
```

Choose terminal `PFI4` on `Dev1` as the start trigger source. Connect the trigger source to the destination terminals `PFI0` on `Dev2` and `PFI0` on `Dev3`.

```
addtrigger(d,"Digital","StartTrigger","Dev1/PFI4","Dev2/PFI0")
addtrigger(d,"Digital","StartTrigger","Dev1/PFI4","Dev3/PFI0")
```

Choose terminal `PFI5` on `Dev1` as the scan clock source. Connect it to destination terminals `PFI1` on `Dev2`, and `PFI1` on `Dev3`.

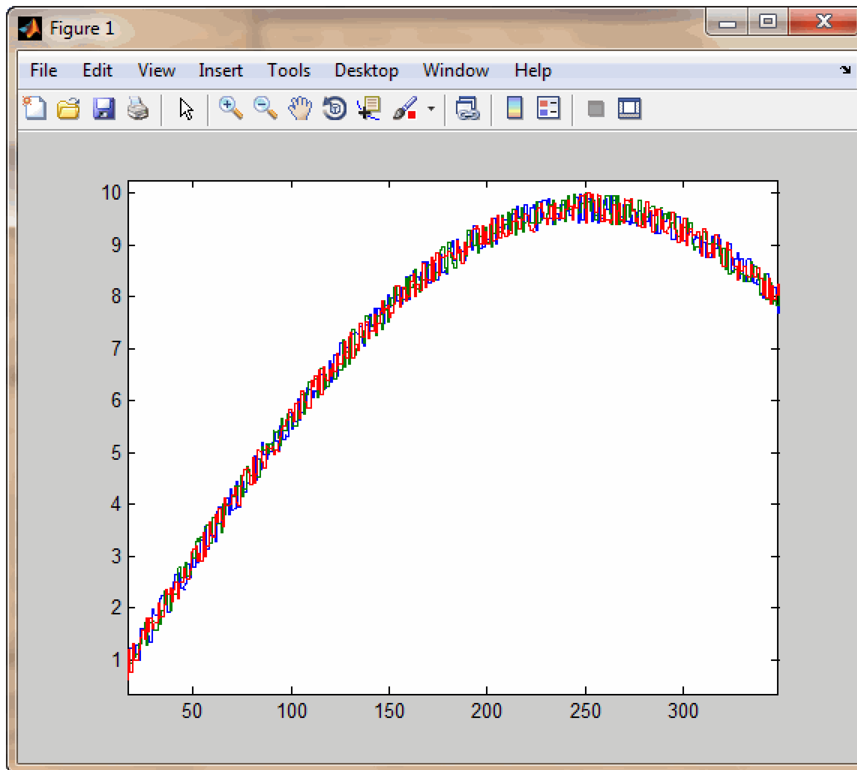
```
addclock(d,"ScanClock","Dev1/PFI5","Dev2/PFI1")
addclock(d,"ScanClock","Dev1/PFI5","Dev3/PFI1")
```

Acquire data and assign it to `dataIn`.

```
dataIn = read(d,350,"OutputFormat","Matrix");
```

Plot the data.

```
plot(dataIn)
```



All channels are connected to the same function generator, so the plot displays overlapping signals, indicating synchronization.

Synchronize Counter Outputs from Multiple Devices

This example shows how to synchronize the start trigger of counter output operations from two channels on different devices.

```
d = daq("ni");
addoutput(d,"Dev1","ctr0","PulseGeneration")
addoutput(d,"Dev2","ctr0","PulseGeneration")
addtrigger(d,"Digital","StartTrigger","Dev1/PFI0","Dev2/PFI0")
start(d)
```

This example uses two USB or PCI devices, but could be modified for channels across CompactDAQ or PXI chassis. If you have counter output CompactDAQ modules in the same chassis, it is not necessary to call `addtrigger`; but it is required for multiple modules in the same PXI chassis.

Synchronize DSA PXI Devices Using AutoSyncDSA

This example shows how to acquire synchronized data from two Dynamic Signal Analyzer (DSA) PXI devices, NI PXI-4462 and NI PXI-4461, using the `AutoSyncDSA` property.

Create a DataAcquisition and add one voltage analog input channel from each of the two PXI devices

```
d = daq("ni");
addinput(d,"PXI1Slot2",0,"Voltage")
addinput(d,"PXI1Slot3",0,"Voltage")
```

Acquire data in the foreground without synchronizing the channels:

```
[data,time] = read(d,seconds(1),"OutputFormat","Matrix");
plot(time,data)
```

The data returned is not synchronized.

Synchronize the two channels using the `AutoSyncDSA` property:

```
d.AutoSyncDSA = true;
```

Acquire data in the foreground and plot it:

```
[data,time] = read(d,seconds(1),"OutputFormat","Matrix");
plot(time,data)
```

The data is now synchronized.

Acquire Synchronized Data Using PXI Devices

This example shows how to acquire voltage data from two PXI devices on the same chassis, using a shared start trigger to synchronize operations within your DataAcquisition. PXI devices have a shared reference clock that automatically synchronizes scan clocking. You need to add only start trigger connections to synchronize operations in your DataAcquisition with PXI devices. Analog input channels on all devices are connected to the same function generator.

Create a DataAcquisition and add one voltage input channel from each NI-PXI 4461 device with IDs `PXI1Slot2` and `PXI1Slot3`.

```
d = daq("ni");
addinput(d,"PXI1Slot2",0,"Voltage")
addinput(d,"PXI1Slot3",0,"Voltage")
```

Add a start trigger connection to terminal `PXI_Trig0` on `PXI1Slot2` and connect it to terminal `PXI_Trig0` on `PXI1Slot3`. PXI cards are connected through the chassis backplane, so you do not have to wire them physically.

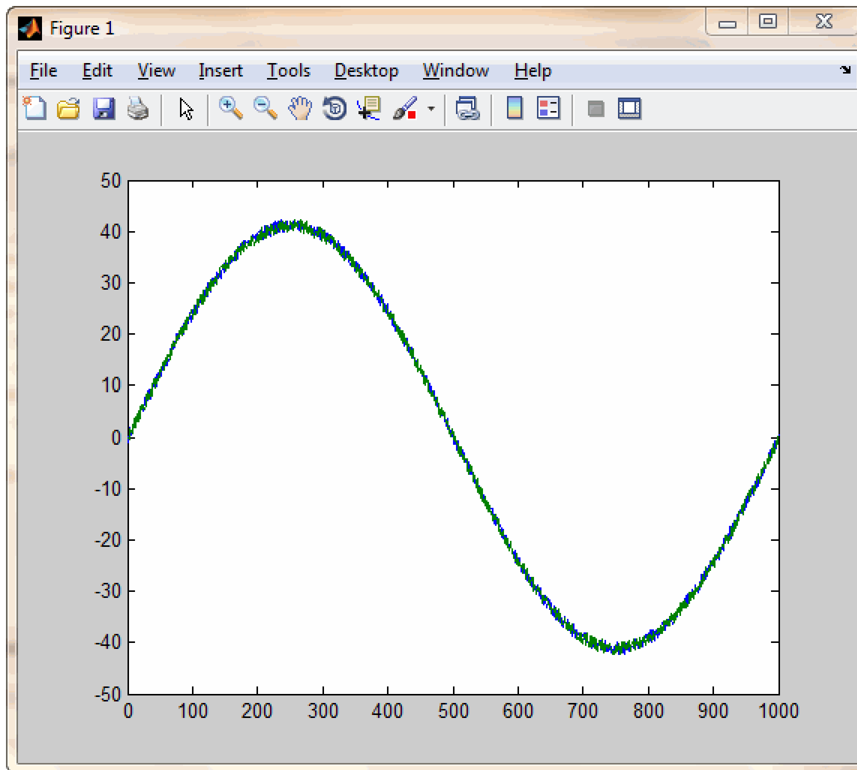
```
addtrigger(d,"Digital","StartTrigger","PXI1Slot2/PXI_Trig0","PXI1Slot3/PXI_Trig0")
```

Acquire data and assign it to `dataIn`.

```
dataIn = read(d,seconds(1),"OutputFormat","Matrix");
```

Plot the data.

```
plot(dataIn)
```



All channels are connected to the same function generator and have a shared reference clock. The signals overlap, indicating synchronization.

See Also

More About

- “Multiple-Chassis Synchronization with CompactDAQ Devices” on page 13-12
- “Generate Pulse Data on a Counter Channel” on page 8-6

Synchronize with PFI on CompactDAQ Chassis Without Terminals

This example shows how to use the external trigger and external clock functionality on a CompactDAQ 9174 chassis without PFI terminals, by using the PFI terminals on digital I/O CompactDAQ modules 9402 with ID `cDAQ2Mod3` and 9201 with ID `cDAQ2Mod4`.

Some CompactDAQ chassis (e.g., NI 9174 and 9172) do not support built-in triggers, because they do not have external BNC PFI connectors on the chassis itself. However, the PFI pins for these chassis can be accessed through a digital module such as the NI 9402.

Add a start trigger from an external source.

```
d = daq("ni");
addinput(d,"cDAQ2Mod4","ai0","Voltage")
addtrigger(d,"Digital","StartTrigger","External","cDAQ2Mod3/PFI0")
[data,timestamps] = read(d,seconds(1),"OutputFormat","Matrix");
plot(timestamps,data)
```

Use an external scan clock from a function generator providing a 100 kHz clock to terminal PFI1 on NI 9402.

```
d = daq("ni");
addinput(d,"cDAQ2Mod3","Port0/Line2","Digital")
addclock(d,"ScanClock","External","cDAQ2Mod3/PFI1)
d.Rate = 100E+3;
[data,timestamps] = read(d,seconds(1),"OutputFormat","Matrix");
plot(timestamps,data);
```

Tip If you want your devices to run at multiple scan rates, use two separate DataAcquisition objects with different scan rate settings.

See Also

Related Examples

- "Start a Multi-Trigger Acquisition on an External Event" on page 18-128
- "Acquire Data from Two Devices at Different Rates" on page 18-136

Multiple-Chassis Synchronization with CompactDAQ Devices

This example shows how to acquire voltage data from two devices, each on a separate CompactDAQ chassis, using a shared trigger and clock to synchronize operations within your DataAcquisition.

You can synchronize multiple CompactDAQ chassis in a DataAcquisition using one chassis to provide clocking and triggering for all chassis in the DataAcquisition. Clock and trigger sources are attached to terminals on the chassis, itself. All modules on the chassis as well as other connected devices, are synchronized using these signals.

Create a DataAcquisition and add channels. Add one voltage input channel each from the two NI 9201 devices with IDs `cDAQ1Mod1` and `cDAQ2Mod1`.

```
d = daq("ni");
addinput(d,"cDAQ1Mod1",0,"Voltage")
addinput(d,"cDAQ2Mod1",0,"Voltage")
```

Choose terminal `PFI0` on `cDAQ1` as your trigger source, and connect it to destination terminal `PFI0` on `cDAQ2`. Make sure the wiring on the hardware runs between these two terminals. Note that you are using the chassis and terminal pair here, not device and terminal pair.

```
addtrigger(d,"Digital","StartTrigger","cDAQ1/PFI0","cDAQ2/PFI0")
```

Choose terminal `PFI1` on `cDAQ1` as your clock source, and connect it to destination terminal `PFI1` on `cDAQ2`. Make sure the wiring on the hardware runs between these terminals.

```
addclock(d,"ScanClock","cDAQ1/PFI1","cDAQ2/PFI1")
```

Acquire data and assign it to `dataIn`.

```
dataIn = read(d,seconds(1),"OutputFormat","Matrix");
```

See Also

More About

- “Synchronize Counter Outputs from Multiple Devices” on page 13-8

Synchronize DSA Devices

The Digital Signal Analyzer (DSA) product family is designed to make highly accurate audio frequency measurements. You can synchronize other PCI and PXI product families using "Trigger Connections" on page 12-2 and "Clock Connections" on page 12-5. To synchronize PXI and PCI families of DSA devices you need to use a sample clock with time-based synchronization or a reference clock time-based synchronization. The DataAcquisition `AutoSyncDSA` property allows you to automatically enable both homogeneous and heterogeneous synchronization between PCI and PXI device families. The `AutoSyncDSA` property automatically configures all the necessary clocks, triggers, and sync pulses needed to synchronize DSA devices in your DataAcquisition.

PXI DSA Devices

PXI devices are synchronized using the PXI chassis backplane, which includes timing and triggering buses. You can automatically synchronize these device series both homogeneously (within the same series) and heterogeneously (across separate series) in the same DataAcquisition, including the following:

- PXI/e 446x series
- PXI/e 449x series
- PXI 447x series

Hardware Restrictions

Before you synchronize, ensure that your device combinations adhere to these hardware restrictions:

PXI/e 446x and 449x Series

Chassis restriction

You can synchronize these series using either a PXI or a PXIe chassis. Make sure all your modules are on the same chassis.

Slot placement restriction

You can use any slot on the chassis that supports your module.

PXI 447x Series

Chassis restriction

You can synchronize this series both homogeneously and heterogeneously only on a PXI chassis. You can use them on a PXIe chassis to acquire unsynchronized data.

Slot placement restriction

On the PXI chassis, only the system timing slot can drive the trigger bus. Refer to your device manual to find the system timing slot. This image shows the system timing slot on a PXIe 1062Q chassis.

- Homogeneous synchronization: You can synchronize PXI 447x devices homogeneously if one device is plugged into the system timing slot of a PXI chassis.
- Heterogeneous synchronization:

- You can synchronize a PXI 447x device with a PXI 446x device when the 446x is plugged into the system timing slot of a PXI chassis.
- You cannot synchronize PXI 447x devices with PXI 449x devices.
- You cannot use hybrid-slot compatible 446x devices.

DSA Device Compatibility Table

	446x Series	447x Series	449x Series
446x Series	✓	• PXI chassis only • Standard 446x device, not hybrid-slot compatible • 446x device in system timing slot	✓
447x Series	• PXI chassis only • Standard 446x device, not hybrid-slot compatible • 446x device in system timing slot	• PXI chassis only • One device in system timing slot	
449x Series	✓		✓

PCI DSA Devices

PCI devices are synchronized use the RTSI cable. You can automatically synchronize these device series both homogeneously (within the same series) and heterogeneously (across separate series) in the same DataAcquisition when they are connected with a RTSI cable. Support includes the following:

- PCI 446x series
- PCI 447x series

Note If you are synchronizing PCI devices make sure you register the RTSI cables in Measurement and Automation Explorer. For more information, see the NI knowledge base article Real-Time System Integration (RTSI) and Configuration Explained.

Synchronize DSA PCI Devices

This example shows how to acquire synchronized data from two DSA PCI devices, NI PCI-4461 and NI PCI-4462.

Connect the two devices with a RTSI cable.

Register your RTSI cable in Measurement and Automation Explorer.

Create a DataAcquisition and add one voltage analog input channel from each of the two PCI devices

```
d = daq("ni");
addinput(d,"Dev1",0,"Voltage")
addinput(d,"Dev2",0,"Voltage")
```

Synchronize the two channels using the `AutoSyncDSA` property:

```
d.AutoSyncDSA = true;
```

Acquire data in the foreground and plot it:

```
[data,time] = read(d,seconds(1),"OutputFormat","Matrix");
plot(time,data)
```

Handle Filter Delays with DSA Devices

DSA devices have a built in digital filter. You must account for filter delays when synchronizing between heterogeneous devices. Refer to your device manuals for filter delay information. For more information, see the NI knowledge base article Synchronized Data Delayed When Using DSA Devices.

Example 13.1. Account for Filter Delays

This example shows how to account for filter delays when you use the same sine wave to acquire from two different channels from two different PXI devices. Perfectly synchronized channels will show zero phase lag between the two acquired signals.

Create a DataAcquisition and add two analog input channels with `'Voltage'` measurement type, from National Instruments PXI-4462 and NI PXI-4472.

```
d = daq("ni");
ch1 = addinput(d,"PXI1Slot2",0,"Voltage");
ch2 = addinput(d,"PXI1Slot3",0,"Voltage");
```

Acquire unsynchronized data and plot it:

```
[data,time] = read(d,seconds(1),"OutputFormat","Matrix");
plot(time,data)
```

Use AutoSyncDSA to automatically configure the triggers, clocks, and sync pulses of the channels to synchronize the devices:

```
d.AutoSyncDSA = true;
```

Acquire synchronized data:

```
[data,time] = read(d,seconds(1),"OutputFormat","Matrix");
plot(time,data)
```

The data sheets for the NI PXI 4462 and PXI-4472 indicate a phase lag for each to be 63 and 38 samples, respectively, when the `EnhancedAliasRejectionEnable` property is disabled. Check to make sure that this property is set to `false` or `0` on both channels:

```
ch1.EnhancedAliasRejectionEnable
```

```
ans =

    0
```

```
ch2.EnhancedAliasRejectionEnable
```

```
ans =

    0
```

Visually verify in the plotted data that the phase difference is 25 samples apart.

See Also

More About

- "Synchronize DSA PXI Devices Using AutoSyncDSA" on page 13-8

14

Transition Your Code to New Interfaces

Transition Your Code from Session to DataAcquisition Interface

This topic helps you transition your code from the session interface to the DataAcquisition interface.

Transition Common Workflow Commands

This table lists the session interface commands for common workflows and their corresponding DataAcquisition interface commands.

To do this	Session Commands	DataAcquisition Commands
Find supported hardware available to your system	`daq.getDevices`	`daqlist`
Reset toolbox to initial state	`daqreset`	`daqreset`
Create interface object	`s = daq.createSession('ni')`	`d = daq("ni");`
Add analog input channel	`addAnalogInputChannel(s,'Dev1',1,'Voltage')`	`addinput(d,"Dev1","ai1","Voltage")`
Add analog output channel	`addAnalogOutputChannel(s,'Dev1',0,'Current')`	`addoutput(d,"Dev1","ao1","Current")`
Add digital input line	`addDigitalChannel...` `(s,'Dev1','Port0/Line0:1','InputOnly')`	`addinput(d,"Dev1","port0/line1","Digital"`
Add counter input channel	`addCounterInputChannel...` `(s,'Dev1','ctr0','EdgeCount')`	`addinput(d,"Dev1","ctr0","EdgeCount");`
Set data scan rate	`s.Rate = 48000`	`d.Rate = 48000;`
Queue data for output	`queueOutputData(s,outputSignal)`	Necessary only for background operation. `preload(d,outputSignal)`

Start synchronous foreground operation that blocks MATLAB	Acquire input signal.
	
	`s.DurationInSeconds = 5;`
	`inData = startForeground(s);`
		Duration is an input argument to the `read` function.
		
		`inData = read(d,seconds(5));`

	Generate output signal.
	
	`queueOutputData(s,outputSignal);`
	`startForeground(s);`
		Output data is provided directly to the `write` function.
		
		`write(d,signalData)`

	Generate and acquire signals simultaneously.
	
	`queueOutputData(s,outputSignal);`
	`inData = startForeground(s);`
		Use `readwrite` for simultaneous input and output.
		
		`inData = readwrite(d,outputSignal);`

Start asynchronous background read operation that runs without blocking MATLAB	`s.DurationInSeconds = 5;`
	`startBackground(s)`
		`start(d,"Duration",seconds(5))`
		⋮
		`inData = read(d,"all")`

Start asynchronous background write operation that runs without blocking MATLAB	`queueOutputData(s,outputSignal);` `startBackground(s);`	`preload(d,outputSignal);` `start(d)`

To do this	Session Commands	DataAcquisition Commands
Start continuous background operation	`s.IsContinuous = true;` `inData = startBackground(s);`	Continuous operation is specified by the `start` function. `start(d,"Continuous")`

Start continuous background write operation	`lh = addlistener(s,'DataRequired',@queueMoreData);`
	`s.IsContinuous = true;`
	`queueOutputData(s,outputSignal);`
	`startBackground(s);`
		If data is preloaded, generation begins with the `start` function.
		
		`d.ScansRequiredFcn = @writeMoreData;`
		`preload(d,outputSignal);`
		`⋮`
		`start(d,"Continuous")`
		
		If data is not preloaded, generation begins with the `write` function.
		
		`d.ScansRequiredFcn = @writeMoreData;`
		`start(d,"Continuous")`
		`⋮`
		`write(d,outputSignal)`

Configure callbacks	`listenDA = addlistener(s,'DataAvailable',@logData);`
	`listenDR = addlistener(s,'DataRequired',@queueMoreData);`
	`listenEO = addlistener(s,'ErrorOccurred',@handleError);`
		`d.ScansAvailableFcn = @logData;`
		`d.ScansRequiredFcn = @writeMoreData;`
		`d.ErrorOccurredFcn = @handleError;`

Specify external trigger	`addTriggerConnection...` `(s,'External','Dev3/PFI0','StartTrigger');`	`addtrigger(d,"Digital","StartTrigger","Ex`

Specify input signal range	`ch = addAnalogInputChannel...`
	`(s,'Dev1',1,'Voltage');`
	`ch.Range = [-5 5];`
		`ch = addinput(d,"Dev1","ai4","Voltage");`
		`ch.Range = [-5 5];`

Acquire Analog Data

Session Interface

Using the session interface, you create a vendor session and add channels to the session. You can use any device or chassis from the same vendor available to your system and can add a combination of analog, digital, and counter input and output channels. All the channels operate together when you start the session.

1. Find hardware available to your system.

```
d = daq.getDevices
```

2. Create a session for National Instruments devices.

```
s = daq.createSession('ni');
```

3. Set the session scan rate to 8000.

```
s.Rate = 8000
```

4. Add an analog input channel for the device with ID Dev1 for voltage measurement, and then start the acquisition.

```
addAnalogInputChannel(s,'Dev1',1,'Voltage');
startForeground(s);
```

DataAcquisition Interface

1. Find hardware available to your system.

```
devs = daqlist
```

2. Create a DataAcquisition for National Instruments devices.

```
d = daq("ni");
```

3. Set the DataAcquisition scan rate to 8000.

```
d.Rate = 8000
```

4. Add an analog input channel for the device with ID `Dev1` for voltage measurement, and then start the acquisition.

```
addinput(d,"Dev1","ai1","Voltage");
data = read(d,4000);
```

Scan results are returned to the timetable `data`.

Use Triggers

Acquire analog data using hardware triggers.

Session Interface

You can specify an external event to trigger data acquisition using the session interface.

1. Create a session and add two analog input channels.

```
s = daq.createSession('ni');
ch = addAnalogInputChannel(s,'Dev1',0:1,'Voltage');
```

2. Configure the terminal and range of the channels in the session.

```
ch(1).TerminalConfig = 'SingleEnded';
ch(1).Range = [-10.0 10.0];
ch(2).TerminalConfig = 'SingleEnded';
ch(2).Range = [-10.0 10.0];
```

3. Create an external trigger connection and set the trigger to run one time.

```
addTriggerConnection(s,'External','Dev1/PFI0','StartTrigger');
s.Connections(1).TriggerCondition = 'RisingEdge';
s.TriggersPerRun = 1;
```

4. Set the rate and the duration of the acquisition.

```
s.Rate = 50000;
s.DurationInSeconds = 0.01;
```

5. Acquire data in the foreground and plot the data.

```
[data,timestamps] = startForeground(s);
plot(timestamps,data)
```

DataAcquisition Interface

1. Create a DataAcquisition and add two analog input channels.

```
d = daq("ni");
ch = addinput(d,"Dev1",0:1,"Voltage");
```

2 Configure the terminal configuration and range of the channels in the DataAcquisition.

```
ch(1).TerminalConfig = "SingleEnded";
ch(1).Range = [-10.0 10.0];
ch(2).TerminalConfig = "SingleEnded";
ch(2).Range = [-10.0 10.0];
```

3 Create an external trigger connection and set the trigger to run one time.

```
addtrigger(d,"Digital","StartTrigger","Dev1/PFI0","External");
d.DigitalTriggers(1).Condition = "RisingEdge";
d.NumDigitalTriggersPerRun = 1;
```

4 Set the scan rate of the acquisition.

```
d.Rate = 50000;
```

5 Acquire data in the foreground for 0.01 seconds and plot the data from all channels.

```
data = read(d,seconds(0.01));
plot(data.Time, data.Variables)
```

Initiate an Operation When Number of Scans Exceeds Specified Value

You can specify your acquisition to watch for a specified number of scans to occur and then initiate some operation.

Session Interface

The session interface uses listeners and events to trigger certain actions. The `NotifyWhenDataAvailableExceeds` property can fire a `DataAvailable` event. A listener defines the operation to execute at that time.

1 Create an acquisition session, add an analog input channel.

```
s = daq.createSession('ni');
addAnalogInputChannel(s,'Dev1','ai0','Voltage');
```

2 Set the scan rate to 800,000 scans per second, which automatically sets the `DataAvailable` notification to automatically fire 10 times per second.

```
s.Rate = 800000;
s.NotifyWhenDataAvailableExceeds

ans =
    80000
```

3 Increase `NotifyWhenDataAvailableExceeds` to 160,000.

```
s.NotifyWhenDataAvailableExceeds = 160000;
```

4 Add a listener to determine the function to call when the event occurs.

```
L = addlistener(s,'DataAvailable', ...
       @(src,event)readAndLogData(src));
```

DataAcquisition Interface

The DataAcquisition interface uses callback functions that execute at occurrences determined by certain properties. The `ScansAvailableFcnCount` property determines when to initiate the callback function defined by `ScansAvailableFcn`.

1. Create a DataAcquisition interface and add an analog input channel.

   ```
   d = daq("ni");
   ch = addinput(d,"Dev1",1,"Voltage");
   ```
2. Set the scan rate to 800,000 scans per second, which automatically adjusts the `ScansAvailableFcnCount` property.

   ```
   d.Rate = 800000;
   d.ScansAvailableFcnCount

       80000
   ```
3. Increase `ScansAvailableFcnCount` to 160,000.

   ```
   d.ScansAvailableFcnCount = 160000;
   ```
4. Identify a callback function for when the count occurs.

   ```
   d.ScansAvailableFcn = @readAndLogData;
   ```

Analog Output Generator Code

To compare session interface code and DataAcquisition interface code you can use the code generated by the Analog Output Generator in MATLAB releases R2019b and R2020a. In both these examples, the generator created a 10 Hz test signal sine wave for 1 second on a single channel of a National Instruments USB-6211.

```
%% Auto-generated by Data Acquisition Toolbox Analog Output Generator in MATLAB R2020a.
%% Create DataAcquisition Object
% Create a DataAcquisition object for the specified vendor.

d = daq("ni");
%% Add Channels
% Add channels and set channel properties, if any.

addoutput(d,"Dev1","ao0","Voltage");
%% Set DataAcquisition Rate
% Set scan rate.

d.Rate = 250000;
%% Define Test Signal
% Create a test sine wave signal of specified peak-to-peak amplitude for each
% channel.

amplitudePeakToPeak_ch1 = 20;

sineFrequency = 10; % 10 Hz
totalDuration = 1; % 1 seconds

outputSignal = [];
outputSignal(:,1) = createSine(amplitudePeakToPeak_ch1/2, ...
                        sineFrequency, d.Rate, "bipolar", totalDuration);
outputSignal(end+1,:) = 0;
%% Generate Signal
% Write the signal data.

write(d,outputSignal);
%% Clean Up
% Clear all DataAcquisition and channel objects.

clear d outputSignal
%% Create Test Signal
```

```
% Helper function for creating test sine wave signal.

function sine = createSine(A, f, sampleRate, type, duration)

numSamplesPerCycle = floor(sampleRate/f);
T = 1/f;
timestep = T/numSamplesPerCycle;
t = (0 : timestep : T-timestep)';

if type == "bipolar"
    y = A*sin(2*pi*f*t);
elseif type == "unipolar"
    y = A*sin(2*pi*f*t) + A;
end

numCycles = round(f*duration);
sine = repmat(y,numCycles,1);
end

%% Auto-generated by Data Acquisition Toolbox Analog Output Generator in MATLAB R2019b
%% Create Data Acquisition Session
% Create a session for the specified vendor.

s = daq.createSession('ni');
%% Set Session Properties
% Set properties that are not using default values.

s.Rate = 250000;
%% Add Channels to Session
% Add channels and set channel properties.

addAnalogOutputChannel(s,'Dev1','ao0','Voltage');
%% Define Test Signal
% Create a test sine wave signal of specified peak-to-peak amplitude for each
% channel.

amplitudePeakToPeak_ch1 = 20;

sineFrequency = 10; % 10 Hz
totalDuration = 1; % 1 seconds

outputSignal(:,1) = createSine(amplitudePeakToPeak_ch1/2, ...
                        sineFrequency, s.Rate, 'bipolar', totalDuration);
outputSignal(end+1,:) = 0;
%% Queue Signal Data
% Make signal data available to session for generation.

queueOutputData(s,outputSignal);
%% Generate Signal
% Start foreground generation

startForeground(s);
%% Clean Up
% Clear the session and channels.

clear s outputSignal
%% Create Test Signal
% Helper function for creating test sine wave signal.

function sine = createSine(amplitude, frequency, sampleRate, type, duration)

sampleRatePerCycle = floor(sampleRate/frequency);
period = 1/frequency;
s = period/sampleRatePerCycle;
t = (0 : s : period-s)';

if strcmpi(type, 'bipolar')
    y = amplitude*sin(2*pi*frequency*t);
elseif strcmpi(type, 'unipolar')
    y = amplitude*sin(2*pi*frequency*t) + amplitude;
end

numCycles = round(frequency*duration);
sine = repmat(y, numCycles, 1);
end
```

Previous Interface Help

The DataAcquisition interface is supported in R2020a and later. If you are using an earlier release, use the session interface instead. For more information and examples of the session interface, see Data Acquisition Toolbox Documentation (R2019b).

15

Functions

addbidirectional

Package: daq.interfaces

Add digital bidirectional channel to device interface

Syntax

```
addbidirectional(d,deviceID,channelID,"Digital")
ch = addbidirectional( ___ )
[ch,idx] = addbidirectional( ___ )
```

Description

`addbidirectional(d,deviceID,channelID,"Digital")` adds the digital bidirectional channel `channelID` of device `deviceID` to the specified DataAcquisition interface, `d`.

The channel information is available from the DataAcquisition `Channels` property.

`ch = addbidirectional( ___ )` adds the channel and returns a channel object.

`[ch,idx] = addbidirectional( ___ )` adds the channel and also returns the channel index from the DataAcquisition interface. The channel index reflects only the sequence in which channels are added to the DataAcquisition; not to be confused with the device channel ID.

Examples

Add Bidirectional Channels to DataAcquisition

Add bidirectional digital channels to a DataAcquisition, and use indices to view their settings.

```
d = daq("ni");
ch1 = addbidirectional(d,"Dev1","port0/line0","Digital");
[ch2,idx2] = addbidirectional(d,"Dev1","port0/line1","Digital");
d.Channels
```

```
    Index    Type     Device     Channel          Measurement Type         Range         Name
    _____    _____    ______    _____________    ________________________    _____    _________________

      1      "dio"    "Dev1"    "port0/line0"    "Bidirectional (Input)"    "n/a"    "Dev1_port0/line0"
      2      "dio"    "Dev1"    "port0/line1"    "Bidirectional (Input)"    "n/a"    "Dev1_port0/line1"
```

Access one of the channel settings using its index.

```
d.Channels(idx2).ID
```

```
    'port0/line1'
```

Input Arguments

d — DataAcquisition interface
DataAcquisition object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

`deviceID` — Device ID
character vector or string

Device ID specified as a character vector or string, as defined by the device vendor. Obtain the device ID by calling `daqlist`.

Example: `"Dev1"`

Data Types: `char` | `string`

`channelID` — Channel ID
numeric value, character vector, or string

Channel ID specified as a numeric value, character vector, or string; often indicating the physical location of the channel on the device. Supported values are specific to the vendor and device. You can add multiple channels by specifying the channel ID as a numeric vector, or a cell array of character vectors. The *index* returned for this channel in the DataAcquisition display indicates the position of this channel. This channel ID is not the same as channel index in the DataAcquisition: if you add a channel with ID `2` as the first channel in a DataAcquisition, the DataAcquisition channel index is `1`.

Example: `"port1/line1"`

Data Types: `char` | `string` | `numeric` | `cell`

Output Arguments

`ch` — Channel
`Channel` object

Channel, returned as a `DigitalBidirectionalChannel` object with the following properties as described in “Channel Properties” on page 4-7:

```
Device
Direction
ID
MeasurementType
Name
```

`idx` — Channel index
numeric

Channel index returned as a numeric value. With this index, you can access the array of the DataAcquisition `Channels` property.

Version History

Introduced in R2020a

See Also

Functions
`daqlist` | `daq` | `addinput` | `addoutput` | `removechannel`

Properties

"Channel Properties" on page 4-7

addclock

Package: `daq.interfaces`

Add clock connection to device interface

Syntax

```
addclock(d,"ScanClock",clkSrc,clkDest)
clk = addclock( ___ )
[clk,idx] = addclock( ___ )
```

Description

`addclock(d,"ScanClock",clkSrc,clkDest)` adds a clock connection to the DataAcquisition interface for sharing, importing, or exporting a clock configuration. The created clock connection is appended to the `Clocks` property of the DataAcquisition object.

`clk = addclock( ___ )` adds the clock and returns the clock object.

`[clk,idx] = addclock( ___ )` adds the clock and returns the clock object and the clock index from the DataAcquisition interface.

Examples

Add Clocks to DataAcquisition Interface

Add clocks to a DataAcquisition interface in various configurations.

Add a clock shared between two devices.

```
d = daq("ni");
addinput(d,"Dev1","ai0","Voltage")
addinput(d,"Dev2","ai0","Voltage")
addclock(d,"ScanClock","Dev1/PFI0","Dev2/PFI0")
```

Add a clock imported from an external source.

```
d = daq("ni");
addinput(d,"Dev1","ai0","Voltage")
addclock(d,"ScanClock","External","Dev1/PFI0")
```

Add a clock exported to external destination.

```
d = daq("ni");
addinput(d,"Dev1","ai0","Voltage")
addclock(d,"ScanClock","Dev1/PFI0","External")
```

Add a scan clock from an external source supplied at terminal `PXI1Slot5/PXI_Star`.

```
d = daq("ni");
addinput(d,"PXI1Slot5",0,"Voltage")
addclock(d,"ScanClock","External","PXI1Slot5/PXI_Star")
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

`clkSrc` — Clock signal source
string | char

Clock signal source, specified as a string or character vector indicating a device terminal, or `"external"` when importing a clock from an external source.

Example: `"external"`

Data Types: `char` | `string`

`clkDest` — Clock signal destination
string | char

Clock signal destination, specified as a string or character vector indicating a device terminal, or `"external"` when exporting a clock to an external destination.

Example: `"external"`

Data Types: `char` | `string`

Output Arguments

`clk` — Clock
`Clock` object

Clock connection, returned as a `Clock` object with properties `Source`, `Destination`, and `Type`.

`idx` — Clock index
numeric

Clock index, returned as a numeric value. With this index, you can access the array of the DataAcquisition `Clocks` property.

Version History

Introduced in R2020a

See Also

Functions
`daq` | `removeclock`

addinput

Package: `daq.interfaces`

Add input channel to device interface

Syntax

```
addinput(d,deviceID,channelID,measurementType)
ch = addinput( ___ )
[ch,idx] = addinput( ___ )
```

Description

`addinput(d,deviceID,channelID,measurementType)` adds the input channel `channelID` from device `deviceID` to the specified DataAcquisition interface, `d`, configured for the specified measurement type.

The channel information is available from the DataAcquisition `Channels` property.

`ch = addinput( ___ )` adds the channel and returns a channel object.

`[ch,idx] = addinput( ___ )` adds the channel and also returns the channel index from the DataAcquisition interface. The channel index indicates only the sequence in which channels are added to the DataAcquisition; not to be confused with the device channel ID.

Examples

Add Input Channels to DataAcquisition

Add multiple input channels to a DataAcquisition, and use indices to view their settings.

```
d = daq('directsound');
ch1 = addinput(d,"Audio0","1","Audio");
[ch2,idx2] = addinput(d,"Audio1","1","Audio");
d.Channels
```

```
    Index     Type      Device     Channel    Measurement Type        Range            Name
    _____    ______    ________    _______    ________________    ______________    __________

      1      "audi"    "Audio0"      "1"          "Audio"         "-1.0 to +1.0"    "Audio0_1"
      2      "audi"    "Audio1"      "1"          "Audio"         "-1.0 to +1.0"    "Audio1_1"
```

Access one of the channel settings using its index.

```
d.Channels(idx2).Range
```

```
  Range with properties:

    Units: ''
```

```
Max: 1
Min: -1
```

Input Arguments

d — DataAcquisition interface
DataAcquisition object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

`deviceID` — Device ID
character vector or string

Device ID specified as a character vector or string, as defined by the device vendor. Obtain the device ID by calling `daqlist`.

Example: `"Dev1"`

Data Types: `char` | `string`

`channelID` — Channel ID
numeric value, character vector, or string

Channel ID specified as a numeric value, character vector, or string; often indicating the physical location of the channel on the device. Supported values are specific to the vendor and device. You can add multiple channels by specifying the channel ID as a numeric vector, or a cell array of character vectors. The *index* returned for this channel in the DataAcquisition display indicates the position of this channel. This channel ID is not the same as channel index in the DataAcquisition: if you add a channel with ID `2` as the first channel in a DataAcquisition, the DataAcquisition channel index is `1`.

Example: `"ai2"`

Data Types: `char` | `string` | `numeric` | `cell`

`measurementType` — Channel measurement type
character vector | string

Channel measurement type, specified as a character vector or string. `measurementType` represents a vendor-defined measurement type. Valid measurement types include the following:

Measurement Type	Subsystem
`'Voltage'`	Analog Input
`'Current'`	Analog Input
`'Thermocouple'`	Analog Input
`'Accelerometer'`	Analog Input
`'RTD'`	Analog Input
`'Bridge'`	Analog Input
`'Microphone'`	Analog Input
`'IEPE'`	Analog Input
`'Digital'`	Digital I/O

Measurement Type	Subsystem
'EdgeCount'	Counter Input
'Frequency'	Counter Input
'PulseWidth'	Counter Input
'Position'	Counter Input
'Audio'	Audio Input

Not all devices support all types of measurement.

Example: `"Voltage"`

Data Types: `char` | `string`

Output Arguments

`ch` — Channel
`channel` object

Channel, returned as a channel object with properties depending on the measurement type as described in "Channel Properties" on page 4-7.

`idx` — Channel index
numeric

Channel index, returned as a numeric value. With this index, you can access the array of the DataAcquisition `Channels` property.

Version History
Introduced in R2020a

See Also

Functions
`addoutput` | `daq` | `daqlist` | `addbidirectional` | `removechannel`

Properties
"Channel Properties" on page 4-7

addoutput

Package: daq.interfaces

Add output channel to device interface

Syntax

```
addoutput(d,deviceID,channelID,measurementType)
ch = addoutput( ___ )
[ch,idx] = addoutput( ___ )
```

Description

`addoutput(d,deviceID,channelID,measurementType)` adds the output channel `channelID` of device `deviceID` to the specified DataAcquisition interface, `d`, configured for the specified measurement type.

The channel information is available from the DataAcquisition `Channels` property.

`ch = addoutput( ___ )` adds the channel and returns a channel object.

`[ch,idx] = addoutput( ___ )` adds the channel and also returns the channel index from the DataAcquisition interface. The channel index reflects only the sequence in which channels are added to the DataAcquisition; not to be confused with the device channel ID.

Examples

Add Output Channels to DataAcquisition

Add multiple channels to a DataAcquisition, and use indices to view their settings.

```
d = daq('directsound');
ch1 = addoutput(d,"Audio3","1","Audio");
[ch2,idx2] = addoutput(d,"Audio6","1","Audio");
d.Channels
```

```
    Index     Type      Device     Channel    Measurement Type         Range            Name
    _____    ______    ________    _______    ________________    ______________    __________

      1      "audo"    "Audio3"      "1"          "Audio"         "-1.0 to +1.0"    "Audio3_1"
      2      "audo"    "Audio6"      "1"          "Audio"         "-1.0 to +1.0"    "Audio6_1"
```

Access one of the channel settings using its index.

```
d.Channels(idx2).Type
```

```
    'AudioOutputChannel'
```

Input Arguments

d — DataAcquisition interface
DataAcquisition object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

`deviceID` — Device ID
character vector or string

Device ID specified as a character vector or string, as defined by the device vendor. Obtain the device ID by calling `daqlist`.

Example: `"Dev1"`

Data Types: `char` | `string`

`channelID` — Channel ID
numeric value, character vector, or string

Channel ID specified as a numeric value, character vector, or string; often indicating the physical location of the channel on the device. Supported values are specific to the vendor and device. You can add multiple channels by specifying the channel ID as a numeric vector, or a cell array of character vectors. The *index* returned for this channel in the DataAcquisition display indicates the position of this channel. This channel ID is not the same as channel index in the DataAcquisition: if you add a channel with ID `2` as the first channel in a DataAcquisition, the DataAcquisition channel index is `1`.

Example: `"ao2"`

Data Types: `char` | `string` | `numeric` | `cell`

`measurementType` — Channel measurement type
string | character vector

Channel measurement type, specified as a string or character vector. `measurementType` represents a vendor-defined measurement type. Valid measurement types include the following:

Measurement Type	Subsystem
`'Voltage'`	Analog Output
`'Current'`	Analog Output
`'Digital'`	Digital I/O
`'PulseGeneration'`	Counter Output
`'Audio'`	Audio Output
`'Sine'`	Function Generator
`'Square'`	Function Generator
`'Triangle'`	Function Generator
`'RampUp'`	Function Generator
`'RampDown'`	Function Generator
`'DC'`	Function Generator
`'Arbitrary'`	Function Generator

Not all devices support all types of measurement.

Example: `"Voltage"`

Data Types: `char` | `string`

Output Arguments

`ch` — Channel
channel object

Channel, returned as a channel object with properties depending on the measurement type as described in “Channel Properties” on page 4-7.

`idx` — Channel index
numeric

Channel index, returned as a numeric value. With this index, you can access the array of the DataAcquisition `Channels` property.

Version History

Introduced in R2020a

See Also

Functions
`daqlist` | `daq` | `addinput` | `addbidirectional` | `removechannel`

Properties
“Channel Properties” on page 4-7

addtrigger

Package: `daq.interfaces`

Add trigger connection to device interface

Syntax

```
addtrigger(d,"Digital","StartTrigger",trigSrc,trigDest)
trg = addtrigger( ___ )
[trg,idx] = addtrigger( ___ )
```

Description

`addtrigger(d,"Digital","StartTrigger",trigSrc,trigDest)` adds a trigger connection to the DataAcquisition interface. The created connection is appended to the `DigitalTriggers` property of the DataAcquisition object.

`trg = addtrigger( ___ )` adds the trigger and returns the trigger object.

`[trg,idx] = addtrigger( ___ )` adds the trigger and returns the trigger object and the trigger index from the DataAcquisition interface.

Examples

Add Trigger to DataAcquisition Interface

Add triggers to a DataAcquisition interface in various configurations.

Add a trigger shared between two devices.

```
d = daq("ni");
addinput(d,"Dev1","ai0","Voltage")
addinput(d,"Dev2","ai0","Voltage")
addtrigger(d,"Digital","StartTrigger","Dev1/PFI0","Dev2/PFI0")
```

Add a trigger imported from an external source.

```
d = daq("ni");
addinput(d,"Dev1","ai0","Voltage")
addtrigger(d,"Digital","StartTrigger","External","Dev1/PFI0")
```

Add a trigger exported to an external destination.

```
d = daq("ni")
addinput(d,"Dev1","ai0","Voltage")
addtrigger(d,"Digital","StartTrigger","Dev1/PFI0","External")
```

Add a trigger from an external source supplied at terminal `PXI_Trig0`.

```
d = daq("ni");
addinput(d,"PXI1Slot5",0,"Voltage")
addtrigger(d,"Digital","StartTrigger","External","PXI1Slot5/PXI_Trig0")
```

Add a trigger from an external source supplied at terminal `PXI1Slot5/PXI_Star`.

```
d = daq("ni");
addinput(d,"PXI1Slot5",0,"Voltage")
addtrigger(d,"Digital","StartTrigger","External","PXI1Slot5/PXI_Star")
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

`trigSrc` — Trigger signal source
string | char

Trigger signal source, specified as a string or character vector indicating a device terminal, or `"external"` when importing a terminal from an external source.

Example: `"external"`

Data Types: `char` | `string`

`trigDest` — Trigger signal destination
string | char

Trigger signal destination, specified as a string or character vector indicating a device terminal, or `"external"` when exporting a trigger to an external destination.

Example: `"external"`

Data Types: `char` | `string`

Output Arguments

`trg` — Trigger
`Trigger` object

Trigger connection, returned as a trigger object, whose type and properties depend on the kind of trigger. For example:

```
DigitalTrigger with properties:

         Source: 'External'
    Destination: 'Dev4/PFI1'
           Type: StartTrigger
      Condition: 'RisingEdge'
```

`idx` — Trigger index
numeric

Trigger index, returned as a numeric value. With this index, you can access the array of the DataAcquisition `DigitalTriggers` property.

Version History

Introduced in R2020a

See Also

Functions
`daq` | `removetrigger`

Topics
“Synchronize NI PCI Devices Using RTSI” on page 18-125
“Start a Multi-Trigger Acquisition on an External Event” on page 18-128
“Acquire Data from Two Devices at Different Rates” on page 18-136

binaryVectorToDecimal

Convert binary vector value to decimal value

Syntax

```
decVal = binaryVectorToDecimal(binaryVector)
binaryVectorToDecimal(binaryVector,bitOrder)
```

Description

This function is part of Data Acquisition Toolbox, and converts binary data represented by a vector of 1s and 0s. To convert binary data from a string or character vector, you can use the MATLAB function `bin2dec`.

`decVal = binaryVectorToDecimal(binaryVector)` converts a binary vector to a decimal.

`binaryVectorToDecimal(binaryVector,bitOrder)` converts a binary vector with the specified bit orientation to a decimal .

Examples

Convert a Binary Vector to a Decimal Value

```
decVal = binaryVectorToDecimal([1 1 0])
```

```
decVal =

     6
```

Convert a Binary Vector Array to a Decimal Value

```
decVal = binaryVectorToDecimal([1 0 0 0; 0 1 0 0])
```

```
decVal =

     8
     4
```

Convert a Binary Vector with LSB First

```
decVal = binaryVectorToDecimal([1 0 0 0; 0 1 0 0],'LSBFirst')
```

```
decVal =

     1
     2
```

Convert a Binary Vector Array with LSB First

```
decVal = binaryVectorToDecimal([1 1 0],'LSBFirst')
```

```
decVal =

     6
```

Input Arguments

binaryVector — Binary vector to convert to decimal
binary vectors

Binary vector to convert to a decimal, specified as a single binary vector or a row or column-based array of binary vectors.

bitOrder — Bit order for binary vector representation
'MSBFirst' (default) | 'LSBFirst'

Bit order for the binary vector representation, specified as a character vector or string. Accepted values are:

- 'MSBFirst' — The first element of the binary vector is the most significant bit.
- 'LSBFirst' — The first element of the binary vector is the least significant bit.

Data Types: char | string

Output Arguments

decVal — Decimal value
double

Decimal value, returned as a double.

Version History

Introduced in R2012b

See Also

Functions
hexToBinaryVector | decimalToBinaryVector | binaryVectorToHex | bin2dec | dec2bin | hex2dec | dec2hex

Topics
“Generate Digital Output Using Decimal Data Across Multiple Lines” on page 9-14

binaryVectorToHex

Convert binary vector value to hexadecimal

Syntax

```
hexVal = binaryVectorToHex(binaryVector)
hexVal = binaryVectorToHex(binaryVector,bitOrder)
```

Description

This function is part of Data Acquisition Toolbox, and converts binary data represented by a vector of 1s and 0s. To convert binary data from a string, character vector, or literal, you can use the MATLAB functions `bin2dec` and `dec2hex`. See “Hexadecimal and Binary Values”.

`hexVal = binaryVectorToHex(binaryVector)` converts the input binary vector to a hexadecimal.

`hexVal = binaryVectorToHex(binaryVector,bitOrder)` converts the input binary vector using the specified bit orientation.

Examples

Convert a Binary Vector to a Hexadecimal

```
hexVal = binaryVectorToHex([0 0 1 1 1 1 0 1])
```

```
hexVal =

    '3D'
```

Convert an Array of Binary Vectors to Hexadecimal

```
hexVal = binaryVectorToHex([1 1 0 0 0 1 0 0 ; 0 0 0 0 1 0 1 0])
```

```
hexVal =

  2×1 cell array

    {'C4'}
    {'0A'}
```

The output is appended with 0s to make all hex values the same length character vectors.

Convert a Binary Vector with LSB First

```
hexVal = binaryVectorToHex([0 0 1 1 1 1 0 1],'LSBFirst')
```

```
hexVal =

    'BC'
```

Convert a Binary Vector Array with LSB First

```
hexVal = binaryVectorToHex([1 1 0 0 0 1 0 0 ; 0 0 0 0 1 0 1 0],'LSBFirst')
```

```
hexVal =

  2×1 cell array

    {'23'}
    {'50'}
```

If necessary, the output is appended with 0s to make all hex values the same length character vectors.

Note The binary vector array is converted to a cell array of hexadecimal numbers. If you input a single binary vector, it is converted to a hexadecimal character vector.

Input Arguments

`binaryVector` — Binary vector to convert to hexadecimal

numeric vector of 1s and 0s

Binary vector to convert to hexadecimal, specified as a numeric vector with 0s and 1s. The vector can be a column or row vector.

`bitOrder` — Bit order for binary vector representation

`'MSBFirst'` (default) | `'LSBFirst'`

Bit order for the binary vector representation, specified as a character vector or string. Accepted values are:

- `'MSBFirst'` — The first element of the binary vector is the most significant bit.
- `'LSBFirst'` — The first element of the binary vector is the least significant bit.

Data Types: `char` | `string`

Output Arguments

`hexVal` — Hexadecimal value

character vector

Hexadecimal value returned as a character vector. Multiple values are returned as a cell array of character vectors.

Version History

Introduced in R2012b

See Also

Functions

`hexToBinaryVector` | `binaryVectorToDecimal` | `decimalToBinaryVector` | `bin2dec` | `dec2bin` | `hex2dec` | `dec2hex`

Topics

"Acquire Digital Data in Hexadecimal Values" on page 9-12

daq

Package: `daq.interfaces`

Create DataAcquisition device interface for specific vendor

Syntax

```
d = daq(vendor)
```

Description

`d = daq(vendor)` creates a `DataAcquisition` interface object for configuring and operating data acquisition devices from the specified vendor.

Examples

Create a DataAcquisition

Create a `DataAcquisition` object for interfacing with Windows sound devices.

```
d = daq("directsound")
```

```
d =

DataAcquisition using DirectSound hardware:

                     Running: 0
                        Rate: 44100
           NumScansAvailable: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Input Arguments

`vendor` — Device vendor
`"ni"` | `"adi"` | `"mcc"` | `"directsound"` | `"digilent"`

Device vendor specified as a string or character vector.

Example: `"ni"`

Data Types: `char` | `string`

Output Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, returned as a `DataAcquisition` object. This interface can accommodate all supported devices from the specified vendor. Interfaces with different vendors require separate `DataAcquisition` objects.

Version History

Introduced in R2020a

See Also

Functions
`daqvendorlist` | `daqlist` | `addinput` | `addoutput` | `addbidirectional` | `removechannel` | `addclock` | `removeclock` | `addtrigger` | `removetrigger`

Objects
`DataAcquisition`

daqhelp

Help for toolbox interface

Syntax

```
daqhelp
daqhelp(functionname)
helptext = daqhelp('functionname')
```

Description

`daqhelp` displays a comprehensive listing of Data Acquisition Toolbox functions along with a brief description of each. Links in the output provide access to more detailed information.

`daqhelp(functionname)` returns help for the function specified as a character vector or string.

`helptext = daqhelp('functionname')` assigns the help text output to the variable `out`.

Examples

Get Toolbox Help

Get overview help for Data Acquisition Toolbox.

```
daqhelp
```

Get Function Help

Get help for a specified function.

```
daqhelp("addinput")
```

Return Function Help Text to Variable

Get help for a specified function, assigning the help text to a variable.

```
helptext = daqhelp("addinput");
```

Input Arguments

`functionname` — Function for which you want help
char vector or string

Function for which you want help, specified as a character vector or string.

Example: `"addinput"`

Data Types: `char` | `string`

Output Arguments

`helptext` — Help text
char vector

Help text, returned as a character vector.

Version History

Introduced before R2006a

daqlist

List data acquisition devices available to toolbox

Syntax

```
daqlist
daqlist(vendor)
dev = daqlist( ___ )
```

Description

`daqlist` displays a table of all available devices for all supported vendors. The information for each device includes device IDs, descriptions, models, and device subsystems.

`daqlist(vendor)` lists all available devices for the specified vendor in table format.

`dev = daqlist( ___ )` assigns the device table to `dev`. You can access individual table cells by indexing position or column labels.

Examples

List Devices for All Vendors

List all available devices.

```
dev = daqlist
```

```
dev =

  12×5 table

    VendorID        DeviceID                          Description                                            Model
    _____________    ___________    __________________________________________________________    ______________________________________

    "ni"            "Dev2"         "National Instruments(TM) PCIe-6363"                          "PCIe-6363"
    "ni"            "PXI1Slot2"    "National Instruments(TM) PXI-4461"                           "PXI-4461"
    "adi"           "SMU1"         "Analog Devices Inc. ADALM1000"                               "ADALM1000"
    "directsound"   "Audio0"       "DirectSound Primary Sound Capture Driver"                    "Primary Sound Capture Driver"
    "directsound"   "Audio1"       "DirectSound Headset Microphone (Plantronics BT600)"          "Headset Microphone (Plantronics BT600
    "directsound"   "Audio2"       "DirectSound Primary Sound Driver"                            "Primary Sound Driver"
    "directsound"   "Audio3"       "DirectSound Headset Earphone (Plantronics BT600)"            "Headset Earphone (Plantronics BT600)"
    "directsound"   "Audio4"       "DirectSound LEN T2454pA (NVIDIA High Definition Audio):1"    "LEN T2454pA (NVIDIA High Definition A
    "directsound"   "Audio5"       "DirectSound LEN T2454pA (NVIDIA High Definition Audio):2"    "LEN T2454pA (NVIDIA High Definition A
    "directsound"   "Audio6"       "DirectSound Speakers (Lenovo USB Soundbar)"                  "Speakers (Lenovo USB Soundbar)"
    "directsound"   "Audio7"       "DirectSound Speakers (Realtek High Definition Audio)"        "Speakers (Realtek High Definition Aud
    "mcc"           "Board0"       "Measurement Computing Corp. USB-1208FS-Plus"                 "USB-1208FS-Plus"
```

List Devices for Specific Vendor

List all available National Instruments devices.

```
dev = daqlist("ni")
```

```
dev =
```

```
  12×5 table

    VendorID      DeviceID                Description                   Model         DeviceInfo
    ________    ___________    __________________________________    ___________    ____________________

    "ni"        "Dev2"         "National Instruments(TM) PCIe-6363"    "PCIe-6363"   [1×1 daq.DeviceInfo]
    "ni"        "PXI1Slot2"    "National Instruments(TM) PXI-4461"     "PXI-4461"    [1×1 daq.DeviceInfo]
```

View details of the first device.

```
devinfo = dev.DeviceInfo(1)
```

```
devinfo =

ni: National Instruments(TM) PCIe-6363 (Device ID: 'Dev2')
   Analog input supports:
      7 ranges supported
      Rates from 0.1 to 2000000.0 scans/sec
      32 channels ('ai0' - 'ai31')
      'Voltage' measurement type

   Analog output supports:
      -5.0 to +5.0 Volts,-10 to +10 Volts ranges
      Rates from 0.1 to 2857142.9 scans/sec
      4 channels ('ao0','ao1','ao2','ao3')
      'Voltage' measurement type

   Digital IO supports:
      Rates from 0.1 to 10000000.0 scans/sec
      48 channels ('port0/line0' - 'port2/line7')
      'InputOnly','OutputOnly','Bidirectional' measurement types

   Counter input supports:
      Rates from 0.1 to 100000000.0 scans/sec
      4 channels ('ctr0','ctr1','ctr2','ctr3')
      'EdgeCount','PulseWidth','Frequency','Position' measurement types

   Counter output supports:
      Rates from 0.1 to 100000000.0 scans/sec
      4 channels ('ctr0','ctr1','ctr2','ctr3')
      'PulseGeneration' measurement type
```

Input Arguments

vendor — Device vendor

"ni" | "adi" | "mcc" | "directsound" | "digilent"

Device vendor specified as a string or character vector.

Example: "ni"

Data Types: char | string

Output Arguments

dev — Table of devices

table

List of available devices, returned as a table.

Version History

Introduced in R2020a

See Also

Functions
`daqvendorlist` | `daq`

daqreset

Reset Data Acquisition Toolbox

Syntax

```
daqreset
```

Description

`daqreset` resets Data Acquisition Toolbox and deletes all data acquisition objects.

Examples

Reset the Toolbox

Create a DataAcquisition interface, then reset the toolbox.

```
d = daq("ni");
daqreset
d
```

```
d = 

  handle to deleted DataAcquisition
```

Version History

Introduced before R2006a

Functions

daqvendorlist

List vendors available to toolbox

Syntax

```
daqvendorlist
v = daqvendorlist
```

Description

`daqvendorlist` displays a list of supported vendors with information about adaptor versions, driver versions, and operational status. Vendor support requires installation of the appropriate support package. See “Data Acquisition Toolbox Supported Hardware”.

`v = daqvendorlist` assigns the table to `v`.

Examples

List Available Vendors

List vendors available to toolbox.

```
daqvendorlist
```

```
ans =

  5×5 table

         ID                   FullName              AdaptorVersion    DriverVersion    Operational
    _____________    __________________________    ______________    _____________    ___________

    "ni"             "National Instruments(TM)"    "4.0 (R2019b)"      "unknown"         false
    "adi"            "Analog Devices Inc."         "4.0 (R2019b)"      "1.0"             true
    "directsound"    "DirectSound"                 "4.0 (R2019b)"      "n/a"             true
    "digilent"       "Digilent Inc."               "4.0 (R2019b)"      "3.7.20"          true
    "mcc"            "Uninitialized"               "4.0 (R2019b)"      "unknown"         false
```

Output Arguments

v — Vendor information

table

Vendor information returned as a table.

Version History

Introduced in R2020a

See Also

Functions
`daqlist` | `daq`

DataAcquisition

Interface to data acquisition device

Description

The `DataAcquisition` object provides access to the devices of a specified vendor.

Creation

Use the `daq` function to create a `DataAcquisition` object.

Properties

`AutoSyncDSA` — Automatically Synchronize DSA devices
false (default) | true

Automatically Synchronize DSA devices, specified as a logical `true` or `false`. Use this property to enable or disable automatic synchronization between DSA (PXI or PCI) devices in the same DataAcquisition. By default automatic synchronization capability is disabled.

Example: `true`

Data Types: `logical`

`Channels` — Device channels
array of channel objects

This property is read-only.

Device channels, returned as an array of channel objects. Create channels with the functions `addinput`, `addoutput`, and `addbidirectional`.

Example: `addinput(d,…)`

`Clocks` — Device clock connections
array of `clock` objects

This property is read-only.

Device clock connections, returned as an array of clock objects. Create clocks with the `addclock` function.

Example: `addclock(d,…)`

`DigitalTriggers` — Device digital trigger connections
array of `DigitalTrigger` objects

This property is read-only.

Device digital trigger connections, returned as an array of `DigitalTrigger` objects. Use the `addtigger` function to add digital triggers to the DataAcquisition.

Example: `addtrigger(d,…)`

`DigitalTriggerTimeout` — Time allowed for occurrence of digital trigger
10 (default) | numeric | duration

Time allowed for occurrence of digital trigger, specified as a numeric value in seconds or a duration.

Example: 30

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`ErrorOccurredFcn` — Callback function to call when error occurs
function handle

Callback function to call when error occurs, specified as a function handle.

Example: `@mycleanup`

Data Types: `function_handle`

`NumDigitalTriggersRemaining` — Number of digital triggers remaining in run
1 (default) | numeric

This property is read-only.

Number of digital triggers remaining in run, returned as a double.

Example: 1

Data Types: `double`

`NumDigitalTriggersPerRun` — Number of digital triggers per DataAcquisition run
numeric

Number of digital triggers per DataAcquisition run, returned as a double.

Example: 2

Data Types: `double`

`NumScansAcquired` — Number of data scans acquired since the last start
numeric

This property is read-only.

Number of data scans acquired in background operation since the last `start`, returned as a uint64 value. This value is reset each time `start` is called, and does not reflect whether the scans have been read into MATLAB.

Example: 1000

Data Types: `uint64`

`NumScansAvailable` — Number of data scans acquired and available for reading
numeric

This property is read-only.

Number of data scans available for reading, returned as a uint64 value. These scans have been acquired by the device input channels in a background operation, but have not yet been read into MATLAB. The value decreases with each call to `read`; and is reset by a call to `start`.

Example: 1000

Data Types: `uint64`

`NumScansOutputByHardware` — Number of scans generated as device output
numeric

This property is read-only.

Number of scans generated as device output, returned as a uint64 value.

Example: 1024

Data Types: `uint64`

`NumScansQueued` — Number of scans prepared for device output
numeric

This property is read-only.

Number of scans queued to the device output channels, returned as a uint64 value.

Example: 4000

Data Types: `uint64`

`Rate` — Data scan rate
numeric

Data scan rate, specified as a numeric value of samples per second.

Example: 44100

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`RateLimit` — Lower and upper scan rate limits
array of doubles

This property is read-only.

Lower and upper scan rate limits, returned as a 1-by-2 vector of doubles indicating minimum and maximum allowed scan rates in samples per second. The scan rate limits depend on the hardware and its configurations. In devices that multiplex channels to a converter, the rate limit is impacted by the number of channels you use. For more information, see “Sampling” on page 1-13.

Example: `[8000 192000]`

Data Types: `double`

`Running` — DataAcquisition running indication
true | false

This property is read-only.

DataAcquisition running indication, returned as true or false.

Example: `true`

Data Types: `logical`

`ScansAvailableFcn` — Callback function when scans are available
function handle

Callback function to execute when scans are available from the input channels, specified as a function handle

Example: `@read`

Data Types: `function_handle`

`ScansAvailableFcnCount` — Number of acquired scans to trigger `ScansAvailableFcn`
numeric

Number of acquired scans to trigger `ScansAvailableFcn`, specified as a numeric value. The function handle specified in `ScansAvailableFcn` executes every time `ScansAvailableFcnCount` scans are acquired from the input channels.

Example: 8000

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ScansRequiredFcn` — Callback function when output scan data is required
function handle

Callback function to execute when scan data is required for device output channels, specified as a function handle.

Example: `@write`

Data Types: `function_handle`

`ScansRequiredFcnCount` — Number of scans to trigger `ScansRequiredFcn`
`"auto"` (default) | numeric

Number of queued scans to trigger `ScansRequiredFcn`, specified as a numeric value or `"auto"`. The function handle specified in `ScansRequiredFcn` executes when `NumScansQueued` drops below the value specified in this property. If this is set to `"auto"`, the value resets to a default.

Example: 2000

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`UserData` — Custom data
any data

Custom data, specified as any MATLAB data type and format.

Example: `datetime('now')`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`

`Vendor` — Data acquisition hardware vendor information
vendor object

This property is read-only.

Data acquisition hardware vendor information, returned as a vendor object with the following properties:

```
ID
FullName
AdaptorVersion
DriverVersion
IsOperational
```

This object is the same as the corresponding vendor object returned by the `daqvendorlist` function.

`WaitingForDigitalTrigger` — Digital trigger waiting indication
false (default) | true

This property is read-only.

Digital trigger waiting indication, returned as a logical.

Example: `true`

Data Types: `logical`

Object Functions

addinput	Add input channel to device interface
read	Read data acquired by hardware
readwrite	Simultaneously read and write device channel data
start	Start DataAcquisition background operation
stop	Stop background operation
removechannel	Remove channel from device interface
flush	Flush DataAcquisition input and output buffers
write	Write output scans to hardware channels
preload	Queue scan data for device output
addoutput	Add output channel to device interface
addbidirectional	Add digital bidirectional channel to device interface
resetcounters	Reset hardware scan count for all counter inputs
addclock	Add clock connection to device interface
removeclock	Remove clock from device interface
addtrigger	Add trigger connection to device interface
removetrigger	Remove trigger from device interface

Examples

Create a DataAcquisition

Create and configure a `DataAcquisition` object for interfacing with National Instruments devices.

```
d = daq("ni")
d.Rate = 20000;
```

Version History

Introduced in R2020a

See Also

Functions

daqlist | daqvendorlist | daqreset | daq | daqhelp

decimalToBinaryVector

Convert decimal value to binary vector

Syntax

```
binVal = decimalToBinaryVector(decimalNumber)
binVal = decimalToBinaryVector(decimalNumber,numberOfBits)
binVal = decimalToBinaryVector(decimalNumber,numberOfBits,bitOrder)
binVal = decimalToBinaryVector(decimalNumber,[],bitOrder)
```

Description

This function is part of Data Acquisition Toolbox, and converts decimal values to binary data represented by a vector of 1s and 0s. To convert to binary data as a character vector, you can use the MATLAB function `dec2bin`.

`binVal = decimalToBinaryVector(decimalNumber)` converts a positive decimal number to a binary vector, represented using the minimum number of bits necessary.

`binVal = decimalToBinaryVector(decimalNumber,numberOfBits)` converts a decimal number to a binary vector with the specified number of bits.

`binVal = decimalToBinaryVector(decimalNumber,numberOfBits,bitOrder)` converts a decimal number to a binary vector with the specified number of bits in the specified bit ordering.

`binVal = decimalToBinaryVector(decimalNumber,[],bitOrder)` converts a decimal number to a binary vector with default number of bits in the specified bit ordering.

Examples

Convert a Decimal to a Binary Vector

```
binVal = decimalToBinaryVector(6)
```

```
binVal =

     1     1     0
```

Convert an Array of Decimals to a Binary Vector Array

```
binVal = decimalToBinaryVector(0:4)
```

```
binVal =

     0     0     0
     0     0     1
     0     1     0
```

```
     0     1     1
     1     0     0
```

Convert a Decimal into a Binary Vector of Specific Bits

```
binVal = decimalToBinaryVector(6,8,'MSBFirst')
```

```
binVal =

     0     0     0     0     0     1     1     0
```

Convert a Decimal into a Binary Vector with LSB First

```
binVal = decimalToBinaryVector(6,[],'LSBFirst')
```

```
binVal =

     0     1     1
```

Convert an Array of Decimals into a Binary Vector Array with LSB First

```
binVal = decimalToBinaryVector(0:4, 4,'LSBFirst')
```

```
binVal =

     0     0     0     0
     1     0     0     0
     0     1     0     0
     1     1     0     0
     0     0     1     0
```

Input Arguments

`decimalNumber` — Number to convert to binary vector
numeric

The number to convert to a binary vector specified as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`numberOfBits` — Number of bits required to correctly represent the decimal number
numeric

The number of bits required to correctly represent the decimal. This is an optional argument. If you do not specify the number of bits, the number is represented using the minimum number of bits needed. By default minimum number of bits needed to represent the value is specified, unless you specify a value

`bitOrder` — Bit order for binary vector representation
`'MSBFirst'` (default) | `'LSBFirst'`

Bit order for the binary vector representation, specified as a character vector or string. Accepted values are:

- `'MSBFirst'` — The first element of the binary vector is the most significant bit.
- `'LSBFirst'` — The first element of the binary vector is the least significant bit.

Data Types: `char` | `string`

Output Arguments

`binVal` — Binary value
array of 1s and 0s

Binary value, returned as a double array of 1s and 0s.

Version History

Introduced in R2012b

See Also

Functions
`hexToBinaryVector` | `binaryVectorToDecimal` | `binaryVectorToHex` | `bin2dec` | `dec2bin` | `hex2dec` | `dec2hex`

Topics
“Generate Digital Output Using Decimal Data Across Multiple Lines” on page 9-14

disableVendorDiagnostics

Suppress vendor diagnostic display in device listing

Syntax

```
disableVendorDiagnostics
```

Description

`disableVendorDiagnostics` turns off the display of diagnostic information in the `daqlist` function output related to non-operational vendors. The display is enabled by default.

Examples

Toggle Diagnostic Display

Control the display of diagnostic information in the device listing.

Allow diagnostic information to display in the device listing. The installation does not include drivers for `'ni'` or `'mcc'`.

```
enableVendorDiagnostics
daqlist
```

```
Unable to detect 'ni' hardware:
National Instruments NI-DAQmx driver is either not installed or the installed version is not supp
Use the Windows Control Panel to uninstall any existing NI-DAQmx driver listed under 'National In
Then, open the Add-On Explorer to install the Data Acquisition Toolbox Support Package for
National Instruments NI-DAQmx Devices.

Unable to detect 'mcc' hardware:
Driver command failed with status code: -30.

ans =

  1×5 table

      VendorID       DeviceID                           Description
    _____________    ________    __________________________________________________________

    "directsound"    "Audio0"    "DirectSound Primary Sound Capture Driver"
```

Suppress diagnostic information in the device listing. The installation is the same.

```
disableVendorDiagnostics
daqlist
```

```
ans =

  1×5 table

      VendorID       DeviceID                           Description
    _____________    ________    __________________________________________________________
```

```
    "directsound"    "Audio0"    "DirectSound Primary Sound Capture Driver"
```

Version History

Introduced in R2020a

See Also

Functions

daqlist | daqvendorlist | enableVendorDiagnostics

enableVendorDiagnostics

Allow diagnostic display in vendor listing

Syntax

```
enableVendorDiagnostics
```

Description

`enableVendorDiagnostics` turns on the display of diagnostic information in the `daqlist` function output related to non-operational vendors. The display is enabled by default.

Examples

Toggle Diagnostic Display

Control the display of diagnostic information in the device listing.

Allow diagnostic information to display in the device listing. The installation does not include drivers for `'ni'` or `'mcc'`.

```
enableVendorDiagnostics
daqlist
```

```
Unable to detect 'ni' hardware:
National Instruments NI-DAQmx driver is either not installed or the installed version is not supported.
Use the Windows Control Panel to uninstall any existing NI-DAQmx driver listed under 'National Instruments Software'.
Then, open the Add-On Explorer to install the Data Acquisition Toolbox Support Package for
National Instruments NI-DAQmx Devices.

Unable to detect 'mcc' hardware:
Driver command failed with status code: -30.

ans =

  1×5 table

      VendorID       DeviceID                          Description
    _____________    ________    ___________________________________________________________

    "directsound"    "Audio0"    "DirectSound Primary Sound Capture Driver"
```

Suppress diagnostic information in the device listing. The installation is the same.

```
disableVendorDiagnostics
daqlist
```

```
ans =

  1×5 table

      VendorID       DeviceID                          Description
    _____________    ________    ___________________________________________________________
```

```
    "directsound"    "Audio0"    "DirectSound Primary Sound Capture Driver"
```

Version History

Introduced in R2020a

See Also

Functions

daqlist | daqvendorlist | disableVendorDiagnostics

flush

Package: `daq.interfaces`

Flush DataAcquisition input and output buffers

Syntax

`flush(d)`

Description

`flush(d)` removes all acquired and queued scans in the input and output buffers of the DataAcquisition interface.

Examples

Flush DataAcquisition Data

Clear all acquired and queued scans.

```
d = daq("ni")
% :
flush(d)
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

Version History

Introduced in R2020a

See Also

Functions
`daq`

hasdata

Package: `matlab.io.datastore`

Determine if data is available to read from TDMS datastore

Syntax

`tf = hasdata(tdmsds)`

Description

`tf = hasdata(tdmsds)` returns logical `1` (`true`) if there is data available to read from the TDMS datastore specified by `tdmsds`. Otherwise, it returns logical `0` (`false`).

Examples

Check TDMS Datastore for Readable Data

In a `while`-loop, use `hasdata` to determine if there is any more data to read at the current location.

```
tdmsds = tdmsDatastore(tdmsDatastore("C:\data\tdms"));
while hasdata(tdmsds)
    m = read(tdmsds);
        ⋮
end
```

Input Arguments

`tdmsds` — TDMS datastore
`TDMSDatastore` object

TDMS datastore, specified as a `TDMSDatastore` object.

Example: `tdmsds = tdmsDatastore("C:\data\tdms")`

Output Arguments

`tf` — Indicator of data to read
`1` | `0`

Indicator of data to read, returned as a logical `1` (`true`) or `0` (`false`).

Version History

Introduced in R2022a

See Also

Functions
`tdmsDatastore` | `read` | `reset`

hexToBinaryVector

Convert hexadecimal value to binary vector

Syntax

```
binVal = hexToBinaryVector(hexNumber)
binVal = hexToBinaryVector(hexNumber,numberOfBits)
binVal = hexToBinaryVector(hexNumber,numberOfBits,bitOrder)
```

Description

This function is part of Data Acquisition Toolbox, and converts hexadecimal data to binary data represented by a vector of 1s and 0s. To convert to binary data as a character vector, you can use the MATLAB functions `hex2dec` and `dec2bin`. See "Hexadecimal and Binary Values".

`binVal = hexToBinaryVector(hexNumber)` converts hexadecimal numbers to a binary vector.

`binVal = hexToBinaryVector(hexNumber,numberOfBits)` converts hexadecimal numbers to a binary vector with the specified number of bits.

`binVal = hexToBinaryVector(hexNumber,numberOfBits,bitOrder)` converts hexadecimal numbers to a binary vector with the specified number of bits in the specified bit ordering.

Examples

Convert a hexadecimal to a binary vector

```
binVal = hexToBinaryVector('A1')
```

```
binVal =

  1×8 logical array

   1   0   1   0   0   0   0   1
```

Convert a hexadecimal with a leading 0 to a binary Vector

```
binVal = hexToBinaryVector('0xA')
```

```
binVal =

  1×4 logical array

   1   0   1   0
```

Convert an Array of Hexadecimal Numbers to a Binary Vector

```
binVal = hexToBinaryVector(['A1';'B1'])
```

```
binVal =

  2×8 logical array

   1   0   1   0   0   0   0   1
   1   0   1   1   0   0   0   1
```

Convert a Hexadecimal Number into a Binary Vector of Specific Bits

```
binVal = hexToBinaryVector('A1',12,'MSBFirst')
```

```
binVal =

  1×12 logical array

   0   0   0   0   1   0   1   0   0   0   0   1
```

Convert a Cell Array of Hexadecimal Numbers into a Binary Vector of Specific Bits

```
binVal = hexToBinaryVector({'A1';'B1'},8)
```

```
binVal =

  2×8 logical array

   1   0   1   0   0   0   0   1
   1   0   1   1   0   0   0   1
```

Convert a Hexadecimal into a Binary Vector with LSB First

```
binVal = hexToBinaryVector('A1', [], 'LSBFirst')
```

```
binVal =

  1×8 logical array

   1   0   0   0   0   1   0   1
```

Input Arguments

`hexNumber` — Hexadecimal to convert to binary vector
hexadecimal value

Hexadecimal number to convert to a binary vector, specified as a character vector or string.

Data Types: `char` | `string`

`numberOfBits` — Number of bits to represent the decimal number
numeric

Number of bits to represent the decimal number, specified as a numeric value. This is an optional argument. If you do not specify the number of bits, the number is represented using the minimum number of bits needed.

`bitOrder` — Bit order for binary vector representation
`'MSBFirst'` (default) | `'LSBFirst'`

Bit order for the binary vector representation, specified as a character vector or string. Accepted values are:

- `'MSBFirst'` — The first element of the binary vector is the most significant bit.
- `'LSBFirst'` — The first element of the binary vector is the least significant bit.

Data Types: `char` | `string`

Output Arguments

`binVal` — Binary value
array of 1s and 0s

Binary value, returned as a logical array of 1s and 0s.

Version History

Introduced in R2012b

See Also

Functions
`decimalToBinaryVector` | `binaryVectorToDecimal` | `binaryVectorToHex` | `bin2dec` | `dec2bin` | `hex2dec` | `dec2hex`

Topics
"Acquire Digital Data in Hexadecimal Values" on page 9-12

preload

Package: `daq.interfaces`

Queue scan data for device output

Syntax

```
preload(d,scanData)
```

Description

`preload(d,scanData)` provides scan data to the DataAcquisition interface `d` for device output.

You queue data before calling `start` on your DataAcquisition. Calling `start` runs the DataAcquisition in the background, without blocking MATLAB.

Examples

Queue Scan Data for Device Output

Queue scan data to the DataAcquisition interface in preparation for device output.

Define and queue a sine wave for output of one cycle on a single channel.

```
scanData = sin(linspace(0,2*pi,5000)');
preload(d,scanData)
% ⋮
start(d)
```

Define and queue a sine wave for repeated output on a single channel.

```
scanData = sin(linspace(0,2*pi,5000)');
preload(d,scanData)
% ⋮
start(d,"RepeatOutput")
% ⋮
stop(d)
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq(...)`

`scanData` — Scan data for device output
double matrix

Scan data for device output, specified as an M-by-N matrix, where M is the number of data scans and N is the number of output channels in the DataAcquisition interface. For a single channel, the data is a column vector.

Data Types: `double`

Version History

Introduced in R2020a

See Also

Functions

`daq` | `start` | `flush`

preview

Package: `matlab.io.datastore`

Read first 8 records from TDMS datastore

Syntax

`data = preview(tdmsds)`

Description

`data = preview(tdmsds)` returns the first 8 records from TDMS datastore `tdmsds`, without changing the current read position in the datastore.

Examples

Examine Preview of TDMS Datastore

```
tdmsds = tdmsds = tdmsDatastore("C:\data\tdms");
data = preview(tdmsds)
```

```
data =

  1×3 cell array

    {8×2 table}    {8×2 table}    {8×2 table}
```

```
data{3}
```

```
ans =

  8×2 table

    Torque1     Torque2
    ________    ________

        0.15    -0.55729
    -0.18286         0.1
    -0.18286    -0.55729
    -0.18286    -0.88593
    -0.18286         0.1
        0.15    -0.22864
    -0.51572    -0.88593
        0.15         0.1
```

Input Arguments

`tdmsds` — TDMS datastore
`TDMSDatastore` object

TDMS datastore, specified as a `TDMSDatastore` object.

Example: `tdmsds = tdmsDatastore("C:\data\tdms")`

Output Arguments

`data` — Start of TDMS datastore records
cell array of tables

Start of TDMS datastore records, returned as a cell array of tables from the TDMS data.

Version History

Introduced in R2022a

See Also

Functions
`tdmsDatastore` | `read` | `readall` | `hasdata`

read

Package: `daq.interfaces`

Read data acquired by hardware

Syntax

```
scanData = read(d)
scanData = read(d,span)
[scanData,triggerTime] = read( ___ )
scanData = read( ___ ,"OutputFormat","Matrix")
[scanData,timeStamp,triggerTime] = read( ___ ,"OutputFormat","Matrix")
```

Description

`scanData = read(d)` reads a single input scan from all input channels on the DataAcquisition, and returns a timetable to `scanData`.

`scanData = read(d,span)` reads a span of input scans from the DataAcquisition interface, and returns a timetable to `scanData`. You can specify `span` as a duration, a number of scans, or `"all"`.

- If the DataAcquisition is not running and has no acquired data, the DataAcquisition starts a foreground finite acquisition to read the requested number of scans. MATLAB is blocked until the acquisition and read are complete.
- If the DataAcquisition is running when you call this function, it reads data already acquired, if necessary waiting until the specified number of scans are available. MATLAB is blocked until the acquisition and read are complete. This is typical when `start` is called to run a background acquisition prior to calling `read`.
- If the DataAcquisition is not running but has acquired data from a previous run, it reads the specified number of scans or all the data, whichever is less.

`[scanData,triggerTime] = read( ___ )` performs the specified read, and returns a timetable to `scanData` and scan trigger time to `triggerTime` as a datetime.

`scanData = read( ___ ,"OutputFormat","Matrix")` performs the specified read, and returns an M-by-N matrix of doubles to `scanData`, where M is the number of scans and N is the number of input channels. Each column contains the data from one channel.

`[scanData,timeStamp,triggerTime] = read( ___ ,"OutputFormat","Matrix")` performs the specified read and returns the scan timestamps to `timeStamp`, as an M-by-1 vector of doubles representing the relative time in seconds after the first scan. The rows of the `timeStamp` vector correspond to the rows of the `scanData` matrix. The scan trigger time is returned to `triggerTime` as a datenum double.

Examples

Read a Single Scan

Without specifying a duration or number of scans, the `read` function acquires a single on-demand scan on all channels.

```
d = daq("ni")
addinput(d,"Dev1",1,"Voltage"); % add more channels as needed
scanData = read(d)
```

```
data =

  timetable

    Time      Dev1_ai1
    _____     ________

    0 sec     -1.9525
```

Initiate a Foreground Acquisition

If there is no data available to be read from the device, the `read` function initiates a foreground acquisition, blocking MATLAB until complete.

```
d = daq("ni");
ch = addinput(d,"Dev1",1:2,"Voltage")
```

```
ch =

    Index    Type    Device    Channel    Measurement Type          Range               Name
    _____    ____    ______    _______    ________________    __________________    __________

      1      "ai"    "Dev1"     "ai1"     "Voltage (Diff)"    "-10 to +10 Volts"    "Dev1_ai1"
      2      "ai"    "Dev1"     "ai2"     "Voltage (Diff)"    "-10 to +10 Volts"    "Dev1_ai2"
```

Read five scans of data on all channels.

```
scanData = read(d,5)
```

```
scanData =

  5×2 timetable

      Time        Dev1_ai1    Dev1_ai2
    _________     ________    ________

    0 sec          0.1621     0.62579
    0.001 sec     0.42124     0.56955
    0.002 sec     0.51069     0.56002
    0.003 sec     0.54193     0.56166
    0.004 sec     0.55377     0.56396
```

Read 5 milliseconds of data on all channels.

```
d.Rate = 1000;
scanData = read(d,seconds(0.005))
```

```
scanData =

  5×2 timetable
```

```
     Time        Dev1_ai1    Dev1_ai2
    _________    ________    ________

    0 sec         0.2259     0.33278
    0.001 sec    0.28871     0.31699
    0.002 sec     0.3068     0.31633
    0.003 sec     0.3137     0.31929
    0.004 sec    0.31732     0.32028
```

You can also read the data into arrays of double values. Five scans on two channels results in a 5-by-2 matrix, with a column for each channel.

```
scanData = read(d,5,"OutputFormat","Matrix")
```

```
scanData =

    0.0424    0.0644
    0.0572    0.0621
    0.0605    0.0638
    0.0618    0.0641
    0.0631    0.0648
```

Read Data from a Background Acquisition

When a background acquisition is initiated with the `start` function, use `read` to import the data.

```
d = daq("ni");
ch = addinput(d,"Dev1",1:2,"Voltage")
start(d,"NumScans",5)
```

```
Background operation has started.
Background operation will stop after 0.005 s.
To read acquired scans, use read.
```

```
scanData = read(d,"all")
```

```
scanData =

  5×2 timetable

     Time        Dev1_ai1    Dev1_ai2
    _________    ________    ________

    0 sec        0.012466    0.023977
    0.001 sec    0.019373    0.023319
    0.002 sec    0.021017     0.02299
    0.003 sec    0.021346     0.02299
    0.004 sec    0.022661    0.023648
```

Input Arguments

d — DataAcquisition interface

`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

span — Length of read operation
duration | double

Length of read operation, specified as a duration or double. If this is a duration type, it specifies the time duration of acquisition; if a double, it specifies the number of scans.

Example: `seconds(5)`

Data Types: `double` | `duration`

Output Arguments

`scanData` — Input scan data from the device
timetable | double

Input scan data from the device, returned as a timetable or matrix of doubles, depending on the `OutputFormat` setting.

The time stamp for each scan in the timetable is a duration, relative to the trigger time. You can access the scan trigger time in the timetable property `scanData.Properties.CustomProperties.TriggerTime`, returned as a datetime.

`triggerTime` — Time that acquisition began
datetime | datenum double

Time that acquisition began, returned as a datetime if `OutputFormat` is `'Timetable'` (default), or as a double if `OutputFormat` is `'Matrix'`. This information is also available as a datetime value in the timetable property `scanData.Properties.CustomProperties.TriggerTime`.

`timeStamp` — Times of scan acquisitions
double

Times of scan acquisitions, returned as a matrix of doubles. Each value represents relative time in seconds after the first scan. This argument is returned only when `OutputFormat` is specified as `"Matrix"`.

Version History

Introduced in R2020a

See Also

Functions
`start`

read

Package: `matlab.io.datastore`

Read data in TDMS datastore

Syntax

```
data = read(tdmsds)
[data,info] = read(tdmsds)
```

Description

`data = read(tdmsds)` reads data from the files in the TDMS datastore `tdmds`, and returns a cell array of tables or timetables. Each element of the cell array corresponds to a channel group in the datastore file data.

The `read` function returns a subset of data from the datastore. The size of the subset is determined by the `ReadSize` property of the datastore object. On the first call, `read` starts reading from the beginning of the datastore, and subsequent calls continue reading from the endpoint of the previous call. Use `reset` to read from the beginning again.

The function returns a cell array of tables or a cell array of timetables, depending on the value of the `tdmds.RowTimes` property. See “Read TDMS-File Data into Timetables” on page 15-58.

`[data,info] = read(tdmsds)` also returns the output argument `info`, with information and metadata about the extracted data.

Examples

Read Datastore by Files

Read data from a TDMS datastore one file at a time. Set the read size and read the first data set.

```
tdmsds = tdmsDatastore("C:\data\tdms",ReadSize="file");
data1 = read(tdmsds);
```

Read the second file and view information about the data.

```
[data2,info2] = read(tdmsds);
info2
```

```
info2 = 
  struct with fields:

    Filename: "C:\data\tdms\Turbine_002.tdms"
    FileSize: 172098
      Offset: 0
```

Read TDMS-File Data into Timetables

By providing a vector of durations, you can read TDMS-file data into timetables.

Define a vector of 1000 elements of 1 ms duration. Set up the datastore object to read 1000 records (ReadSize) and return a timetable (RowTimes).

```
durvec = milliseconds(1:1000);
tdmsds = tdmsDatastore("C:\data\tdms",ReadSize=1000,RowTimes=durvec)
```

```
tdmsds = 

  TDMSDatastore with properties:

                  Files: ["C:\data\tdms\Turbine_001.tdms"    "C:\data\tdms\Turbine_002.tdms"
            ChannelList: [6×8 table]
    SelectedChannelGroup: [0×0 string]
        SelectedChannels: [0×0 string]
                RowTimes: [0.001 sec    0.002 sec    0.003 sec    0.004 sec    0.005 sec    0.00
                ReadSize: 1000
```

Read 1000 records from the datastore into timetables.

```
dd = read(tdmsds)
```

```
dd =

  1×3 cell array

    {1000×2 timetable}    {1000×2 timetable}    {1000×2 timetable}
```

View part of one of the timetables.

```
dd{1}
```

```
ans =

  1000×2 timetable

      Time       Acceleration1    Acceleration2
    _________    _____________    _____________

    0.001 sec       -1.9851                 0
    0.002 sec       -3.9702                 0
    0.003 sec        11.911            1.5521
    0.004 sec        5.9553           -1.5521
    0.005 sec        1.9851           -4.6562
    0.006 sec        5.9553            4.6562
    0.007 sec        3.9702           -1.5521
    0.008 sec        3.9702           -4.6562

        :              :                 :

    0.993 sec        50.656           -469.05
    0.994 sec        46.686           -475.26
    0.995 sec         40.73           -472.16
    0.996 sec         40.73           -462.84
    0.997 sec        34.775           -461.29
    0.998 sec        38.745           -472.16
```

```
0.999 sec        38.745           -464.4
1 sec            34.775          -462.84

Display all 1000 rows.
```

Input Arguments

`tdmsds` — TDMS datastore

`TDMSDatastore` object

TDMS datastore, specified as a `TDMSDatastore` object.

Example: `tdmsds = tdmsDatastore("C:\data\tdms")`

Output Arguments

`data` — Output data

cell array of tables

Output data, returned as a cell array of tables from the TDMS records.

`info` — Information about data

structure array

Information about the data source file, returned as a structure with the following fields:

```
Filename
FileSize
Offset
```

The `offset` field indicates the position of the data in the file.

Version History

Introduced in R2022a

See Also

Functions

`tdmsDatastore` | `reset` | `preview` | `readall` | `hasdata`

readall

Package: `matlab.io.datastore`

Read all data in TDMS datastore

Syntax

`data = readall(tdmsds)`

Description

`data = readall(tdmsds)` reads all the data in the datastore specified by `tdmsds`, and returns a cell array of tables or timetables. Each element of the cell array corresponds to a channel group in the datastore file data.

The function returns a cell array of tables or a cell array of timetables, depending on the value of the `tdmds.RowTimes` property. See `tdmsDatastore`.

After the `readall` function returns all the data, it resets `tdmsds` to point to the beginning of the datastore.

Examples

Read All the Data in a Datastore

Read all the data from a multiple file TDMS datastore into an array of tables.

Set up datastore and read all its data.

```
tdmsds = tdmsDatastore("C:\data\tdms");
data = readall(td)
```

```
data =

  1×3 cell array

    {9936×2 table}    {9936×2 table}    {9936×2 table}
```

View part of the data.

```
data{1}
```

```
ans =

  9936×2 table

    Acceleration1    Acceleration2
    _____________    _____________

       -1.9851                 0
       -3.9702                 0
        11.911            1.5521
        5.9553           -1.5521
        1.9851           -4.6562
        5.9553            4.6562
        3.9702           -1.5521
```

```
    3.9702          -4.6562

      :                :

   -4.8046           6.7826
   -7.2068           2.2609
   -7.2068           4.5218
   -7.2068           6.7826
   -2.4023           9.0435
   -2.4023           4.5218
   -9.6091           2.2609
   -12.011           4.5218
```

Input Arguments

`tdmsds` — TDMS datastore
`TDMSDatastore` object

TDMS datastore, specified as a `TDMSDatastore` object.

Example: `tdmsds = tdmsDatastore("C:\data\tdms")`

Output Arguments

`data` — Output data
cell array of tables

Output data, returned as a cell array of tables from all TDMS-files in the datastore.

Version History

Introduced in R2022a

See Also

Functions
`tdmsDatastore` | `read` | `preview`

readwrite

Package: `daq.interfaces`

Simultaneously read and write device channel data

Syntax

```
inScanData = readwrite(d,outScanData)
[inScanData,triggerTime] = readwrite(d,outScanData)
inScanData = readwrite(d,outScanData,"OutputFormat","Matrix")
[inScanData,timeStamp,triggerTime] = readwrite( ___ ,"OutputFormat","Matrix")
```

Description

`inScanData = readwrite(d,outScanData)` writes `outScanData` to the DataAcquisition interface output channels, and reads `inScanData` from the DataAcquisition interface input channels. Input and output have the same number of scans, determined by the number of rows in the matrix `outScanData`. By default, data is returned to `inScanData` as a timetable. `readwrite` supports only foreground clocked operations, blocking MATLAB until complete.

`[inScanData,triggerTime] = readwrite(d,outScanData)` performs the read and write operations, and also returns the scan trigger time to `triggerTime` as a datetime.

`inScanData = readwrite(d,outScanData,"OutputFormat","Matrix")` performs the read and write operations, returning a matrix of double values to `inScanData`.

`[inScanData,timeStamp,triggerTime] = readwrite( ___ ,"OutputFormat","Matrix")` performs the read and write operations, also returning the scan times as a column vector of doubles to `timeStamps`, and the scan trigger time to `triggerTime` as a datenum double. The rows of the `timeStamp` vector correspond to the rows of the `inScanData` matrix.

Examples

Measure and Generate Simultaneously

Configure the DataAcquisition to measure and generate voltage simultaneously, in the foreground.

```
d = daq("ni");
addinput(d, "Dev1","ai0","Voltage");
addoutput(d, "Dev1","ao0","Voltage");
outScanData = linspace(0,1,d.Rate)'; % Increase output voltage with each scan.
inScanData = readwrite(d,outScanData);
```

Input Arguments

d — DataAcquisition interface

`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

`outScanData` — Scan data for device output
double matrix

Scan data for device output, specified as an M-by-N matrix, where M is the number of data scans and N is the number of output channels in the DataAcquisition interface. For a single channel, the data is a column vector. Single scans are not supported by this function, so M must be greater than 1.

Data Types: `double`

Output Arguments

`inScanData` — Input scan data from the device
timetable | double

Input scan data from the device, returned as a timetable or matrix of doubles, depending on the `OutputFormat` setting.

You can access the scan trigger time in the timetable property `inScanData.Properties.CustomProperties.TriggerTime`, returned as a datetime.

`triggerTime` — Time that acquisition began
datetime | datenum double

Time that acquisition began, returned as a datetime if `OutputFormat` is `"Timetable"` (default), or as a double if `OutputFormat` is `"Matrix"`. This information is also available as a datetime value in the timetable property `inScanData.Properties.CustomProperties.TriggerTime`.

`timeStamp` — Times of scan acquisitions
double

Times of scan acquisitions, returned as a matrix of doubles. Each value represents relative time in seconds after the first scan. This argument is returned only when `OutputFormat` is specified as `"Matrix"`.

Version History
Introduced in R2020a

See Also

Functions
`daq`

removechannel

Package: `daq.interfaces`

Remove channel from device interface

Syntax

`removechannel(d,idx)`

Description

`removechannel(d,idx)` removes the specified channels from the DataAcquisition interface. If the DataAcquisition has channels with indices higher than the channels being removed, they are renumbered to fill the empty gaps left by the removal, but the channel names do not change.

Examples

Remove Channels from DataAcquisition Interface

Remove channels from a DataAcquisition and note the index changes.

```
d = daq("directsound");
addinput(d,"Audio0","1","Audio");
addinput(d,"Audio1","1","Audio");
addoutput(d,"Audio3","1","Audio");
addoutput(d,"Audio6","1","Audio");
d.Channels
```

```
index    Type      Device     Channel   Measurement Type        Range          Name
_____    ______    ________   _______   ________________    ______________    _____

  1      "audi"    "Audio1"     "1"         "Audio"         "-1.0 to +1.0 "   "ch1"
  2      "audi"    "Audio0"     "1"         "Audio"         "-1.0 to +1.0 "   "ch2"
  3      "audo"    "Audio3"     "1"         "Audio"         "-1.0 to +1.0 "   "ch3"
  4      "audo"    "Audio6"     "1"         "Audio"         "-1.0 to +1.0 "   "ch4"
```

```
removechannel(d,2)
d.Channels
```

```
index    Type      Device     Channel   Measurement Type        Range          Name
_____    ______    ________   _______   ________________    ______________    _____

  1      "audi"    "Audio1"     "1"         "Audio"         "-1.0 to +1.0 "   "ch1"
  2      "audo"    "Audio3"     "1"         "Audio"         "-1.0 to +1.0 "   "ch3"
  3      "audo"    "Audio6"     "1"         "Audio"         "-1.0 to +1.0 "   "ch4"
```

Note that after removal of the second channel, the remaining channels are numbered 1, 2, and 3. The channel names are not changed.

Remove all remaining channels.

```
removechannel(d,[1:length(d.Channels)])
```

Input Arguments

d — DataAcquisition interface
DataAcquisition object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

idx — Channel index
numeric scalar | numeric vector

Channel index, specified as a numeric scalar or vector. Removing a channel shifts down the indices of remaining higher channels, but does not change the channel names. Do not confuse the channel index in the DataAcquisition with the channel ID of the data acquisition device.

Example: `[1,3]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2020a

See Also

Functions
`daqlist` | `daq` | `addinput` | `addoutput` | `addbidirectional`

removeclock

Package: `daq.interfaces`

Remove clock from device interface

Syntax

```
removeclock(d,idx)
```

Description

`removeclock(d,idx)` removes the specified clock from the DataAcquisition interface. If the DataAcquisition has clocks with indices higher than the clock being removed, they are renumbered to fill the empty gaps left by the removal.

Examples

Remove Clock from DataAcquisition Interface

Remove a clock from a DataAcquisition interface.

```
d = daq("ni");
% :
Cidx = addclock(d,"ScanClock","Dev1/PFI0","Dev2/PFI0");
% :
removeclock(d,Cidx);
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

idx — Clock index
numeric scalar | numeric vector

Clock index, specified as a numeric scalar or vector. Removing a clock shifts down the indices of remaining higher clocks.

Example: `1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2020a

See Also

Functions
daq | addclock

removetrigger

Package: `daq.interfaces`

Remove trigger from device interface

Syntax

`removetrigger(d,idx)`

Description

`removetrigger(d,idx)` removes the specified trigger from the DataAcquisition interface. If the DataAcquisition has triggers with indices higher than the trigger being removed, they are renumbered to fill the empty gaps left by the removal.

Examples

Remove Trigger from DataAcquisition Interface

Remove a trigger from a DataAcquisition interface.

```
d = daq("ni");
% :
Tidx = addtrigger(d,"Digital","StartTrigger","Dev1/PFI0","Dev2/PFI0");
% :
removetrigger(d,Tidx);
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

idx — Trigger index
numeric scalar | numeric vector

Trigger index, specified as a numeric scalar or vector. Removing a trigger shifts down the indices of remaining higher triggers.

Example: 1

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2020a

See Also

Functions
`daq` | `addtrigger`

reset

Package: `matlab.io.datastore`

Reset TDMS datastore to initial state

Syntax

`reset(tdmsds)`

Description

`reset(tdmsds)` resets the TDMS datastore specified by `tdmsds` to its initial read state, where no data has been read from it. Resetting allows you to reread from the same datastore.

Examples

Reset TDMS Datastore

Reset a TDMS datastore so that you can read from it again.

```
tdmsds = tdmsDatastore("C:\data\tdms");
data = read(tdmsds);
    ⋮
reset(tdmsds);
data = read(tdmsds);
```

Input Arguments

`tdmsds` — TDMS datastore
`TDMSDatastore` object

TDMS datastore, specified as a `TDMSDatastore` object.

Example: `tdmsds = tdmsDatastore("C:\data\tdms")`

Version History

Introduced in R2022a

See Also

Functions
`tdmsDatastore` | `read` | `hasdata`

resetcounters

Package: daq.interfaces

Reset hardware scan count for all counter inputs

Syntax

```
resetcounters(d)
```

Description

`resetcounters(d)` resets hardware scan count for all counter inputs between on-demand reads on the DataAcquisition `d`.

Examples

Acquire Edge Count and Reset Counter

Configure a DataAcquisition to measure an EdgeCount until the count exceeds a threshold, then reset the counter.

```
d = daq("ni");
addinput(d,"Dev1","ctr0","EdgeCount");
maxCount = 100;
count = read(d);
while count <= maxCount
    count = read(d);
end
resetcounters(d);
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

Version History

Introduced in R2020a

start

Package: `daq.interfaces`

Start DataAcquisition background operation

Syntax

```
start(d)
start(d,"Continuous")
start(d,"RepeatOutput")
start(d,"Duration",span)
start(d,"NumScans",span)
```

Description

`start(d)` starts the DataAcquisition interface background operation. When the input acquisition and output generation begin depends on channel configuration and preloaded output data:

- If the DataAcquisition has only input channels, the acquisition begins immediately, collecting scan data, which you can access later with the `read` function. The default scan duration is 1 second.
- If the DataAcquisition interface has only output channels, generation begins immediately if data is already queued with the `preload` function. If no data is queued, output begins when data is made available with `write` function.
- If the DataAcquisition has both input and output channels, the input acquisition begins and ends at the same time as the output generation, resulting in the same number of scans.

`start(d,"Continuous")` starts the background operation running continuously. If there is data already available from the `preload` function, output generation begins immediately along with acquisition on any input channels. Otherwise, acquisition begins when you execute `write`. The operation continues until you call `stop`. As output scan data is generated or input scan data is acquired, you might need to call `write` or `read` while the DataAcquisition is still running.

`start(d,"RepeatOutput")` starts the background operation, generating periodic output in a repeating loop of the output scan data. If there is data already available from the `preload` function, output generation begins immediately along with acquisition on any input channels. Otherwise, generation and acquisition begin when you execute `write`. The operation continues until you call `stop`. If input scan data is being acquired, you might need to call `read` while the DataAcquisition is still running.

`start(d,"Duration",span)` or `start(d,"NumScans",span)` starts the background input acquisition to run for a finite span of time, specified as either a duration or a number of scans. If the DataAcquisition has any output channels, the start occurs but the duration specification is ignored.

Examples

Read Data from a Background Acquisition

When a background acquisition is initiated with the `start` function, use `read` to import the data.

```
d = daq("ni");
ch = addinput(d,"Dev1",1:2,"Voltage")
start(d,"NumScans",5)

Background operation has started.
Background operation will stop after 0.005 s.
To read acquired scans, use read.

scanData = read(d,"all")

scanData =

  5×2 timetable

      Time       Dev1_ai1    Dev1_ai2
    _________    ________    ________

    0 sec        0.012466    0.023977
    0.001 sec    0.019373    0.023319
    0.002 sec    0.021017     0.02299
    0.003 sec    0.021346     0.02299
    0.004 sec    0.022661    0.023648
```

Generate a Repeating Signal in the Background

Define and preload data for device output, then start output generation to repeat in the background while MATLAB continues.

```
d = daq("ni");
addoutput(d,"Dev1",1,"Voltage");
signalData = sin((1:1000)*2*pi/1000);
preload(d,signalData') % Column of data for one channel
start(d,"RepeatOutput")
% Device output now repeated while MATLAB continues.
stop(d)
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

span — Length of background operation
duration | double

Length of background operation, specified as a duration or double. For `"Duration"` specify a duration type; for `"NumScans"` specify a double for the number of scans. The default is 1 second.

Example: `seconds(5)`

Data Types: `double` | `duration`

Version History

Introduced in R2020a

See Also

Functions

`daq` | `preload` | `stop` | `read` | `write`

Topics

"Generate Signals in the Background" on page 6-13
"Generate Signals in the Background Continuously" on page 6-14
"Generate Continuous and Background Signals Using NI Devices" on page 18-57
"Acquire Continuous and Background Data Using NI Devices" on page 18-18
"Log Analog Input Data to a File Using NI Devices" on page 18-64
"Communicate with I2C Devices and Analyze Bus Signals Using Digital IO" on page 18-118
"Generate Pulse Width Modulated Signals Using NI Devices" on page 18-108
"Acquire Continuous Audio Data" on page 18-83
"Create an Echometer Using Audio Measurements" on page 18-162
"Acquire Data from Two Devices at Different Rates" on page 18-136

stop

Package: `daq.interfaces`

Stop background operation

Syntax

`stop(d)`

Description

`stop(d)` stops the DataAcquisition interface background operations, and flushes queued output data. Input data acquired by the operation is not flushed.

Examples

Stop DataAcquisition Operations

Stop DataAcquisition interface operations.

```
d = daq("ni")
% :
start(d)
% :
stop(d)
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

Version History

Introduced in R2020a

See Also

Functions
`daq` | `start`

tdmsDatastore

Datastore for collection of TDMS-files

Description

Use the `TDMSDatastore` object to access data from a collection of TDMS-files.

Creation

Syntax

```
tdmsds = tdmsDatastore(location)
tdmsds = tdmsDatastore(__,Name=Value)
```

Description

`tdmsds = tdmsDatastore(location)` creates a `TDMSDatastore` object based on a TDMS-file or a collection of files in the folder specified by `location`. Within the folder, all files with the extension `.tdms` are included in the datastore.

`tdmsds = tdmsDatastore(__,Name=Value)` specifies function options and properties of `tdmsds` using optional name-value pairs.

Input Arguments

`location` — Location of TDMS datastore files
string | character vector | cell array

Location of TDMS datastore files, specified as a string, character vector, or cell array identifying either files or folders. The path can be relative or absolute, and can contain the wildcard characters `?` and `*`. If `location` specifies a folder, the datastore includes all files in that folder with the extension `.tdms`.

Example: `"C:\data\tdms_set1"`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

You can specify file information or object “Properties” on page 15-78. Allowed options are `IncludeSubfolders`, `AlternateFileSystemRoots`, and the properties `SelectedChannelGroup`, `SelectedChannels`, `RowTimes`, and `ReadSize`.

Example: `SelectedChannelGroup="Acceleration"`

`IncludeSubfolders` — Include files in subfolders
`false` (default) | `true`

Include files in subfolders, specified as a logical. Specify `true` to include files in each folder and recursively in subfolders.

Example: `IncludeSubfolders=true`

Data Types: `logical`

`AlternateFileSystemRoots` — Root paths for different platforms
string array | cell array

Root paths to the TDMS-files for different platforms, specified as an array of strings.

Example: `AlternateFileSystemRoots=["Z:\datasets", "/tdms/datasets"]`

Data Types: `char` | `string` | `cell`

Properties

`Files` — Files included in datastore
char | string | cell

This property is read-only.

Files included in the datastore, specified as a character vector or string identifying a relative or absolute path to a file or folder. The wildcard characters `?` and `*` are supported. All TDMS-files in the specified folder are included in the datastore. The property value is stored as a string vector of file names.

Example: `"file*.tdms"`

Data Types: `char` | `string`

`ChannelList` — All channels present in first TDMS-file
table

This property is read-only.

All channels present in first TDMS-file, returned as a table.

Those channels targeted for reading must have the same name and belong to the same channel group in each file of the TDMS datastore.

Data Types: `table`

`SelectedChannelGroup` — Channel group containing the channels to read from
string | char

Channel group containing the channels to read from, specified as a string or character vector.

Example: `"Torque"`

Data Types: `string` | `char`

`SelectedChannels` — Names of channels to read
char | string | cell

Names of channels to read, specified as a character vector, string, or array of either. The channels must be in the channel group specified by `SelectedChannelGroup` in each file of the TDMS datastore.

Example: `["Torque1" "Torque2"]`

Data Types: `char` | `string` | `cell`

`RowTimes` — Times associated with rows of table
datetime | duration | channel name

Times associated with rows of the table, specified as a selected time channel name, a datetime vector, or a duration vector. Setting this property causes the `read` and `readall` functions to output a cell array of timetables. Each time element labels a row in the output timetable.

Example: `duration(seconds([1:1000]/1000))`

Data Types: `datetime` | `duration` | `string`

`ReadSize` — Size of data returned by `read`
`20000` (default) | numeric | `"file"`

Size of data returned by the `read` function, specified as `"file"` or a numeric value. A string value of `"file"` causes a read of one TDMS-file at a time; a numeric value specifies the number of records to read. The `readall` function ignores this property.

If you change the `ReadSize` property value after creating the `TDMSDatastore` object, the datastore resets.

Example: `5000`

Data Types: `double` | `string` | `char`

Object Functions

read	Read data in TDMS datastore
readall	Read all data in TDMS datastore
preview	Read first 8 records from TDMS datastore
hasdata	Determine if data is available to read from TDMS datastore
reset	Reset TDMS datastore to initial state
combine	Combine data from multiple datastores
transform	Transform datastore

Examples

Read Data from a TDMS Datastore

Create a TDMS datastore from all the TDMS-files in the folder `C:\data\tdms`, and read the data into tables.

Set up the datastore and view its channel list.

```
td = tdmsDatastore("C:\data\tdms");
td.ChannelList(:,1:4)
```

```
ans =
```

```
  6×4 table

    ChannelGroupNumber    ChannelGroupName    ChannelGroupDescription      ChannelName
    __________________    ________________    _______________________    _______________

            1              "Acceleration"        "CGAcceleration"        "Acceleration1"
            1              "Acceleration"        "CGAcceleration"        "Acceleration2"
            2              "Force"               "CGForce"               "Force1"
            2              "Force"               "CGForce"               "Force2"
            3              "Torque"              "CGTorque"              "Torque1"
            3              "Torque"              "CGTorque"              "Torque2"
```

Read all available data.

```
td.SelectedChannelGroup = "Force"
data_set = readall(td)
```

```
data_set =

  1×3 cell array

    {9936×2 table}    {9936×2 table}    {9936×2 table}
```

View the data from the first channel group.

```
data_set{1}
```

```
ans =

  9936×2 table

    Acceleration1    Acceleration2
    _____________    _____________

       -1.9851                 0
       -3.9702                 0
        11.911            1.5521
        5.9553           -1.5521
        1.9851           -4.6562
        5.9553            4.6562
        3.9702           -1.5521
        3.9702           -4.6562
        13.896                 0
          :                  :

       -4.8046           -2.2609
       -4.8046            6.7826
       -7.2068            2.2609
       -7.2068            4.5218
       -7.2068            6.7826
       -2.4023            9.0435
       -2.4023            4.5218
       -9.6091            2.2609
       -12.011            4.5218

    Display all 9936 rows.
```

Read all available data for the channel group `Force`.

```
td.SelectedChannelGroup = "Force";
data_set = readall(td)
```

```
data_set =

  1×1 cell array

    {9936×2 table}
```

Read 500 entries of data.

```
td.ReadSize = 500;
data_set = read(td)
```

```
data_set =

  1×1 cell array

    {500×2 table}
```

Limitations

- As a special case of a datastore, the `TDMSDatastore` object does not support the following functionality:
  - Partitioning for parallel computing
  - Shuffling

Version History

Introduced in R2022a

See Also

External Websites

The NI TDMS File Format

tdmsinfo

Information about TDMS-file

Syntax

```
info = tdmsinfo(tdmsfile)
```

Description

`info = tdmsinfo(tdmsfile)` returns a `TdmsInfo` object with properties containing general information about the TDMS-file, such as file name, location, description, author, version, and list of channels.

Examples

Get Information on a TDMSFile

Get information on a TDMS-file and its channels.

```
info = tdmsinfo("Turbine_003.tdms")
```

```
info = 

  TdmsInfo with properties:

           Path: "C:\data\tdms\Turbine_003.tdms"
           Name: "Turbine_003"
    Description: "Test the Acceleration, Force and Torque of Turbine"
          Title: "Turbine Tests"
         Author: "xyz"
        Version: "2.0"
    ChannelList: [6×8 table]
```

View the channel information in the file.

```
>> info.ChannelList
```

```
ans =

  6×8 table
```

ChannelGroupNumber	ChannelGroupName	ChannelGroupDescription	ChannelName	ChannelDescription	Unit	DataType
1	"Acceleration"	"CGAcceleration"	"Acceleration1"	"from Clipboard"	"m/s^2"	"Double"
1	"Acceleration"	"CGAcceleration"	"Acceleration2"	"from Clipboard"	"m/s^2"	"Double"
2	"Force"	"CGForce"	"Force1"	"from Clipboard"	"N"	"Double"
2	"Force"	"CGForce"	"Force2"	"from Clipboard"	"N"	"Double"
3	"Torque"	"CGTorque"	"Torque1"	"from Clipboard"	"Nm"	"Double"
3	"Torque"	"CGTorque"	"Torque2"	"from Clipboard"	"Nm"	"Double"

Input Arguments

`tdmsfile` — TDMS file name

string

TDMS file name, specified as a string.

For local files, use a full or relative path that contains a file name and extension. You also can specify a file on the MATLAB path.

For Internet files, specify the URL. For example, to read a remote file from the Amazon S3 cloud:

```
data = tdmsread("s3://bucketname/path_to_file/data.tdms");
```

Example: "airlinesmall.tdms"

Data Types: char | string

Output Arguments

info — TDMS-file information

TdmsInfo object

TDMS-file information, returned as a TdmsInfo object with the following properties:

Property	Type	Description
Path	string	Full path to file
Name	string	TDMS-file name attribute
Description	string	TDMS-file description attribute
Title	string	TDMS-file title attribute
Author	string	TDMS-file author attribute
Version	string	TDMS-file version attribute
ChannelList	table	TDMS channel and channel group attributes

The ChannelList property value is a table with the following variables:

Table Variable	Type	Notes
ChannelGroupNumber	double	Internally created representational index of channel group
ChannelGroupName	string	
ChannelGroupDescription	string	
ChannelName	string	
ChannelDescription	string	
Unit	string	Unit of channel data
DataType	string	TDMS datatype in file
NumSamples	uint64	Number of samples in channel

Version History

Introduced in R2022a

See Also

Functions

`tdmsread` | `tdmsreadprop` | `tdmswriteprop` | `tdmswrite`

tdmsread

Read data from TDMS-file

Syntax

```
data = tdmsread(tdmsfile)
data = tdmsread(tdmsfile,Name=Value)
```

Description

`data = tdmsread(tdmsfile)` retrieves data from the specified TDMS-file and returns a cell array of tables. Each element of the cell array is a table corresponding to a channel group.

`data = tdmsread(tdmsfile,Name=Value)` uses name-value pairs to filter the data reading and specify output format.

Examples

Read TDMS File Data

Read data from a specified TDMS-file. You can determine which channels are read, and what format the result has.

Read all data from a TDMS-file into a table.

```
data = tdmsread("airlinesmall.tdms");
```

Read a subset of the variables in a TDMS-file into MATLAB as a timetable. Use the variable `ArrTime` in the TDMS-file as the time vector of the output timetable.

```
data = tdmsread("airlinesmall.tdms", ...
           ChannelGroupName = "Airline", ...
           ChannelNames = ["ArrTime" "FlightNum" "ArrDelay"], ...
           RowTimes = "ArrTime");
```

Read the channel data into a timetable with a specified start time and step duration.

```
data = tdmsread("airlinesmall.tdms", ...
           ChannelGroupName = "Airline", ...
           ChannelNames = ["ArrTime" "FlightNum" "ArrDelay"], ...
           TimeStep = seconds(0.01), StartTime = seconds(30));
```

Input Arguments

`tdmsfile` — TDMS file name
string

TDMS file name, specified as a string.

For local files, use a full or relative path that contains a file name and extension. You also can specify a file on the MATLAB path.

For Internet files, specify the URL. For example, to read a remote file from the Amazon S3 cloud:

```
data = tdmsread("s3://bucketname/path_to_file/data.tdms");
```

Example: `"airlinesmall.tdms"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ChannelGroupName="Torque",ChannelNames="Torque1"`

Supported name-value pairs are:

`ChannelGroupName` — Channel group containing the channels to read from
string | char

Channel group containing the channels to read from, specified as a string or character vector.

Example: `"Torque"`

Data Types: `string` | `char`

`ChannelNames` — Names of channels to read
char | string | cell

Names of channels to read, specified as a string, string array, character vector, or cell array of character vectors. The channels must be in the channel group specified by `ChannelGroupName`.

Example: `["Torque1" "Torque2"]`

Data Types: `char` | `string` | `cell`

`RowTimes` — Times associated with rows of table
datetime | duration | channel name

Times associated with rows of the table, specified as a selected time channel name, a datetime vector, or a duration vector. Specifying this option causes the function to output a cell array of timetables. Each time element labels a row in the output timetable.

Example: `duration(seconds([1:1000]/1000))`

Data Types: `datetime` | `duration` | `string`

`StartTime` — Start time of output timetable
datetime | duration

Start time of the output timetable, specified as a scalar datetime or duration indicating the time of the first data record in the timetable.

Example: `StartTime=seconds(60)`

Data Types: `datetime` | `duration`

`SampleRate` — Sample rate of output timetable
double

Sample rate of the output timetable, specified as a positive scalar double indicating samples per second.

Example: `SampleRate=1000`

Data Types: `double`

`TimeStep` — Step time of output timetable
duration | calendarDuration

Time step of the output timetable, specified as a scalar duration or calendarDuration indicating the time span between data records.

Example: `TimeStep=seconds(0.01)`

Data Types: `duration` | `calendarDuration`

Output Arguments

`data` — Output data
cell array of tables

Output data, returned as a cell array of tables or timetables with data records from the TDMS-file. Each element of the cell array is a table or timetable for a channel group. The cell array index corresponds to the channel group number.

When the start time for the first sample is 0 and the sample times are relative to that (duration), the sample times returned to the timetable are based on seconds since the epoch in the local time zone equivalent to 01/01/1904 00:00:00.00 UTC (using the Gregorian calendar and ignoring leap seconds). For more information, see TDMS File Format Internal Structure.

Version History

Introduced in R2022a

See Also

Functions
`tdmsinfo` | `tdmsreadprop` | `tdmswrite` | `tdmswriteprop`

tdmsreadprop

Read properties as single row table from TDMS-file

Syntax

```
props = tdmsreadprop(tdmsfile)
props = tdmsreadprop(tdmsfile,Name=Value)
```

Description

`props = tdmsreadprop(tdmsfile)` returns a table of properties from the specified TDMS-file.

`props = tdmsreadprop(tdmsfile,Name=Value)` uses name-value pairs to filter the information to get specific properties.

Examples

Read Properties from TDMS-File

Read the high level properties of a TDMS-file.

```
props = tdmsreadprop("Turbine_003.tdms")
```

```
props =

  1×7 table

        name             title          author                       description
    _____________    _______________    ______    ___________________________________________________

    "Turbine_003"    "Turbine Tests"    "xyz"     "Test the Acceleration, Force and Torque of Tu
```

Read the properties of one channel group.

```
props = tdmsreadprop("Turbine_003.tdms",ChannelGroupName="Torque")
```

```
props =

  1×2 table

      name       description
    ________    ___________

    "Torque"    "CGTorque"
```

Narrow the scope to a single channel.

```
props = tdmsreadprop("Turbine_003.tdms",ChannelGroupName="Torque",ChannelName="Torque2")
```

```
props =

  1×19 table
```

```
      name          datatype       ...
    _________     ___________      ...

    "Torque2"     "DT_DOUBLE"      ...
```

Filter on specific properties.

```
props = tdmsreadprop("Turbine_003.tdms",PropertyNames=["title" "datetime" "datestring"])
```

```
props =

  1×3 table

         title                      datetime                   datestring
    _______________    _____________________________    ____________

    "Turbine Tests"    2021-10-18 01:57:17.000000000    "10/18/2021"
```

Input Arguments

`tdmsfile` — TDMS file name
string

TDMS file name, specified as a string.

For local files, use a full or relative path that contains a file name and extension. You also can specify a file on the MATLAB path.

For Internet files, specify the URL. For example, to read a remote file from the Amazon S3 cloud:

```
data = tdmsread("s3://bucketname/path_to_file/data.tdms");
```

Example: `"airlinesmall.tdms"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ChannelGroupName="Torque",ChannelName="Torque2"`

Supported name-value pairs are:

`ChannelGroupName` — Channel group containing the channels to read from
string | char

Channel group containing the channels to read from, specified as a string or character vector.

Example: `"Torque"`

Data Types: `string` | `char`

`ChannelName` — Name of channel to read
char | string

Name of channel to read, specified as a string or character vector. The channel must be in the channel group specified by `ChannelGroupName`.

Example: `"Torque2"`

Data Types: `char` | `string`

`PropertyNames` — Property names to read
string | char | cell

Property names to read, specified as a string, string array, character vector, or cell array of character vectors.

Example: `["Torque1" "Torque2"]`

Data Types: `char` | `string` | `cell`

Output Arguments

`props` — Table of properties from TDMS-file
table

Properties in the TDMS-file, returned as a table.

Version History

Introduced in R2022a

See Also

Functions
`tdmswriteprop` | `tdmsinfo` | `tdmsread` | `tdmswrite`

tdmswrite

Write data to TDMS-file

Syntax

```
tdmswrite(tdmsfile,tdmsdata)
tdmswrite(tdmsfile,tdmsdata,ChannelGroupNames=chGrpName)
tdmswrite( ___ ,TimeChannel=timeChan)
```

Description

With the `tdmswrite` function you can write table or timetable data to a new or existing TDMS-file.

`tdmswrite(tdmsfile,tdmsdata)` writes data to the specified TDMS-file from a table, timetable, or cell array of tables or timetables. Each table is written to the file as a new channel group, automatically incrementing the channel group name with each write.

`tdmswrite(tdmsfile,tdmsdata,ChannelGroupNames=chGrpName)` specifies an existing channel group to write the data to. When specifying data as a cell array, use a cell array of strings to identify the corresponding channel group names, sequentially mapped by elements.

`tdmswrite( ___ ,TimeChannel=timeChan)` specifies how measurement time is included in the file when writing data from a timetable. A `TimeChannel` value of `"none"` adds the start time and step time to the channel properties. A value of `"single"` adds a single channel with a timestamp for every measurement. If you are writing data from a regular table, the `TimeChannel` setting is ignored.

Examples

Read TDMS File Data

Write data to a specified TDMS-file. You can use default channel groups or specify channel group names.

Write a table or timetable of data, `T`, to a new channel group in the TDMS-file named `sinewave.tdms`.

```
tdmswrite("sinewave.tdms",T)
```

Write table or timetable of data, `T`, to a specific channel group in a TDMS-file. If the channel group does not exist, it is added to the file.

```
tdmswrite("sinewave.tdms", T, ChannelGroupNames="MeasuredData")
```

Write two tables of data to multiple channel groups in a TDMS-file.

```
tdmswrite("sinewave.tdms", {T1,T2}, ChannelGroupNames=["Measurement1" "Measurement2"])
```

Input Arguments

`tdmsfile` — TDMS file name
string

TDMS file name, specified as a string.

For local files, use a full or relative path that contains a file name and extension. You also can specify a file on the MATLAB path.

Example: `"sample332.tdms"`

Data Types: `char` | `string`

`tdmsdata` — TDMS data
table | timetable | cell array of tables and timetables

TDMS data, specified as table, timetable, or cell array of tables and timetables. Alternatively, you can specify several tables or timetables as a series of arguments, such as `T1,T2,T3`.

For a duration timetable, the written start time is 0. When reading this file with `tdmsread`, the start time is the epoch in the local time zone equivalent to 01/01/1904 00:00:00.00 UTC (using the Gregorian calendar and ignoring leap seconds). For more information, see TDMS File Format Internal Structure.

Data Types: `table` | `timetable` | `cell`

`chGrpName` — Channel group name
string | char

Channel group name, specified as a string or character vector. Use an array of channel group names when writing multiple tables.

- If the channel group does not exist in the TDMS-file, a new channel group is created.
- If the channel group exists, data is appended to channels with names matching the table variables. New channels are added to the channel group for table variables not already represented by existing channel names.

Example: `"ChannelGroup1"`

Data Types: `char` | `string` | `cell`

`timeChan` — Time channel format
`"single"` (default) | `"none"`

Time channel format layout, specified as a string or character vector with value `"single"` or `"none"`:

- A value of `"single"` (default) adds a single channel with a timestamp for every measurement. This is appropriate for timetables with irregular timing, when each measurement has a unique datetime or duration, shared across the channels in the channel group. This `Time` channel is derived from the `Time` variable of the input timetable.

- A value of `"none"` adds only the start time and step time to the channel properties `wf_start_time` and `wf_increment`, respectively. Appropriate for regular timetables with fixed sample rates, this option can reduce the size of the TDMS-file.

Example: `"none"`

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

Functions
`tdmswriteprop` | `tdmsread` | `tdmsinfo` | `tdmsreadprop`

External Websites
TDMS Fragmentation: Why Your TDMS Files Use Too Much Memory

tdmswriteprop

Write properties to TDMS-file

Syntax

```
tdmswriteprop(tdmsfile,propname,propvalue)
tdmswriteprop(tdmsfile,propname,propvalue,ChannelGroupName=chGrpName)
tdmswriteprop(tdmsfile,propname,propvalue,ChannelGroupName=
chGrpName,ChannelName=chName)
```

Description

With the `tdmswriteprop` function you can write file properties, channel properties and channel group properties to a TDMS-file.

`tdmswriteprop(tdmsfile,propname,propvalue)` writes file property names and corresponding values to the specified TDMS-file. You can set a single property or multiple properties. To set multiple properties, use arrays to specify property names and values. With `tdmswriteprop` you can specify and modify existing properties, or add new properties.

`tdmswriteprop(tdmsfile,propname,propvalue,ChannelGroupName=chGrpName)` and `tdmswriteprop(tdmsfile,propname,propvalue,ChannelGroupName=chGrpName,ChannelName=chName)` specify the existing channel group or channel that the property is assigned to. You can set the properties of only one channel or channel group at a time.

Examples

Write Properties to TDMS-File

Write the `title` file property of a TDMS-file.

```
tdmswriteprop("sinewave.tdms","title","Measurement Data")
```

Write a channel group description property of a TDMS-file.

```
tdmswriteprop("sinewave.tdms","description","Amplitude and Phase Sweep",...
    ChannelGroupName="ChannelGroup1")
```

Write a channel property of a TDMS-file.

```
tdmswriteprop("sinewave.tdms","max_value", max(data), ...
    ChannelGroupName="ChannelGroup1", ChannelName="Amplitude Sweep")
```

Write two properties of a channel.

```
tdmswriteprop("sinewave.tdms",["max_value" "min_value"],[max(data),min(data)] ...
    ChannelGroupName="ChannelGroup1", ChannelName="Amplitude Sweep")
```

Input Arguments

`tdmsfile` — TDMS file name
string | char

TDMS file name, specified as a string or character vector.

For local files, use a full or relative path that contains a file name and extension. You also can specify a file on the MATLAB path.

Example: `"sinewave.tdms"`

Data Types: `char` | `string`

`propname` — Property name
string | char

Property name, specified as a string or character vector. When writing multiple properties, use an array of strings to identify them. To create custom properties, specify a name that does not already exist.

Example: `"title"`

Data Types: `char` | `string`

`propvalue` — Property value
property-dependent

Property value, specified as a supported type for its property. To set multiple properties, use an array of values. If the values are different types, for example numeric and string, use a cell array.

Example: `{"Input Channel",2.0,"mV"}`

`chGrpName` — Channel group name
string | char

Channel group name, specified as a string or character vector.

Example: `"ChannelGroup1"`

Data Types: `char` | `string`

`chName` — Channel name
string | char

Channel name, specified as a string or character vector.

Example: `"Amplitude sweep"`

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

Functions
`tdmsinfo` | `tdmsreadprop` | `tdmsread` | `tdmswrite`

write

Package: `daq.interfaces`

Write output scans to hardware channels

Syntax

`write(d,scanData)`

Description

`write(d,scanData)` writes scan data to the DataAcquisition interface for the device output. The DataAcquisition might already be started or not.

- If the DataAcquisition has not been started, `write` sends the data and starts device output generation. As a finite foreground generation, this blocks MATLAB until completed.
- If the DataAcquisition had already been started, `write` provides the data for the output operation to begin, which then runs in the background without blocking MATLAB. The `start` function arguments determine if the generation is finite, repeating, or continuous. Continuous output requires `write` to provide data for as long as output is needed; multiple calls to `write` might be necessary.

Examples

Write a Single Scan

If the supplied data value specifies only a single scan of data for all output channels, the `write` function generates an on-demand output without clocking.

Create interface and add two output channels.

```
d = daq("ni");
ch = addoutput(d,"Dev1",0:1,"Voltage");
```

```
ch =

    Index    Type    Device    Channel      Measurement Type           Range            Name
    _____    ____    ______    _______    ____________________    ________________    __________

      1      "ao"    "Dev1"     "ao0"     "Voltage (SingleEnd)"   "-10 to +10 Volts"   "Dev1_ao0"
      2      "ao"    "Dev1"     "ao1"     "Voltage (SingleEnd)"   "-10 to +10 Volts"   "Dev1_ao1"
```

Output 5 volts on both channels.

```
write(d,[5 5])
```

Generate a Repeating Signal in the Background

Start a DataAcquisition interface for background operation, then provide data for device output.

```
d = daq("ni");
addoutput(d,"Dev1",1,"Voltage");
signalData = sin((1:1000)*2*pi/1000);
start(d,"RepeatOutput")
% :
write(d,signalData')
% Device output now repeated while MATLAB continues.
pause(5)
stop(d)
```

Input Arguments

d — DataAcquisition interface
`DataAcquisition` object

DataAcquisition interface, specified as a `DataAcquisition` object, created using the `daq` function.

Example: `d = daq()`

`scanData` — Scan data for device output
double matrix

Scan data for device output, specified as an M-by-N matrix, where M is the number of data scans and N is the number of output channels in the DataAcquisition interface. Each column of `scanData` contains the data for one channel. For a single channel, the data is a column vector.

Data Types: `double`

Version History

Introduced in R2020a

See Also

Functions
`start` | `read`

Apps

Analog Input Recorder

Acquire and visualize analog input signals

Description

The **Analog Input Recorder** app provides a graphical interface to data acquisition devices.

Using this app, you can:

- Configure device channels and acquisition properties.
- Preview signals on several analog input channels for a selected device.
- Record analog input data for a finite period (foreground) or continuously (background).
- Create scripts in the Live Editor from the app configuration.
- Open the Signal Analyzer app of Signal Processing Toolbox™ to perform analysis on your recorded data.

Open the Analog Input Recorder App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app.
- MATLAB command prompt: Enter `analogInputRecorder`.

Note Opening the **Analog Input Recorder** deletes all your existing DataAcquisition interfaces in MATLAB.

The DataAcquisition interface created by the **Analog Input Recorder** is not accessible from the MATLAB command line.

Limitations

This app supports devices only from the following vendors:

- National Instruments (`"ni"`)
- Windows Sound Cards (`"directsound"`)
- Analog Devices (`"adi"`)
- Measurement Computing (`"mcc"`)

The app device list includes only those devices with supported subsystems.

Version History

Introduced in R2017b

See Also

Apps
Analog Output Generator

Topics
"Acquire Data with Analog Input Recorder" on page 6-17

Analog Output Generator

Define and generate analog output signals

Description

The **Analog Output Generator** provides a graphical interface to data acquisition devices.

Using this app, you can:

- Configure device channels and properties.
- Define waveforms in a workspace variable as a vector of double values, or as a timetable.
- Preview defined signals on several analog output channels for a selected device.
- Generate analog or audio output signals for a finite period (foreground) or continuously (background).
- Create scripts in the Live Editor from the app configuration.

Open the Analog Output Generator App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app.
- MATLAB command prompt: Enter `analogOutputGenerator`.

Note Opening the **Analog Output Generator** deletes all your existing DataAcquisition interfaces in MATLAB.

The DataAcquisition interface created by the **Analog Output Generator** is not accessible from the MATLAB command line.

Limitations

The **Analog Output Generator** currently supports only analog voltage and current outputs, and audio output generation.

This app supports devices only from the following vendors:

- National Instruments (`"ni"`)
- Windows Sound Cards (`"directsound"`)
- Measurement Computing (`"mcc"`)

The app device list includes only those devices with supported subsystems.

Version History

Introduced in R2019a

See Also

Apps
Analog Input Recorder

Topics
“Generate Signals with Analog Output Generator” on page 6-21

Blocks

Analog Input

Acquire data from multiple analog channels of data acquisition device

Libraries:
Data Acquisition Toolbox

Description

The Analog Input block opens, initializes, configures, and controls an analog data acquisition device. The opening, initialization, and configuration of the device occur once at the start of the model execution. During the model run time, the block acquires data either synchronously (deliver the current block of data the device is providing) or asynchronously (stream buffered incoming data).

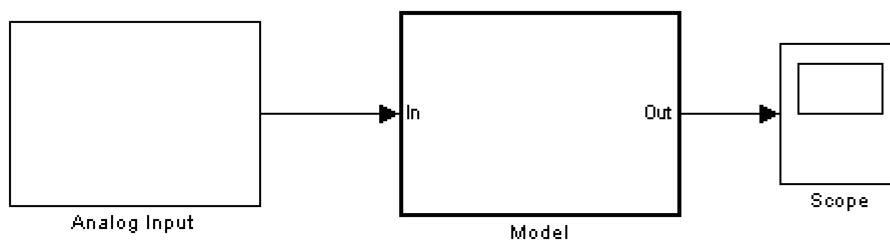
The block has no input ports. It has one or more output ports, depending on the configuration you choose in its dialog box.

Use the Analog Input block to incorporate live measured data into Simulink for:

- System characterization
- Algorithm verification
- System and algorithm modeling
- Model and design validation
- Controller design

The following diagram shows the basic analog input usage configuration, with which you can:

- Read acquired data at each time step or once per model execution.
- Analyze the data, or use it as input to a system in the model.
- Optionally display results.



Notes To use this block, you need both Data Acquisition Toolbox and Simulink software.

Some devices are not supported by the Simulink blocks in Data Acquisition Toolbox. To see if your device supports Simulink, refer to Supported Hardware.

You can use the Analog Input block only with devices that support clocked acquisition. To acquire data using devices that do not support clocking, use the Analog Input (Single Sample) block.

Other Supported Features

- If you have DSP System Toolbox™, you can use this block for signal applications.
- This block supports the use of Simulink Accelerator™ mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Ports

Output

Data — Acquired analog input
double

Acquired analog input data, returned as doubles. If using only one output port for all channels, each scan is available as a matrix of scan blocksize by number of channels, M-by-N. If using a port for each channel, each scan results in a blocksize-by-1 column vector on each port. Multiple ports are named by channel names or device specified channel IDs.

Data Types: `double`

Relative timestamp — Relative timestamps of scans

Relative timestamp of each scan, returned as a double. This port is available when you check the Output relative timestamps on page 17-0 parameter.

Data Types: `double`

Parameters

Use the Block Parameters dialog box to select your acquisition mode and to set other configuration options.

Device — Device from which you want to acquire data

The device from which you want to acquire data. The items in the list vary, depending on which devices you have connected to your system. Devices in the list are specified by adaptor or vendor name and unique device ID, followed by the model name of the device, for example, `ni Dev1 (USB-6255)`. The first available device is selected by default. A CompactDAQ chassis would be shown as a single device identified by vendor name, chassis ID, and chassis model; for example, `ni cDAQ2 (cDAQ-9172)`.

Acquisition Mode — Synchronous setting
Asynchronous | Synchronous

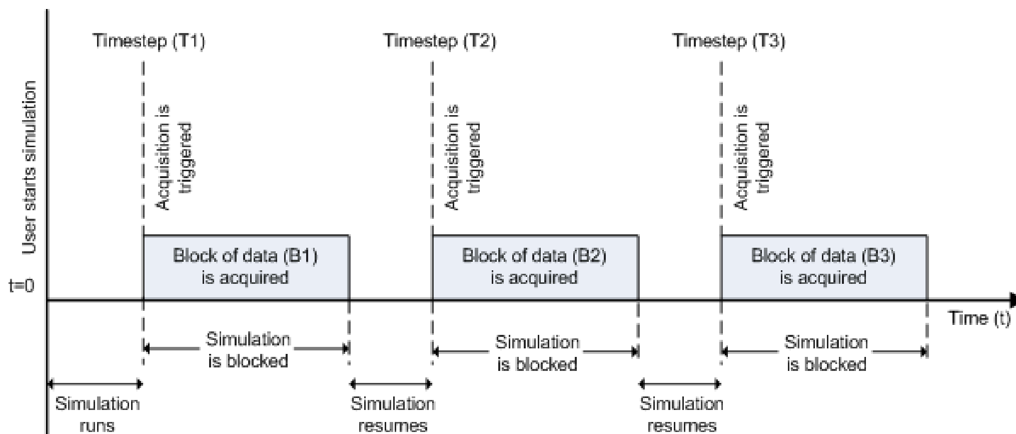
Synchronous setting, specified as one of the following options.

Asynchronous — In asynchronous mode, the data acquisition from the device and the simulation happen in parallel. The model initiates the acquisition from the device when the simulation starts. Data from the device is continuously acquired into a FIFO (first in, first out) buffer in parallel as the simulation runs. At each time step, the model fetches data from the FIFO buffer and outputs a block of data. The data in the FIFO buffer is contiguous according to the hardware acquisition clock.

Synchronous — In synchronous mode, the simulation is blocked while acquiring data from the device. The model initiates the acquisition from the device at each time step and immediately enters a wait state until the acquisition request has completed. This is unbuffered input; the block outputs the latest block of data at each time step.

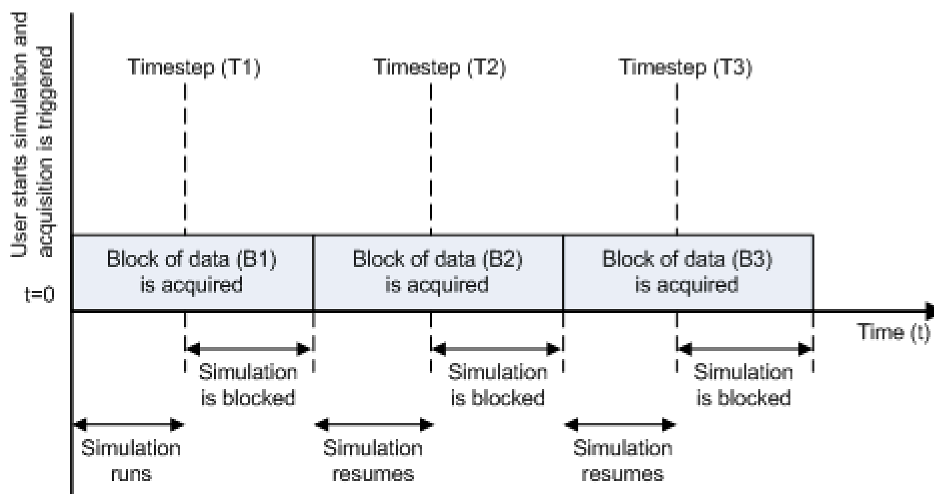
The following diagrams show the difference between synchronous and asynchronous modes for the Analog Input block.

Synchronous Analog Input



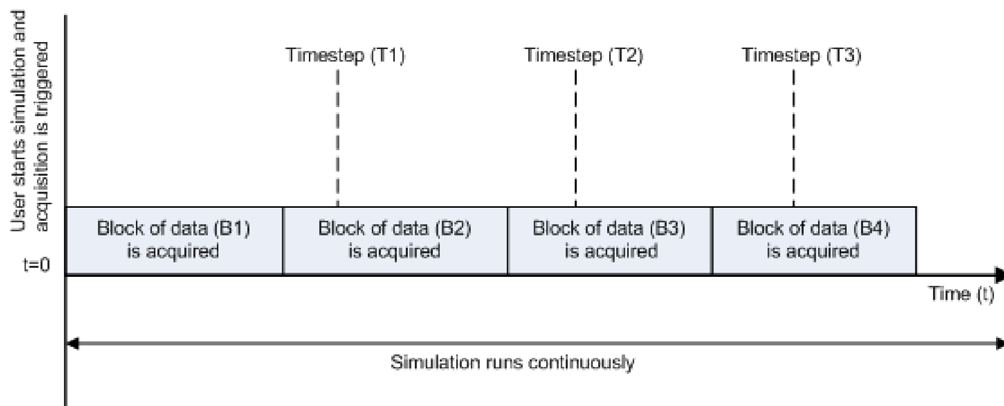
At the first time step (T1), the acquisition is initiated for the required block of data (B1). The simulation does not continue until B1 is completely acquired.

Asynchronous Analog Input – Scenario 1



Scenario 1 shows the case when simulation speed outpaces data acquisition speed. At the first time step (T1), the required block of data (B1) is still being acquired. Therefore, the simulation does not continue until B1 is completely acquired.

Asynchronous Analog Input – Scenario 2



Scenario 2 shows the case when data acquisition speed outpaces simulation speed. At the first time step (T1), the required block of data (B1) has been completely acquired. Therefore, the simulation runs continuously.

Note Several factors, including device hardware and model complexity, can affect the simulation speed, causing both scenarios 1 and 2 to occur within the same simulation.

Channels — Device channel selection and configuration
options depend on device

Device channel selection and configuration table. The channel configuration table lists the hardware channels of your device, and lets you select and configure them. Specify which channels to acquire data from (by default all the channels are selected). The following parameters are specified for each selected channel:

Channel ID — Hardware channel ID specified by the device. The Channel ID column is read-only, and the parameters are defined when the device is selected.

Name — Channel name. By default the table displays any names provided by the hardware, but you can edit the names. For example, if the device is a sound card with two channels, you can name them `Left` and `Right`.

Module — Device ID the channel belongs to. The Module column is read-only. If a CompactDAQ chassis is selected, it shows the ID of the CompactDAQ module which the channel belongs to; otherwise the ID of the device.

Measurement Type — Measurement type of the channel. This block supports only voltage measurement types. (For other measurement types, use a DataAcquisition object in MATLAB.)

Input Range — Input ranges available for each channel supported by the hardware, defined when a device is selected.

Terminal Configuration — Specifies the hardware terminal configuration, such as single-ended, differential, etc. The terminal configuration options are defined by the capabilities of the selected channel.

Coupling — Hardware coupling configuration, such as AC or DC. The coupling type is defined when a device is selected

Number of ports — Number of output data ports
1 for all channels | 1 per channel

Number of output data ports, specified as:

1 for all channels — Output data from a single port as a matrix, with a size of blocksize by number of channels selected.

1 per channel — Output data from N ports, where N is equal to the number of selected channels. Each output port is a column vector with a size of blocksize-by-1. For naming, each output port uses the channel name if one was specified, otherwise the channel ID, for example, `ai0`.

Input sample rate — Device sampling rate
numeric value

The rate at which samples are acquired from the device, in samples per second. This is the sampling rate for the hardware. The sample rate must be a positive real number within the range supported by the selected hardware.

Block size — Number of scans per time step
integer value

The number of data samples to read from the block output at each time step for each channel. It must be a positive integer greater than or equal to 2, within the range supported by the selected hardware.

Output relative timestamps — Add timestamp output port

Select this option to output the relative data timestamps, one for each sample. This option adds a new output port to the block. The data type of this port is double, and corresponds to the time offset in seconds of the sample related to the start of acquisition. For asynchronous acquisition, the acquisition is initiated once at the start of model execution, the relative timestamp is a monotonically-increasing number relative to the start of simulation. For synchronous acquisition, an acquisition is initiated at every time step; as a result, the relative timestamp is reset to zero every time an acquisition is initiated.

Version History

Introduced in R2016b

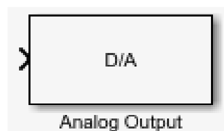
See Also

Blocks

Analog Output | Analog Input (Single Sample) | Analog Output (Single Sample) | Digital Input (Single Sample) | Digital Output (Single Sample)

Analog Output

Output data to multiple analog channels of data acquisition device

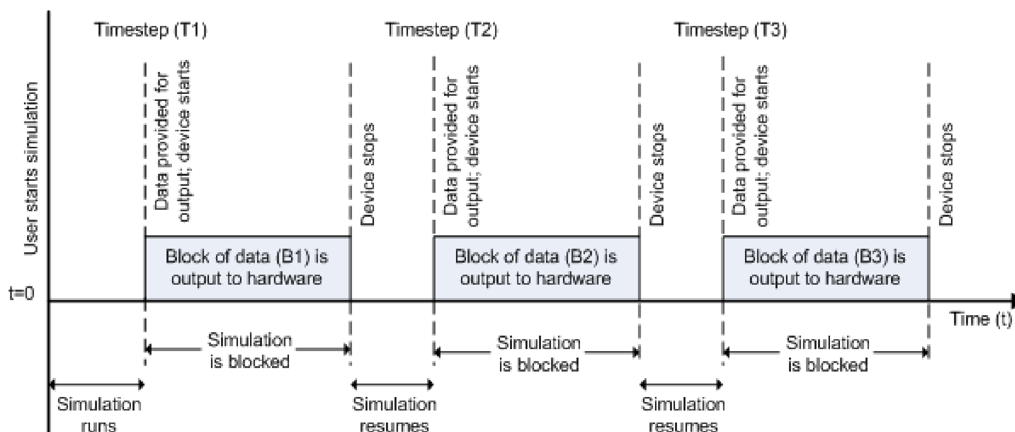


Libraries:
Data Acquisition Toolbox

Description

The Analog Output block opens, initializes, configures, and controls an analog data acquisition device. The opening, initialization, and configuration of the device occur once at the start of the model execution. During the model run time, the block outputs data to the hardware synchronously (outputs the block of data as it is provided). On every time step, the block performs a blocking synchronous write to the hardware, outputting the entire input data.

The following diagram shows the timing of the synchronous analog output.



At the first time step (T1), data output is initiated and the corresponding block of data (B1) is output to the hardware. The simulation does not continue until B1 is output completely.

The block has one or more input ports, depending on the option you choose in its parameters dialog box. It has no output ports.

The Analog Output block inherits the sample time from the driving block connected to the input port. The valid data types of the signal at the input port are double or native data types supported by the hardware.

Notes To use this block, you need both Data Acquisition Toolbox and Simulink software.

You can use the Analog Output block only with devices that support clocked generation. To generate data using devices that do not support clocking, use the Analog Output (Single Sample) block.

Some devices are not supported by the Simulink blocks in Data Acquisition Toolbox. To see if your device supports Simulink, refer to Supported Hardware.

Other Supported Features

- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Ports

Input

Data — Analog output to generate
double

Analog output to generate, specified as doubles. If using only one input port for all channels, specify an M-by-N matrix for a blocksize of M scans on N channels. Each scan is a row across N channels. Each channel outputs a column of M scans.

If using a port for each channel, specify a column of data for each channel on each port. Multiple ports are named by channel names or device specified channel IDs.

Data Types: `double`

Parameters

Device — Device through which you want to output data
select available device

The device from which you want to generate data. The items in the list vary, depending on which devices you have connected to your system. Devices in the list are specified by adaptor/vendor name and unique device ID, followed by the model name of the device, for example, `ni Dev1 (USB-6255)`. The first available device is selected by default. A CompactDAQ chassis is shown as a single device; vendor name, chassis ID, and chassis model would be shown in the list, for example, `ni cDAQ2 (cDAQ-9172)`.

Channels — Device channel selection and configuration
options depend on device

Device channel selection and configuration table. The channel configuration table lists the hardware channels of your device, and lets you select and configure them. Specify which channels to acquire data from (by default all the channels are selected). The following parameters are specified for each selected channel:

Channel ID — Hardware channel ID specified by the device. The Channel ID column is read-only, and the parameters are defined when the device is selected.

Name — Channel name. By default the table displays any names provided by the hardware, but you can edit the names. For example, if the device is a sound card with two channels, you can name them `Left` and `Right`.

Module — Device ID the channel belongs to. The Module column is read-only. If a CompactDAQ chassis is selected, it shows the ID of the CompactDAQ module which the channel belongs to; otherwise the ID of the device.

Measurement Type — Measurement type of the channel. This block supports only voltage measurement types. (For other measurement types, use a DataAcquisition object in MATLAB.)

Output Range — Output ranges available for each channel supported by the hardware, defined when a device is selected.

Number of ports — Number of input data ports
1 for all channels | 1 per channel

Number of input data ports, specified as:

1 for all channels (default) — One input port on the block for all channels. Provide data as a matrix, with a size of scan blocksize by number of channels, M-by-N.

1 per channel — N input ports on the block, where N is equal to the number of selected channels. Provide each port data as a column vector with a size of blocksize-by-1. For naming, each output port uses the channel name if one was specified, otherwise the channel ID, for example, `ao1`.

Output sample rate — Device sampling rate
numeric value

The rate at which samples are output from Simulink to the device, in samples per second. This is the sampling rate for the hardware. The default is defined when a device is selected. The sample rate must be a positive real number within the range allowed for the selected hardware.

Version History

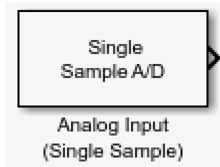
Introduced in R2016b

See Also

Blocks
Analog Input | Analog Input (Single Sample) | Analog Output (Single Sample) | Digital Input (Single Sample) | Digital Output (Single Sample)

Analog Input (Single Sample)

Acquire single sample from multiple analog channels of data acquisition device



Libraries:
Data Acquisition Toolbox

Description

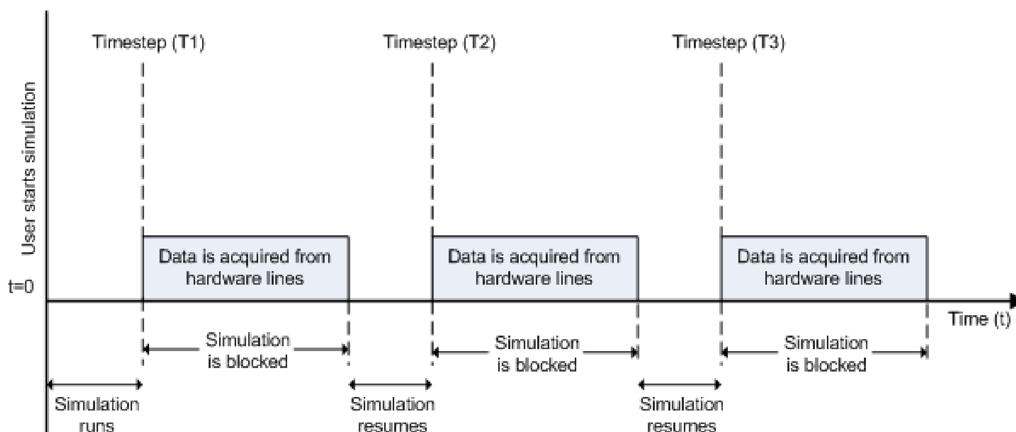
The Analog Input (Single Sample) block opens, initializes, configures, and controls an analog data acquisition device. The opening, initialization, and configuration of the device occur once at the start of the model execution. The block acquires a single sample every time step, synchronously from the device, during the model run time.

The block has no input ports. It has one or more output ports, depending on the configuration you choose in its dialog box.

Use the Analog Input (Single Sample) block to incorporate live measured data into Simulink for:

- System characterization
- Algorithm verification
- System and algorithm modeling
- Model and design validation
- Controller design

Analog input acquisition is done synchronously, according to the following diagram.



At the first time step (T1), data is acquired from the selected hardware channels. The simulation does not continue until data is read from all channels.

Notes To use this block, you need both Data Acquisition Toolbox and Simulink software.

Some devices are not supported by the Simulink blocks in Data Acquisition Toolbox. To see if your device supports Simulink, refer to Supported Hardware.

You can use Analog Input (Single Sample) block only with devices that support single sample acquisition. If the device does not support single sample acquisition, the model generates an error. To acquire data from devices that do not support acquisition of a single sample (such as devices designed for sound and vibration), use the Analog Input block.

Other Supported Features

- If you have DSP System Toolbox, you can use this block for signal applications.
- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Ports

Output

Data — Acquired analog input
double

Acquired analog input data, returned as doubles. If using only one output port for all channels, the output is an array of data. If using a port for each channel, each scan results in a single value on each port. Multiple ports are named by channel names or device specified channel IDs.

Data Types: `double`

Timestamp — Timestamp of scan

Timestamp of scan, returned as a double. This port is available when you check the Output timestamp on page 17-0 parameter.

Data Types: `double`

Parameters

Use the Block Parameters dialog box to select your device and to set other configuration options.

Device — Device from which you want to acquire data

The device from which you want to acquire data. The items in the list vary, depending on which devices you have connected to your system. Devices in the list are specified by adaptor or vendor name and unique device ID, followed by the model name of the device, for example, `ni Dev1 (USB-6255)`. The first available device is selected by default. A CompactDAQ chassis would be shown as a single device identified by vendor name, chassis ID, and chassis model; for example, `ni cDAQ2 (cDAQ-9172)`.

Channels — Device channel selection and configuration
options depend on device

Device channel selection and configuration table. The channel configuration table lists the hardware channels of your device, and lets you select and configure them. Specify which channels to acquire data from (by default all the channels are selected). The following parameters are specified for each selected channel:

Channel ID — Hardware channel ID specified by the device. The Channel ID column is read-only, and the parameters are defined when the device is selected.

Name — Channel name. By default the table displays any names provided by the hardware, but you can edit the names. For example, if the device is a sound card with two channels, you can name them `Left` and `Right`.

Module — Device ID the channel belongs to. The Module column is read-only. If a CompactDAQ chassis is selected, it shows the ID of the CompactDAQ module which the channel belongs to; otherwise the ID of the device.

Measurement Type — Measurement type of the channel. This block supports only voltage measurement types. (For other measurement types, use a DataAcquisition object in MATLAB.)

Input Range — Input ranges available for each channel supported by the hardware, defined when a device is selected.

Terminal Configuration — Specifies the hardware terminal configuration, such as single-ended, differential, etc. The terminal configuration options are defined by the capabilities of the selected channel.

Coupling — Hardware coupling configuration, such as AC or DC. The coupling type is defined when a device is selected

Number of ports — Number of output data ports
1 for all channels | 1 per channel

Number of output data ports, specified as:

1 for all channels — Outputs the acquired data from a single port as a 1-by-N vector with a length equal to the number of channels selected.

1 per channel — Outputs the acquired data from N ports, where N is equal to the number of selected channels. Each port output is a 1-by-1 double. For naming, each output port uses the channel name if one was specified, otherwise the channel ID, for example, `ai0`.

Sample time — Block execution rate
1 (default)

Specifies the sample time of the block during the simulation. This is the rate at which the block is executed during simulation. The default value is 1 (seconds). For more information, see “What Is Sample Time?” (Simulink).

Output timestamp — Add timestamp output port

Select this option to output the absolute timestamp of the scan. This option adds a new output port to the block. The data type of this port is double (datenum), which corresponds to a serial date number. You can convert the datenum into a datetime value with the `datetime` function.

Version History

Introduced in R2016b

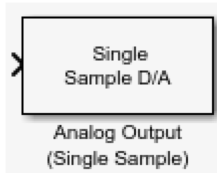
See Also

Blocks

Analog Input | Analog Output | Analog Output (Single Sample) | Digital Input (Single Sample) | Digital Output (Single Sample)

Analog Output (Single Sample)

Output single sample to multiple analog channels of data acquisition device



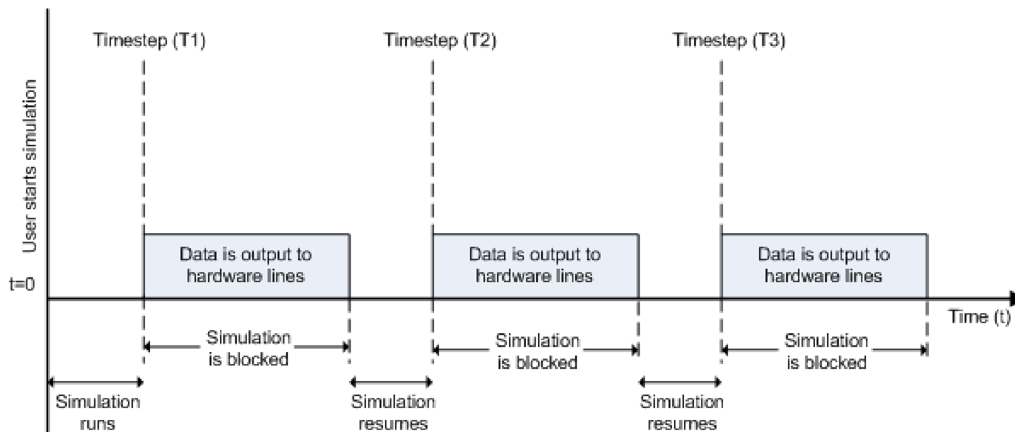
Libraries:
Data Acquisition Toolbox

Description

The Analog Output (Single Sample) block opens, initializes, configures, and controls an analog data acquisition device. The opening, initialization, and configuration of the device occur once at the start of the model execution. The block outputs a single sample every time step, synchronously to the hardware, during the model run time.

The block has one or more input ports, depending on the option you choose in its dialog box. It has no output ports. The valid data type of the signal at the input port is double.

The Analog Output (Single Sample) block inherits the sample time from the driving block connected to the input port. Analog output is done synchronously, according to the following diagram.



At the first time step (T1), data is output to the selected hardware channels. The simulation does not continue until data is output to all channels.

Notes To use this block, you need both Data Acquisition Toolbox and Simulink software.

You can use the Analog Output (Single Sample) block only with devices that support single sample output. To send data using devices that do not support acquisition of a single sample (such as devices designed for sound and vibration), use the Analog Output block.

Some devices are not supported by the Simulink blocks in Data Acquisition Toolbox. To see if your device supports Simulink, refer to Supported Hardware.

Other Supported Features

- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Ports

Input

Data — Analog output to generate
double

Analog output to generate, specified as doubles. If using only one input port for all channels, provide a 1-by-N vector for a single scan on all N channels.

If using a port for each channel, provide a double value to each port. Multiple ports are named by channel names or device specified channel IDs.

Data Types: `double`

Parameters

Device — Device through which you want to output data
select available device

The device from which you want to generate data. The items in the list vary, depending on which devices you have connected to your system. Devices in the list are specified by adaptor/vendor name and unique device ID, followed by the model name of the device, for example, `ni Dev1 (USB-6255)`. The first available device is selected by default. A CompactDAQ chassis is shown as a single device; vendor name, chassis ID, and chassis model would be shown in the list, for example, `ni cDAQ2 (cDAQ-9172)`.

Channels — Device channel selection and configuration
options depend on device

Device channel selection and configuration table. The channel configuration table lists the hardware channels of your device, and lets you select and configure them. Specify which channels to acquire data from (by default all the channels are selected). The following parameters are specified for each selected channel:

Channel ID — Hardware channel ID specified by the device. The Channel ID column is read-only, and the parameters are defined when the device is selected.

Name — Channel name. By default the table displays any names provided by the hardware, but you can edit the names. For example, if the device is a sound card with two channels, you can name them `Left` and `Right`.

Module — Device ID the channel belongs to. The Module column is read-only. If a CompactDAQ chassis is selected, it shows the ID of the CompactDAQ module which the channel belongs to; otherwise the ID of the device.

Measurement Type — Measurement type of the channel. This block supports only voltage measurement types. (For other measurement types, use a DataAcquisition object in MATLAB.)

Output Range — Output ranges available for each channel supported by the hardware, defined when a device is selected.

Number of ports — Number of input data ports
1 for all channels | 1 per channel

Number of input data ports, specified as:

1 for all channels (default) — One input port on the block provides data for all channels. Provide data as a 1-by-N vector for N channels.

1 per channel — N input ports on the block, where N is equal to the number of selected channels. Provide data as a double value to each port. For naming, each output port uses the channel name if one was specified, otherwise the channel ID, for example, `ao1`.

Sample time — Block sample time
numeric value

Block sample time, specifies the sample time of the block during the simulation. This is the rate at which the block is executed during simulation. The default value is 1. For more information, see "What Is Sample Time?" (Simulink).

Version History

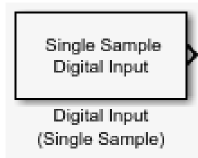
Introduced in R2016b

See Also

Blocks
Analog Input | Analog Output | Analog Input (Single Sample) | Digital Input (Single Sample) | Digital Output (Single Sample)

Digital Input (Single Sample)

Acquire single sample from multiple digital lines of data acquisition device



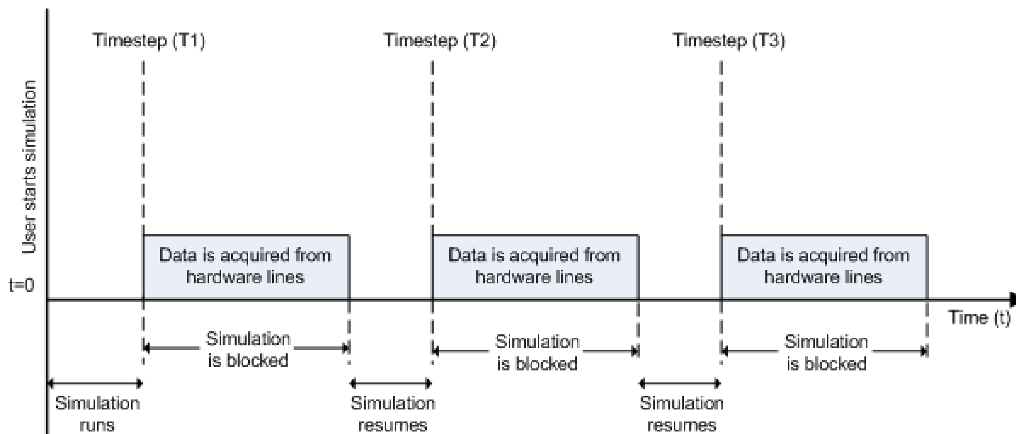
Libraries:
Data Acquisition Toolbox

Description

The Digital Input (Single Sample) block synchronously outputs the latest scan of data available from the digital lines selected at each simulation time step. It acquires unbuffered digital data, and delivers this as a vector of boolean values.

The block has no input ports. It has one or more output ports, depending on the option you choose in its dialog box.

The block inherits the sample time of the model. Digital input acquisition is done synchronously, according to the following diagram.



At the first time step (T1), data is acquired from the selected hardware lines. The simulation does not continue until data is read from all lines.

Note To use this block, you need both Data Acquisition Toolbox and Simulink software.

Some devices are not supported by the Simulink blocks in Data Acquisition Toolbox. To see if your device supports Simulink, refer to Supported Hardware.

Other Supported Features

- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.

- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the "Simulink" documentation.

Ports

Output

Data — Acquired digital input
boolean

Acquired digital input data, returned as booleans. If using only one output port for all lines, the output is a 1-by-N vector for N channels. If using a port for each line, each scan results in a single boolean on each port. Multiple ports are named by line names or device specified line IDs.

Data Types: `Boolean`

Timestamp — Timestamp of scan
double

Timestamp of scan, returned as a double. This port is available when you check the Output timestamp on page 17-0 parameter.

Data Types: `double`

Parameters

Device — Device from which you want to acquire data

The device from which you want to acquire data. The items in the list vary, depending on which devices you have connected to your system. Devices in the list are specified by adaptor or vendor name and unique device ID, followed by the model name of the device, for example, `ni Dev1 (USB-6255)`. The first available device is selected by default. A CompactDAQ chassis would be shown as a single device identified by vendor name, chassis ID, and chassis model; for example, `ni cDAQ2 (cDAQ-9172)`.

Lines — Device line selection and configuration
options depend on device

Line ID — ID of the hardware line (for example, port0/line0). This is automatically detected and filled in by the selected device, and is read-only.

Name — Hardware line name. This is automatically detected and filled in from the hardware, though you can edit the name.

Module — Device ID that the line belongs to. The Module column is read-only. If a CompactDAQ chassis is selected, it shows the ID of the CompactDAQ module which the line belongs to; otherwise the ID of the device.

Number of ports — Number of output data ports
1 for all lines | **1 per line**

Number of output data ports, specified as:

1 for all lines — The block has only one output port for all of the lines that are selected in the table. Acquired data is returned as a 1-by-N vector of boolean values, whose size is the number of lines.

1 per line — The block has one output port per selected line. Data is returned as a 1-by-1 boolean value on each port. The name of each output port is the name specified in the table for each line. If no name is provided, the name is the Line ID. For example, if line 2 of hardware port 3 is selected, and you did not specify a name in the line table, `port3/line2` appears in the block. Data size for each line is 1-by-1.

Sample time — Block execution rate
1 (default)

Specifies the sample time of the block during the simulation. This is the rate at which the block is executed during simulation. The default value is 1 (seconds). For more information, see “What Is Sample Time?” (Simulink).

Output timestamp — Add timestamp output port

Select this option to output the absolute timestamp of the scan. This option adds a new output port to the block. The data type of this port is double (datenum), which corresponds to a serial date number. You can convert the datenum into a datetime value with the `datetime` function.

Version History

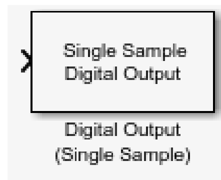
Introduced in R2016b

See Also

Blocks
Analog Input | Analog Output | Analog Input (Single Sample) | Analog Output (Single Sample) | Digital Output (Single Sample)

Digital Output (Single Sample)

Output single sample to multiple digital lines of data acquisition device



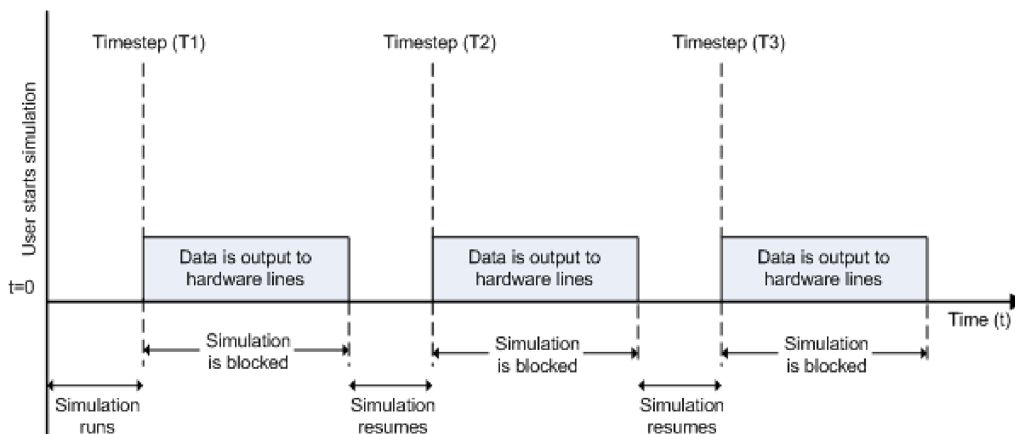
Libraries:
Data Acquisition Toolbox

Description

The Digital Output (Single Sample) block synchronously outputs the latest set of data to the hardware at each simulation time step. It outputs unbuffered digital data. Specify the output data as a vector of boolean values.

The block has no output ports. It can have one or more input ports, depending on the option you choose in its dialog box. The data type of the signal at the input port must be a boolean data type.

The Digital Output (Single Sample) block inherits the sample time from the driving block connected to the input port. Digital output is done synchronously, according to the following diagram.



At the first time step (T1), data is output to the selected hardware lines. The simulation does not continue until data is output to all lines.

Note To use this block, you need both Data Acquisition Toolbox and Simulink software.

Some devices are not supported by the Simulink blocks in Data Acquisition Toolbox. To see if your device supports Simulink, refer to Supported Hardware.

Other Supported Features

- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.

- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Ports

Input

Data — Generated digital output data
boolean

Generated digital output data, specified as booleans. If using only one input port for all lines, provide a 1-by-N vector of data. If using a port for each line, provide a single value on each port. Multiple ports are named by line names or device specified line IDs.

Data Types: `Boolean`

Parameters

Device — Device with which you want to generate data

The device from which you want to acquire data. The items in the list vary, depending on which devices you have connected to your system. Devices in the list are specified by adaptor or vendor name and unique device ID, followed by the model name of the device, for example, `ni Dev1 (USB-6255)`. The first available device is selected by default. A CompactDAQ chassis would be shown as a single device identified by vendor name, chassis ID, and chassis model; for example, `ni cDAQ2 (cDAQ-9172)`.

Lines — Device line selection and configuration
options depend on device

Line ID — ID of the hardware line (for example, port0/line0). This is automatically detected and filled in by the selected device, and is read-only.

Name — Hardware line name. This is automatically detected and filled in from the hardware, though you can edit the name.

Module — Device ID that the line belongs to. The Module column is read-only. If a CompactDAQ chassis is selected, it shows the ID of the CompactDAQ module which the line belongs to; otherwise the ID of the device.

Number of ports — Number of input data ports
1 for all lines | 1 per line

Number of input data ports, specified as:

1 for all lines — The block has only one input port for all of the lines that are selected in the table. Generated data is defined as a 1-by-N row vector of boolean values, whose size is the number of lines.

1 per line — The block has one input port per selected line. The name of each input port is the name specified in the table for each line. If no name is provided, the name is the Line ID. For example, if line 2 of hardware port 3 is selected, and you did not specify a name in the line table, `port3/line2` appears in the block. Data size for each line is 1-by-1.

Sample time — Block execution rate
1 (default)

Specifies the sample time of the block during the simulation. This is the rate at which the block is executed during simulation. The default value is 1 (seconds). For more information, see “What Is Sample Time?” (Simulink).

Version History

Introduced in R2016b

See Also

Blocks
Analog Input | Analog Output | Analog Input (Single Sample) | Analog Output (Single Sample) | Digital Input (Single Sample)

Troubleshooting Your Hardware

This appendix describes simple tests you can perform to troubleshoot your data acquisition hardware. The tests involve using software provided by the vendor, the operating system (sound cards), or Data Acquisition Toolbox software. The sections are as follows.

Troubleshooting Tips

In this section...

"Find Devices and Create a DataAcquisition Interface" on page A-2
"Is My Device Driver Supported?" on page A-3
"Cannot Find Hardware Vendor" on page A-3
"Cannot Detect My Device" on page A-4
"Why Doesn't My NI Hardware Work?" on page A-5
"Why Was My DataAcquisition Object Deleted?" on page A-5
"What Is a Reserved Hardware Error?" on page A-5
"Network Device Appears Unsupported" on page A-5
"Cannot Add Channels" on page A-6
"ADC Overrun Error with External Clock" on page A-6
"Cannot Add Clock Connection to PXI Devices" on page A-6
"Cannot Complete Long Foreground Acquisition" on page A-6
"Cannot Use PXI 4461 and 4462 Together" on page A-6
"Cannot Get Correct Scan Rate with Digilent Devices" on page A-7
"Cannot Simultaneously Acquire and Generate with myDAQ Devices" on page A-7
"Simultaneous Analog Input and Output Not Synchronized Correctly" on page A-7
"Counter Single Scan Returns NaN" on page A-7
"External Clock Will Not Trigger Scan" on page A-7
"Why Does My S/PDIF Device Time Out?" on page A-7
"MOTU Device Not Working Correctly" on page A-7

Find Devices and Create a DataAcquisition Interface

Identify the devices you can access:

```
dev  = daqlist
```

```
dev =

  9×5 table

     VendorID        DeviceID                     Description                                         Model
    _____________    ___________    ____________________________________________________    ___________________________________________

    "ni"             "Dev1"         "National Instruments(TM) USB-6211"                     "USB-6211"
    "ni"             "Dev2"         "National Instruments(TM) USB-6218"                     "USB-6218"
    "ni"             "Dev3"         "National Instruments(TM) USB-6255"                     "USB-6255"
    "ni"             "Dev4"         "National Instruments(TM) USB-6509"                     "USB-6509"
    "ni"             "PXI1Slot2"    "National Instruments(TM) PXIe-6341"                    "PXIe-6341"
    "directsound"    "Audio0"       "DirectSound Primary Sound Capture Driver"              "Primary Sound Capture Driver"
    "directsound"    "Audio1"       "DirectSound Headset Microphone (Plantronics BT600)"    "Headset Microphone (Plantronics BT600)"
    "directsound"    "Audio2"       "DirectSound Primary Sound Driver"                      "Primary Sound Driver"
    "directsound"    "Audio3"       "DirectSound Headset Earphone (Plantronics BT600)"      "Headset Earphone (Plantronics BT600)"
```

Create a DataAcquisition object for a specific vendor:

```
d = daq("ni")
```

For more information on the DataAcquisition interface, see "The DataAcquisition Object" on page 3-2.

To learn more about how to communicate with CompactDAQ devices, see “Interface Workflow” on page 4-2.

Is My Device Driver Supported?

For the supported device drivers for each vendor, see Hardware Support from Data Acquisition Toolbox, and click the link for your vendor. To see your installed driver version, in MATLAB type:

```
v = daqvendorlist

v = 

  5×4 table

        ID                 FullName              AdaptorVersion      DriverVersion  
    _____________    ____________________________    ______________    _________________

    "ni"             {'National Instruments(TM)'}    "4.1 (R2020a)"    "18.5.0 NI-DAQmx"
    "adi"            {'Analog Devices Inc.'     }    "4.1 (R2020a)"    "1.0"            
    "directsound"    {'DirectSound'             }    "4.1 (R2020a)"    "n/a"            
    "digilent"       {'Digilent Inc.'           }    "4.1 (R2020a)"    "3.7.20"         
    "mcc"            {'Not Operational'         }    "4.1 (R2020a)"    "unknown"        
```

If the `DriverVersion` field does not match the minimum requirements specified on the product page on the MathWorks website, use the Add-On Manager to update your support package.

If your driver is incompatible with Data Acquisition Toolbox, verify that your hardware is functioning properly before updating drivers. If your hardware is not functioning properly, you might be using unsupported drivers.

- **NI-DAQmx Drivers**

  Data Acquisition Toolbox software is compatible with only specific versions of the NI-DAQmx driver, and is not guaranteed to work with other versions. For a list of the NI-DAQmx driver versions that are compatible with Data Acquisition Toolbox software, see NI-DAQmx Support from Data Acquisition Toolbox.

  For the latest NI-DAQmx drivers, visit the NI™ website at `https://www.ni.com/`.

  To find your installed driver version in the NI **Measurement & Automation Explorer** use these steps:

  1. In the Windows taskbar, click **Start > NI MAX**.
  2. In the **Measurement & Automation Explorer** select **Help > System Information**.

- **Measurement Computing (MCC) Drivers**

  For a list of the MCC driver versions that are compatible with Data Acquisition Toolbox software, see Measurement Computing DAQ Support from Data Acquisition Toolbox.

Cannot Find Hardware Vendor

If you try to get vendor information using `daqvendorlist`, and receive one of the following errors:

- No vendors found:

  ```
  No data acquisition vendors available.
  ```

  Reinstall Data Acquisition Toolbox software and applicable support packages.

- Corrupted or missing toolbox components:

  ```
  Diagnostic Information from vendor: NI: The required MEX file to communicate
  with National Instruments hardware is not in the expected location.
  ```

  Reinstall Data Acquisition Toolbox software and applicable support packages.

  ```
  Diagnostic Information from vendor: NI: The required MEX file to communicate
  with National Instruments hardware exists but appears to be corrupt.
  ```

  Reinstall Data Acquisition Toolbox software and applicable support packages.
- The MCC entry indicates `'Not operational'`:

  There are various reasons that the adaptor can appear as `'Not operational'` related to MCC installation issues. Click the `'Not operational'` text link to get its error code. When you have the error code, see specific troubleshooting steps at Why Does the MCC Data Acquisition Adaptor Show as 'Not Operational'?

Cannot Detect My Device

If you try to find information using `daqlist` and do not see the expected device listed, refresh the toolbox and get a new device listing with the commands:

```
daqreset
daqlist
```

If you still do not see the expected devices, try the following.

- Make sure that your system is properly set up as described in “Set Up Your System for Device Detection” on page A-10.
- **NI Devices**
  - Go to the NI Measurement & Automation Explorer (NI MAX) and examine your devices to make sure your device is listed as available.
  - If you cannot see your device in NI MAX, check all device connections and power sources.
  - If you can see your device in NI MAX, run `daqreset` and `daqlist` in MATLAB again.
  - If you are using an Ethernet or WiFi network CompactDAQ chassis or FieldDAQ device, reserve the chassis or device in NI MAX first. Only one system can reserve a network device at any one time. For more information, see Why can't Data Acquisition Toolbox detect my NI DAQ devices connected through a cDAQ network chassis?.
- **Measurement Computing (MCC) Devices**
  - If your MCC device does not appear in the output of `daqlist`, close MATLAB and make sure that that the device is working correctly in Instacal. You must configure the device in Instacal before MATLAB can recognize it.
  - If your device is still not recognized in MATLAB, make sure your device is supported by checking Measurement Computing DAQ Support from Data Acquisition Toolbox.

Why Doesn’t My NI Hardware Work?

Use the **Test Panel** to troubleshoot your NI hardware. The **Test Panel** allows you to test each subsystem supported by your device, and is installed as part of the NI-DAQmx driver software. Right-click the device in the Measurement & Automation Explorer and choose **Test Panel**.

For example, to verify that the analog input subsystem on your PCIe-6363 device is operating, connect a known signal (similar to the signal produced by a function generator) to one or more channels, using a screw terminal panel.

If the **Test Panel** does not provide you with the expected results for the subsystem, and you are sure that your test setup is configured correctly, then the hardware is not performing correctly.

For NI hardware support, visit `https://www.ni.com/`.

Why Was My DataAcquisition Object Deleted?

An interface object might silently be deleted while executing a background operation. This could be caused by the object going out of scope at the end of a MATLAB function, before the background task completes. To avoid this, insert a `pause` after starting the operation.

What Is a Reserved Hardware Error?

If you receive the following error:

```
The hardware is reserved. If you are using it in another
object use the release function to unreserve the hardware. If you are using it in an
external program exit that program. Then try this operation again.
```

Identify the DataAcquisition that is currently not using this device but has reserved it, and release the associated hardware resources. If the device is reserved by:

- Another DataAcquisition in the current MATLAB session, do one of the following:
  - Use `release` to release the device from the other DataAcquisition.
  - Delete the other DataAcquisition object.
- Another DataAcquisition in a separate MATLAB session, do one of the following:
  - Use `release` to release the device from the other DataAcquisition.
  - Delete the other DataAcquisition object.
  - Exit the other MATLAB session.
- Another application, exit that application.

If these measures do not work, reset the device from NI MAX.

Network Device Appears Unsupported

- If your network device appears as unsupported or unavailable, make sure that the device is connected and reserved in the NI Measurement and Automation Explorer. Use `daqreset` to reset devices settings.
- If you see this timeout error when communicating with a network device:

  ```
  Network timeout error while communicating with device 'cDAQ9188-1595393Mod4'
  ```

reconnect the device in the NI Measurement and Automation Explorer and execute `daqreset` in MATLAB to reset the devices settings.

- **Note** Your network device might also appear as unsupported in the device information if it is reserved or disconnected.

Cannot Add Channels

- **Missing Subsystem**

  When attempting to add a channel generates the error:

  ```
  The requested subsystem does not exist on this device
  ```

  A possible cause could be:

  - Using an output device to add input channels, or an input device to add output channels.
  - Using an unsupported device. See “Data Acquisition Toolbox Supported Hardware”.
- **Counter Channels**

  If you are using an NI 9402 with a counter/timer subsystem in a cDAQ-9172 chassis, plug the module only into slot 5 or 6. If you plug the module into any other slot, it will not show any counter/timer subsystem.

ADC Overrun Error with External Clock

If you see this error when you synchronize acquisition using an external clock,

```
ADC Overrun Error: If you are using an external clock, make sure that
the clock frequency matches scan rate.
```

- Check your external clock for the presence of noise or glitches.
- Check the frequency of your external clock. Make sure that it matches the DataAcquisition `Rate` property value.

Cannot Add Clock Connection to PXI Devices

When you try to synchronize operations using a PXI 447x series device, you see this error:

```
"DSA device 'PXI1Slot2' does not support sample clock synchronization. Check device's user manual.
```

NI DSA devices like the PXI 447x, do not support sample clock synchronization. You cannot synchronize these devices in the DataAcquisition interface using `addclock`.

Cannot Complete Long Foreground Acquisition

When you try to acquire data in the foreground for a long period, you might get an out-of-memory error. Switch to background acquisitions and process data as it is received or save the data to a file to mitigate this issue.

Cannot Use PXI 4461 and 4462 Together

You cannot use a PXI 4461 and a 4462 together for synchronization, when the PXI 4461 is in the timing slot of the chassis.

Cannot Get Correct Scan Rate with Digilent Devices

The scan rate of a Digilent device can be limited by the hardware buffer size. See "Digilent Analog Discovery Hardware Limitations" on page B-4 for more information on maximum and minimum allowable rates.

Cannot Simultaneously Acquire and Generate with myDAQ Devices

You cannot acquire and generate synchronous data using myDAQ devices because they do not share a hardware clock. If you have both input and output channels in a DataAcquisition, when you start it you achieve near-simultaneous acquisition and generation. See "Automatic Synchronization" on page 13-4 for more information.

Simultaneous Analog Input and Output Not Synchronized Correctly

To simultaneously acquire and generate synchronized analog signals in the same DataAcquisition, try using an external trigger.

Counter Single Scan Returns NaN

An input single scan on counter input channels might return a `NaN`. If this occurs:

- Make sure that the signal voltage complies with TTL voltage specifications.
- Make sure that the channel frequency is within the specified frequency range.

External Clock Will Not Trigger Scan

Adding an external clock to your DataAcquisition might not trigger a scan unless you set the `Rate` property value to match the expected external clock frequency.

Why Does My S/PDIF Device Time Out?

S/PDIF audio ports appear in the device list even when you have no devices plugged in.

- If you add this device (port) to your DataAcquisition and you have no device plugged into the port, the operation times out.
- If you have a device plugged into the S/PDIF port, you may need to match the DataAcquisition rate to the device scan rate to get accurate readings. Refer to your device documentation for information.

MOTU Device Not Working Correctly

MOTU devices Ultralight-mk3 and Traveler-mk3 might not work with DirectSound and Data Acquisition Toolbox. If you have these devices, specify the device to use stereo pairs:

- In your MOTU Audio Console, select the **Use Stereo Pairs for Windows Audio** check box.
- Specify the required sample rate in the **Sample Rate** field.

See Also

Functions
`daqvendorlist` | `daqlist` | `daqreset`

Related Examples

- “Set Up Your System for Device Detection” on page A-10
- “Limitations by Vendor” on page B-2

Contact MathWorks for Technical Support

If you need support from MathWorks, visit the support website at `https://www.mathworks.com/support/contact_us.html`.

Before contacting MathWorks, you should run the `daqsupport` function in MATLAB. This function returns diagnostic information such as:

- The versions of MathWorks products you are using
- Your MATLAB software path
- The characteristics of your hardware

The output from `daqsupport` is automatically saved to a text file, which you can use to help troubleshoot your problem or send to MathWorks technical support if requested.

Set Up Your System for Device Detection

In this section...
“NI Devices” on page A-10
“DirectSound Audio Devices” on page A-10
“Measurement Computing (MCC) Devices” on page A-11
“Digilent Analog Discovery Devices” on page A-11
“Analog Devices ADALM1000 Devices” on page A-12

For interfacing with your device from MATLAB and Data Acquisition Toolbox, you must have the required support package, drivers, and sometimes third-party apps installed. See the following sections for your particular vendor to set up your system or to troubleshoot device discovery issues.

NI Devices

- Make sure the supported NI-DAQmx version is installed. You can install the Data Acquisition Toolbox Support Package for National Instruments NI-DAQmx Devices using Add-On Explorer.
- Reboot the computer after the NI-DAQmx support package installation completes.
- If the support package fails to install or if you are using a MATLAB release that does not have a support package for NI-DAQmx, you can install NI-DAQmx separately from MATLAB as described in Why does "Data Acquisition Toolbox Support Package for National Instruments NI-DAQmx Devices" fail to install?
- Confirm that the `"ni"` vendor is listed as operational by `daqvendorlist`.
- Confirm that you are using the supported NI-DAQmx driver version.
  - Data Acquisition Toolbox software is compatible with only specific versions of the NI-DAQmx driver, and is not guaranteed to work with other versions. For a list of the NI-DAQmx driver versions that are compatible with Data Acquisition Toolbox, see the driver version table in the “Required Products and Hardware section” in NI-DAQmx Support from Data Acquisition Toolbox.
  - You can find the installed NI-DAQmx driver version with `daqvendorlist` or in the NI **Measurement & Automation Explorer** (NI MAX).
- If an unsupported NI-DAQmx version is currently installed on the computer, you can:
  1. Uninstall it from NI Package Manager from Windows Control Panel > Programs > Uninstall a program.
  2. Install the supported NI-DAQmx version as described above.
- If you are using an Ethernet or WiFi network CompactDAQ chassis or FieldDAQ device, first reserve the chassis or device in NI MAX. Only one system can reserve a network device at any one time. For more information, see Why can't Data Acquisition Toolbox detect my NI DAQ devices connected through a cDAQ network chassis?

DirectSound Audio Devices

- Make sure the Data Acquisition Toolbox Support Package for Windows Sound Cards is installed. You can install it with Add-On Explorer.

- Some audio interface devices require the installation of DirectSound compatible drivers provided by the device vendor.
- Audio input devices might require connecting a microphone to be detected correctly.
- Verify that your audio device is detected and operating as expected when using other applications that use DirectSound, such as Audacity.

Measurement Computing (MCC) Devices

- Make sure the Data Acquisition Toolbox Support Package for Measurement Computing Hardware is installed. You can install it with Add-On Explorer.
- Confirm that the `"mcc"` vendor is listed as operational by `daqvendorlist`.
- Confirm that you are using the supported MCC DAQ driver (Instacal) version.
  - Data Acquisition Toolbox software is compatible with only specific versions of the MCC DAQ driver (Instacal), and is not guaranteed to work with other versions. For a list of the driver versions that are compatible with Data Acquisition Toolbox, see the driver version table in the “Required Products and Hardware” section of Measurement Computing DAQ Support from Data Acquisition Toolbox.
  - You can find the installed Instacal version with the Instacal app or with the MATLAB function `daqvendorlist`.
- If an unsupported Instacal version is currently installed on the computer, you can:
  1. Uninstall it from Windows Control Panel > Programs > Uninstall a program.
  2. Uninstall the Data Acquisition Toolbox Support Package for Measurement Computing Hardware using Add-On Manager.
  3. Install the Data Acquisition Toolbox Support Package for Measurement Computing Hardware using Add-On Explorer.
- Open the Instacal app and configure your device. The device must be configured in Instacal before MATLAB can recognize it.

Digilent Analog Discovery Devices

- Make sure the Data Acquisition Toolbox Support Package for Digilent Analog Discovery™ Hardware is installed. You can install it with Add-On Explorer.
- Confirm that the `"digilent"` vendor is listed as operational by `daqvendorlist`.
- Confirm that you are using the supported Digilent WaveForms driver version.
  - Data Acquisition Toolbox software is compatible with only specific versions of the Digilent WaveForms driver, and is not guaranteed to work with other versions. For a list of the driver versions that are compatible with Data Acquisition Toolbox, see the driver version table in the “Required Products and Hardware” section in Digilent Analog Discovery Support from MATLAB.
  - You can find the installed WaveForms version with `daqvendorlist` or in Digilent Waveforms.
- If an unsupported Waveforms version is currently installed on the computer, you can:
  1. Uninstall it from Windows Control Panel > Programs > Uninstall a program.
  2. Uninstall the Data Acquisition Toolbox Support Package for Digilent Analog Discovery Hardware using Add-On Manager

3. Install the Data Acquisition Toolbox Support Package for Digilent Analog Discovery Hardware using Add-On Explorer.

- Confirm that the device is detected by Digilent WaveForms.

Analog Devices ADALM1000 Devices

- Make sure the Data Acquisition Toolbox Support Package for Analog Devices ADALM1000 Hardware is installed. You can install it with Add-On Explorer.
- Plug in your device, and run `daqreset`.
- Run `daqvendorlist`, and confirm that the `"adi"` vendor is listed as operational.

See Also

Functions
`daqvendorlist` | `daqlist` | `daqreset`

Related Examples

- "Troubleshooting Tips" on page A-2
- "Limitations by Vendor" on page B-2

Hardware Limitations by Vendor

This topic describes limitations of using hardware in the Data Acquisition Toolbox based on limitations places by the hardware vendor:

Limitations by Vendor

For some vendors, there are limitations in the toolbox support for their functionality. See the following topics for each vendor.

- “Digilent Analog Discovery Hardware Limitations” on page B-4
- “Measurement Computing Hardware Limitations” on page B-5
- “National Instruments Hardware Limitations” on page B-3
- “Analog Devices ADALM1000 Limitations” on page B-6

See Also

More About

- “Set Up Your System for Device Detection” on page A-10
- “Troubleshooting Tips” on page A-2

National Instruments Hardware Limitations

- Required hardware drivers and any other device-specific software is described in the documentation provided by your hardware vendor. For more information, see NI-DAQmx Support from Data Acquisition Toolbox.
- You can use PXI_STAR with the `addtrigger` and `addclock` functions. All supported PXI modules automatically use the reference Clock PXI_CLK10.
- Objects created for National Instruments devices, and used with the NI-DAQmx adaptor, have the following behavior when you attempt single scan (on-demand) operations:

  - The first time the command is used with the object, the corresponding subsystem of the device is reserved by the DataAcquisition object in MATLAB.
  - If you then try to access that subsystem in a different MATLAB DataAcquisition, or any other application from the same computer, you might receive an error message informing you that the subsystem is reserved. Use `release` to unreserve the subsystem from the other DataAcquisition.
- You cannot acquire and generate synchronous data using myDAQ devices because they do not share a hardware clock. If you have both input and output channels in a DataAcquisition, when you start it you achieve near-simultaneous acquisition and generation. See “Automatic Synchronization” on page 13-4 for more information.
- NI USB devices that have their own power supply can shut down if the driver does not set the USB power correctly.
- Data Acquisition Toolbox does not support direct access to device onboard clocks for clocked sampling when using only digital input/output channels with a DataAcquisition object. For workarounds and information on clocked digital sampling, see the following topics:

  - “Acquire Digital Data Using a Shared Clock” on page 9-5
  - “Acquire Digital Data Using an External Clock” on page 9-6
  - “Acquire Digital Data Using a Counter Output Channel as External Clock” on page 9-8

Digilent Analog Discovery Hardware Limitations

- You cannot use multiple Digilent devices in the same DataAcquisition interface. If you need to use multiple devices, add one device per DataAcquisition and start them sequentially.
- Digilent devices limit the minimum and maximum allowable rate of sampling based on channel types:

  - Analog input only: 0.1 – 1,000,000
  - Analog output only: 4,096 – 1,000,000
  - Input and output: 8,192 – 300,000

  Data Acquisition Toolbox conforms to the Digilent Player Mode for the Arbitrary Waveform Generator.
- Digilent devices are not supported in the **Analog Input Recorder** or **Analog Output Generator** apps.
- You cannot use background operations with Digilent devices. You can only perform foreground operations.
- You cannot perform synchronous and triggered operations using a Digilent device.
- You cannot access the digital input and output capabilities of a Digilent device.

Measurement Computing Hardware Limitations

- For your Measurement Computing device to appear in the output of the `daqlist` function, you must first detect it in InstaCal.
- MCC devices are not supported by the Simulink blocks of the Data Acquisition Toolbox block library.
- External clocking and triggering of MCC devices is not supported.
- Support for MCC devices is limited to analog output voltage and analog input voltage measurements.
- MCC `DEMO-BOARD` devices simulated in InstaCal are not supported.

Analog Devices ADALM1000 Limitations

The following restrictions and limitations apply when programming the Analog Devices ADALM1000. Some are restrictions of the hardware, some are restrictions imposed by Data Acquisition Toolbox.

- For clocked (non-single-scan) operations, only the following are supported:
  - Foreground — Finite source, measurement, and simultaneous source and measurement
  - Background — Continuous measurement (No source or any finite operations)
- You cannot simultaneously source and measure voltage on the same channel, nor simultaneously source and measure current on the same channel.
- You cannot measure current without generating an output voltage.
- You cannot execute a single-scan operation that performs both source and measurement simultaneously.
- You cannot add channels from multiple ADALM1000 modules in the same DataAcquisition object. To recover from attempting this, you might need to execute `daqreset`.
- You cannot use AC coupling, nor differential terminal configurations.
- You cannot use triggers or digital pins.
- When specified output ranges are exceeded, the device might reset itself. Any measurements taken during this time might be unreliable until the reset is complete.
- Analog Devices ADALM1000 devices are not supported in the **Analog Output Generator** app.

Examples by Vendor

See the following topics for examples of each hardware vendor.

- "Analog Devices ADALM1000 Examples" on page B-8
- "Digilent Analog Discovery Hardware Examples" on page B-9
- "Measurement Computing Hardware Examples" on page B-10
- "National Instruments Hardware Examples" on page B-11
- "Windows Sound Card Examples" on page B-13

Analog Devices ADALM1000 Examples

"Characterize an LED with ADALM1000" on page 18-139

"Estimate the Transfer Function of a Circuit with ADALM1000" on page 18-143

See Also

More About

- "Digilent Analog Discovery Hardware Examples" on page B-9
- "Measurement Computing Hardware Examples" on page B-10
- "National Instruments Hardware Examples" on page B-11
- "Windows Sound Card Examples" on page B-13

Digilent Analog Discovery Hardware Examples

"Getting Started Acquiring Data with Digilent Analog Discovery" on page 18-68

"Getting Started Generating Data with Digilent Analog Discovery" on page 18-71

"Acquiring and Generating Data at the Same Time with Digilent Analog Discovery" on page 18-73

"Generate Standard Periodic Waveforms Using Digilent Analog Discovery" on page 18-76

"Generate Arbitrary Periodic Waveforms Using Digilent Analog Discovery" on page 18-79

See Also

More About

- "Analog Devices ADALM1000 Examples" on page B-8
- "Measurement Computing Hardware Examples" on page B-10
- "National Instruments Hardware Examples" on page B-11
- "Windows Sound Card Examples" on page B-13

Measurement Computing Hardware Examples

“Getting Started with MCC Devices” on page 18-7

“Discover MCC Devices” on page 18-12

“Acquire Data from Multiple Channels using an MCC Device” on page 18-22

See Also

More About

- “Analog Devices ADALM1000 Examples” on page B-8
- “Digilent Analog Discovery Hardware Examples” on page B-9
- “National Instruments Hardware Examples” on page B-11
- “Windows Sound Card Examples” on page B-13

National Instruments Hardware Examples

Getting Started and Device Discovery

"Getting Started with NI Devices" on page 18-3

"Discover NI Devices" on page 18-10

Analog Input and Output

"Acquire Data Using NI Devices" on page 18-14

"Acquire Continuous and Background Data Using NI Devices" on page 18-18

"Acquire Data from an Accelerometer" on page 18-26

"Measure Strain Using an Analog Bridge Sensor" on page 18-28

"Acquire Temperature Data From a Thermocouple" on page 18-31

"Acquire Temperature Data From an RTD" on page 18-34

"Acquire and Analyze Sound Pressure Data From an IEPE Microphone" on page 18-37

"Acquire and Analyze Noisy Clock Signals" on page 18-41

"Generate Voltage Signals Using NI Devices" on page 18-51

"Generate Signals on NI Devices That Output Current" on page 18-54

"Generate Continuous and Background Signals Using NI Devices" on page 18-57

"Simultaneously Acquire Data and Generate Signals" on page 18-60

"Log Analog Input Data to a File Using NI Devices" on page 18-64

"Capture Data with Software-Analog Triggering" on page 18-92

"Create an App for Analog Triggered Data Acquisition" on page 18-150

"Create an App for Live Data Acquisition" on page 18-157

Digital Input and Output

"Control Stepper Motor Using Digital Outputs" on page 18-115

"Communicate with I2C Devices and Analyze Bus Signals Using Digital IO" on page 18-118

Counters and Timers

"Count Pulses on a Digital Signal Using NI Devices" on page 18-101

"Measure Frequency Using NI Devices" on page 18-104

"Measure Pulse Width Using NI Devices" on page 18-106

"Generate Pulse Width Modulated Signals Using NI Devices" on page 18-108

"Measure Angular Position with an Incremental Rotary Encoder" on page 18-110

Simultaneous and Synchronized Operations

"Synchronize NI PCI Devices Using RTSI" on page 18-125

"Start a Multi-Trigger Acquisition on an External Event" on page 18-128

"Acquire Data from Two Devices at Different Rates" on page 18-136

Simulink Data Acquisition

"Perform Live Acquisition, Signal Processing, and Generation" on page 18-130

"Perform Spectral Analysis on Live Data" on page 18-132

See Also

More About

- "Analog Devices ADALM1000 Examples" on page B-8
- "Digilent Analog Discovery Hardware Examples" on page B-9
- "Measurement Computing Hardware Examples" on page B-10
- "Windows Sound Card Examples" on page B-13

Windows Sound Card Examples

"Acquire Continuous Audio Data" on page 18-83

"Generate Audio Signals" on page 18-86

"Generating Multichannel Audio" on page 18-88

"Analog Triggered Data Acquisition Using Stateflow Charts" on page 18-153

See Also

More About

- "Analog Devices ADALM1000 Examples" on page B-8
- "Digilent Analog Discovery Hardware Examples" on page B-9
- "Measurement Computing Hardware Examples" on page B-10
- "National Instruments Hardware Examples" on page B-11

Data Acquisition Toolbox Examples

- “Getting Started with NI Devices” on page 18-3
- “Getting Started with MCC Devices” on page 18-7
- “Discover NI Devices” on page 18-10
- “Discover MCC Devices” on page 18-12
- “Acquire Data Using NI Devices” on page 18-14
- “Acquire Continuous and Background Data Using NI Devices” on page 18-18
- “Acquire Data from Multiple Channels using an MCC Device” on page 18-22
- “Acquire Data from an Accelerometer” on page 18-26
- “Measure Strain Using an Analog Bridge Sensor” on page 18-28
- “Acquire Temperature Data From a Thermocouple” on page 18-31
- “Acquire Temperature Data From an RTD” on page 18-34
- “Acquire and Analyze Sound Pressure Data From an IEPE Microphone” on page 18-37
- “Acquire and Analyze Noisy Clock Signals” on page 18-41
- “Generate Voltage Signals Using NI Devices” on page 18-51
- “Generate Signals on NI Devices That Output Current” on page 18-54
- “Generate Continuous and Background Signals Using NI Devices” on page 18-57
- “Simultaneously Acquire Data and Generate Signals” on page 18-60
- “Log Analog Input Data to a File Using NI Devices” on page 18-64
- “Getting Started Acquiring Data with Digilent Analog Discovery” on page 18-68
- “Getting Started Generating Data with Digilent Analog Discovery” on page 18-71
- “Acquiring and Generating Data at the Same Time with Digilent Analog Discovery” on page 18-73
- “Generate Standard Periodic Waveforms Using Digilent Analog Discovery” on page 18-76
- “Generate Arbitrary Periodic Waveforms Using Digilent Analog Discovery” on page 18-79
- “Acquire Continuous Audio Data” on page 18-83
- “Generate Audio Signals” on page 18-86
- “Generating Multichannel Audio” on page 18-88
- “Capture Data with Software-Analog Triggering” on page 18-92
- “Count Pulses on a Digital Signal Using NI Devices” on page 18-101
- “Measure Frequency Using NI Devices” on page 18-104
- “Measure Pulse Width Using NI Devices” on page 18-106
- “Generate Pulse Width Modulated Signals Using NI Devices” on page 18-108
- “Measure Angular Position with an Incremental Rotary Encoder” on page 18-110
- “Control Stepper Motor Using Digital Outputs” on page 18-115
- “Communicate with I2C Devices and Analyze Bus Signals Using Digital IO” on page 18-118
- “Synchronize NI PCI Devices Using RTSI” on page 18-125

- “Start a Multi-Trigger Acquisition on an External Event” on page 18-128
- “Perform Live Acquisition, Signal Processing, and Generation” on page 18-130
- “Perform Spectral Analysis on Live Data” on page 18-132
- “Acquire Data from Two Devices at Different Rates” on page 18-136
- “Characterize an LED with ADALM1000” on page 18-139
- “Estimate the Transfer Function of a Circuit with ADALM1000” on page 18-143
- “Create an App for Analog Triggered Data Acquisition” on page 18-150
- “Analog Triggered Data Acquisition Using Stateflow Charts” on page 18-153
- “Create an App for Live Data Acquisition” on page 18-157
- “Acquire Data Using NI FieldDAQ Device” on page 18-159
- “Create an Echometer Using Audio Measurements” on page 18-162
- “Get Started Reading a TDMS-File” on page 18-172
- “Read Multiple TDMS-Files into MATLAB” on page 18-175
- “Read a Large TDMS-File into MATLAB” on page 18-178
- “Get Started Writing a TDMS-File” on page 18-180
- “Write Timetable Data to TDMS-file” on page 18-184
- “Write Metadata to TDMS-File” on page 18-187
- “Merge Multiple TDMS-Files” on page 18-190
- “Analyze TDMS-Files Using Tall Tables” on page 18-192
- “Impulse Response Measurement Using a NI USB-4431 Device” on page 18-197

Getting Started with NI Devices

This example shows how to get started with National Instruments devices from the command line.

Discover Available Devices

Discover devices connected to your system using `daqlist`. To learn more about an individual device, access the entry in the device table.

```
d = daqlist;
d(1, :)
```

```
ans =

  1×5 table

    VendorID     DeviceID              Description               Model              DeviceInf
    ________    ___________    ______________________________    _________    ____________________

      "ni"      "cDAQ1Mod1"    "National Instruments NI 9205"    "NI 9205"    [1×1 daq.ni.Compac
```

```
d{1, "DeviceInfo"}
```

```
ans =

ni: National Instruments NI 9205 (Device ID: 'cDAQ1Mod1')
   Analog input supports:
      4 ranges supported
      Rates from 0.6 to 250000.0 scans/sec
      32 channels ('ai0' - 'ai31')
      'Voltage' measurement type

This module is in slot 1 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Create a DataAcquisition

The `daq` command creates a DataAcquisition object. The DataAcquisition contains information describing hardware, scan rate, and other properties associated with the acquisition.

```
dq = daq("ni")
```

```
dq =

DataAcquisition using National Instruments hardware:

                     Running: 0
                        Rate: 1000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
```

```
                   RateLimit: []

Show channels
Show properties and methods
```

Add an Analog Input Channel

The `addinput` command attaches an input channel to the DataAcquisition.

```
ch = addinput(dq,"cDAQ1Mod1", "ai0","Voltage")
```

```
ch =

    Index    Type       Device       Channel     Measurement Type          Range             Nar
    _____    ____    ___________    _______    _________________    __________________    ________

      1      "ai"    "cDAQ1Mod1"     "ai0"     "Voltage (Diff)"    "-10 to +10 Volts"    "cDAQ1Mc
```

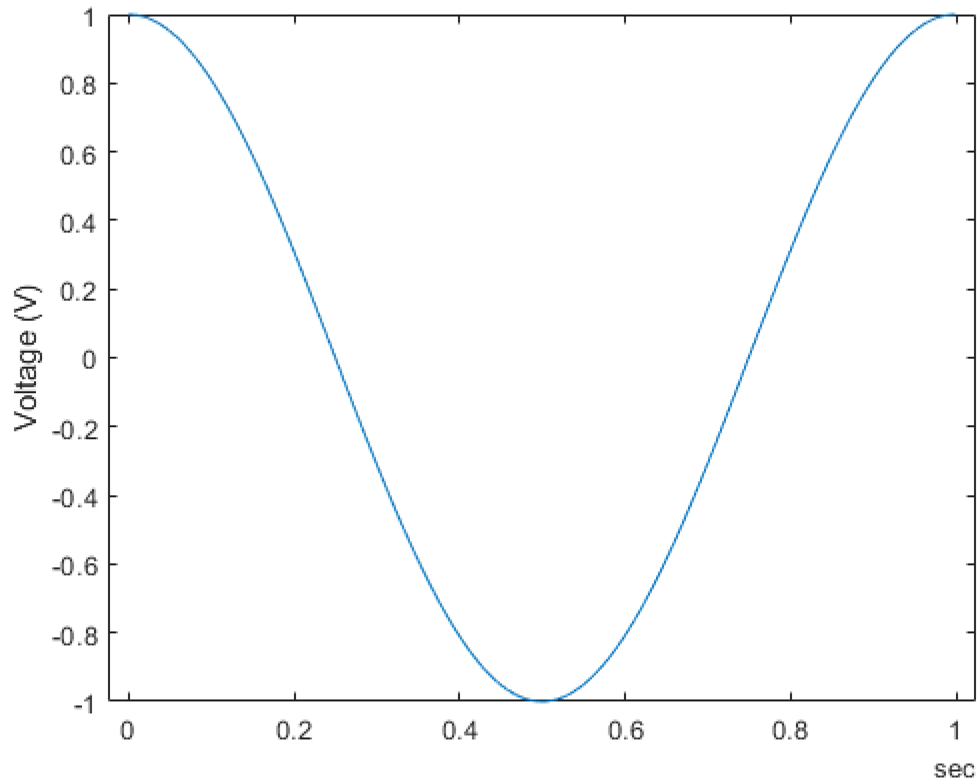
Acquire Timestamped Data

The `read` command starts the acquisition and returns the results as a timetable.

```
data = read(dq, seconds(1));
```

Plot Data

```
plot(data.Time, data.cDAQ1Mod1_ai0);
ylabel("Voltage (V)");
```



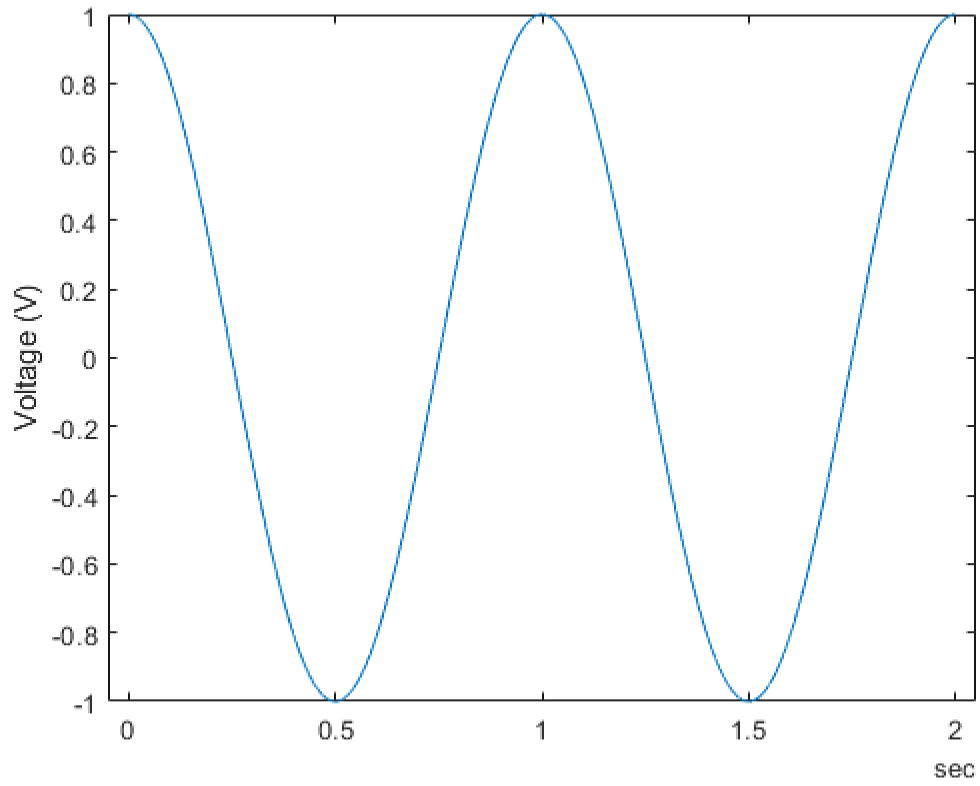
Change Default Properties of the Acquisition

By default, run at a scan rate of 1000 scans per second. To acquire at a higher rate, change the `Rate` property.

```
dq.Rate = 5000;
```

Run the acquisition and plot the acquired data:

```
[data, startTime] = read(dq, seconds(2));
plot(data.Time, data.cDAQ1Mod1_ai0);
ylabel("Voltage (V)");
```


Voltage (V)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
0.5
1
1.5
2
sec

Getting Started with MCC Devices

This example shows how to get started with MCC devices from the command line.

Discover Available Devices

Discover devices connected to your system using `daqlist`. To learn more about an individual device, access the entry in the device table.

```
d = daqlist("mcc");
d(1, :)
```

```
ans =

  1×4 table

    DeviceID                   Description                          Model              Devi
    ________    __________________________________________    __________________    ___________

    "Board0"    "Measurement Computing Corp. USB-1608FS-Plus"    "USB-1608FS-Plus"    [1×1 daq.s
```

Create a DataAcquisition

The `daq` function creates a DataAcquisition object. The DataAcquisition contains information describing hardware, scan rate, and other properties associated with the acquisition.

```
dq = daq("mcc")
```

```
dq = 

DataAcquisition using Measurement Computing Corp. hardware:

                     Running: 0
                        Rate: 1000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Add an Analog Input Channel

The `addinput` function attaches an input channel to the DataAcquisition. You can add more than one channel to a DataAcquisition. This example uses one input channel, Ai0, which is connected to a function generator channel outputting a 10 Hz sine wave.

```
addinput(dq, "Board0", "Ai0", "Voltage");
dq
```

```
dq = 
```

```
DataAcquisition using Measurement Computing Corp. hardware:

                     Running: 0
                        Rate: 1000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: [0.1000 100000]

Show channels
Show properties and methods
```

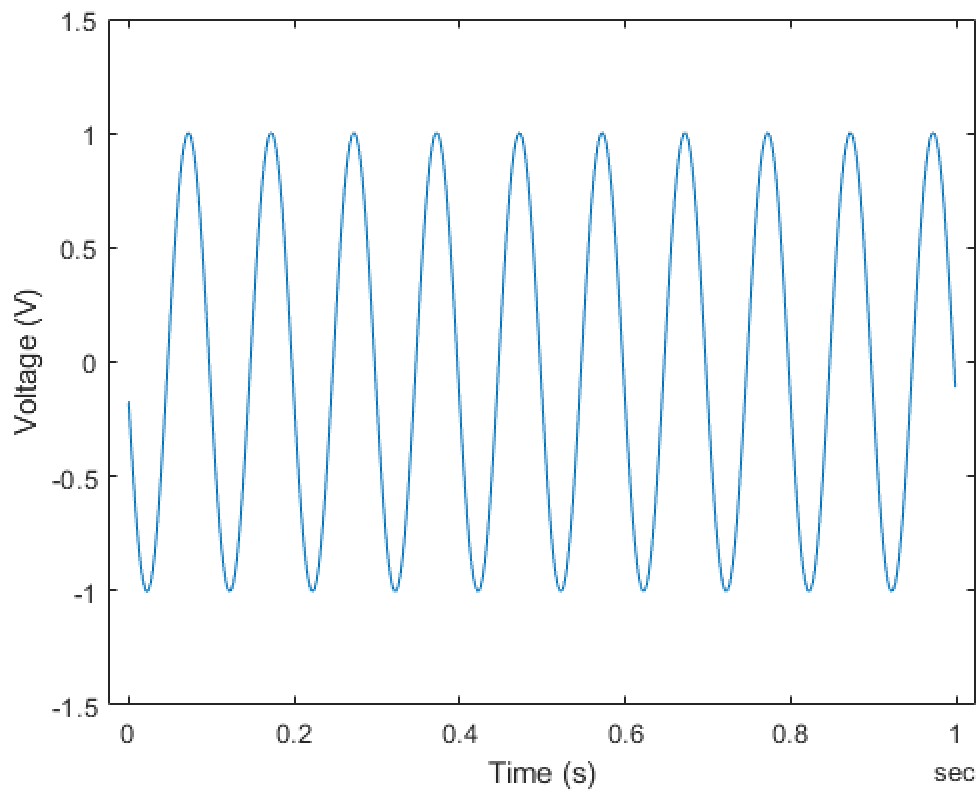
Acquire Timestamped Data

The `read` function starts the acquisition and returns the results as a timetable.

```
[data, startTime] = read(dq, seconds(1));
```

Plot Acquired Data

```
plot(data.Time, data.Board0_Ai0);
xlabel("Time (s)");
ylabel("Voltage (V)");
```



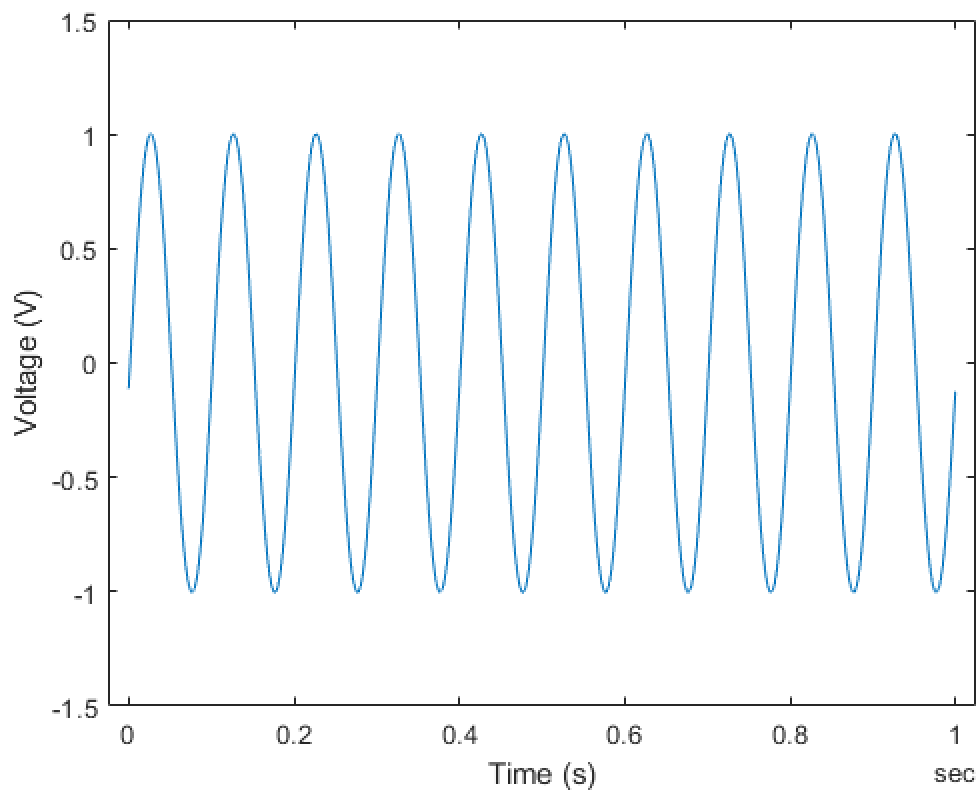
Change Default Properties of the Acquisition

By default, acquisitions run for one second at 1000 scans per second. To acquire at a different rate, change the `Rate` property.

```
dq.Rate = 5000;
```

Run the acquisition and plot the acquired data:

```
[data, startTime] = read(dq, seconds(1));
plot(data.Time, data.Board0_Ai0);
xlabel("Time (s)");
ylabel("Voltage (V)");
```



Discover NI Devices

This example shows how to discover National Instruments devices visible to MATLAB® and get information about channel and measurement types available in those devices.

Display a List of Available Devices

Use `daqlist` to display a list of devices available to your machine and MATLAB.

```
d = daqlist("ni")
```

```
d =

  12×4 table

      DeviceID               Description                      Model              DeviceInfo
    ___________    __________________________________    ______________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"         [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"       [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"     [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"       [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"       [1×1 daq.DeviceInfo]
```

Get Details About a Device

The `daqlist` command shows you the overview of devices available. To obtain more information about a particular device, view the "DeviceInfo" table cell for it.

```
deviceInfo = d{1, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9205 (Device ID: 'cDAQ1Mod1')
   Analog input supports:
      4 ranges supported
      Rates from 0.6 to 250000.0 scans/sec
      32 channels ('ai0' - 'ai31')
      'Voltage' measurement type

This module is in slot 1 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Dynamic Hardware Discovery

When the hardware configuration changes (for example, a new CompactDAQ module is plugged into the chassis), use `daqreset` followed by `daqlist` to observe the changes.

Discover MCC Devices

This example shows how to discover devices visible to MATLAB and get information about channel and measurement types available in those devices.

Display a List of Available Vendors

Discover available vendors for your system using `daqvendorlist`.

```
v = daqvendorlist
```

```
v =

  1×4 table

     ID               FullName                AdaptorVersion    DriverVersion
    _____    _______________________________    ______________    _____________

    "mcc"    {'Measurement Computing Corp.'}    "4.1 (R2020a)"      "6.60.0"
```

Display a List of Available Devices

Discover devices connected to your system using `daqlist`.

```
d = daqlist("mcc")
```

```
d =

  1×4 table

    DeviceID                     Description                          Model              Devi
    ________    ____________________________________________    _________________    __________

    "Board0"    "Measurement Computing Corp. USB-1608FS-Plus"    "USB-1608FS-Plus"    [1×1 daq.sc
```

Get Details About a Device

The `daqlist` command shows you the overview of devices available. You can find additional device details by reviewing the `DeviceInfo` field of the table.

```
deviceInfo = d{1, "DeviceInfo"}
```

```
deviceInfo =

mcc: Measurement Computing Corp. USB-1608FS-Plus (Device ID: 'Board0')
   Analog input supports:
      4 ranges supported
      Rates from 0.1 to 100000.0 scans/sec
      8 channels ('Ai0' - 'Ai7')
      'Voltage' measurement type
```

Display Subsystems of a Device

Use the `Subsystems` property to find all the subsystem information. To display all details about the first subsystem including the channel, type:

```
deviceInfo.Subsystems
```

```
ans =

Analog input supports:
   4 ranges supported
   Rates from 0.1 to 100000.0 scans/sec
   8 channels ('Ai0' - 'Ai7')
   'Voltage' measurement type
```

Dynamic Hardware Discovery

When you change your hardware configuration (for example, plug in a new USB device), first detect the device in InstaCal. Then, use the `daqreset` command to refresh Data Acquisition toolbox before using `daqlist` to discover the changes.

Acquire Data Using NI Devices

This example shows how to acquire data from a National Instruments device.

Discover Analog Input Devices

To discover a device that supports input measurements, access the device in the table returned by the `daqlist` command. This example uses an NI 9201 module in a National Instruments® CompactDAQ Chassis NI cDAQ-9178. This is an 8 channel analog input device and is module 4 in the chassis.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID               Description                    Model              DeviceInfo
    ___________    __________________________________    ______________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"         [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"       [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"     [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"       [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"       [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{4, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9201 (Device ID: 'cDAQ1Mod4')
   Analog input supports:
      -10 to +10 Volts range
      Rates from 0.6 to 500000.0 scans/sec
      8 channels ('ai0' - 'ai7')
      'Voltage' measurement type

This module is in slot 4 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Create a DataAcquisition and Add Analog Input Channels

Create a DataAcquisition, set the `Rate` property (the default is 1000 scans per second), and add analog input channels using `addinput`.

```
dq = daq("ni");
dq.Rate = 8000;
```

```
addinput(dq, "cDAQ1Mod4", "ai0", "Voltage");
addinput(dq, "cDAQ1Mod4", "ai1", "Voltage");
```

Acquire a Single Scan as a Table

Use `read` to acquire a single scan. The result is a table with two data columns because two input channels are used to acquire the scan.

```
tabledata = read(dq)
```

```
tabledata =

  1×2 timetable

    Time     cDAQ1Mod4_ai0    cDAQ1Mod4_ai1
    _____    _____________    _____________

    0 sec     0.00081472       0.00090579
```

Acquire a Single Scan as a Matrix

Use `read` to acquire a single scan. The result is an array of size 1x2 because two input channels are used to acquire the scan.

```
matrixdata = read(dq, "OutputFormat", "Matrix")
```

```
matrixdata =

   1.0e-03 *

    0.1270    0.9134
```

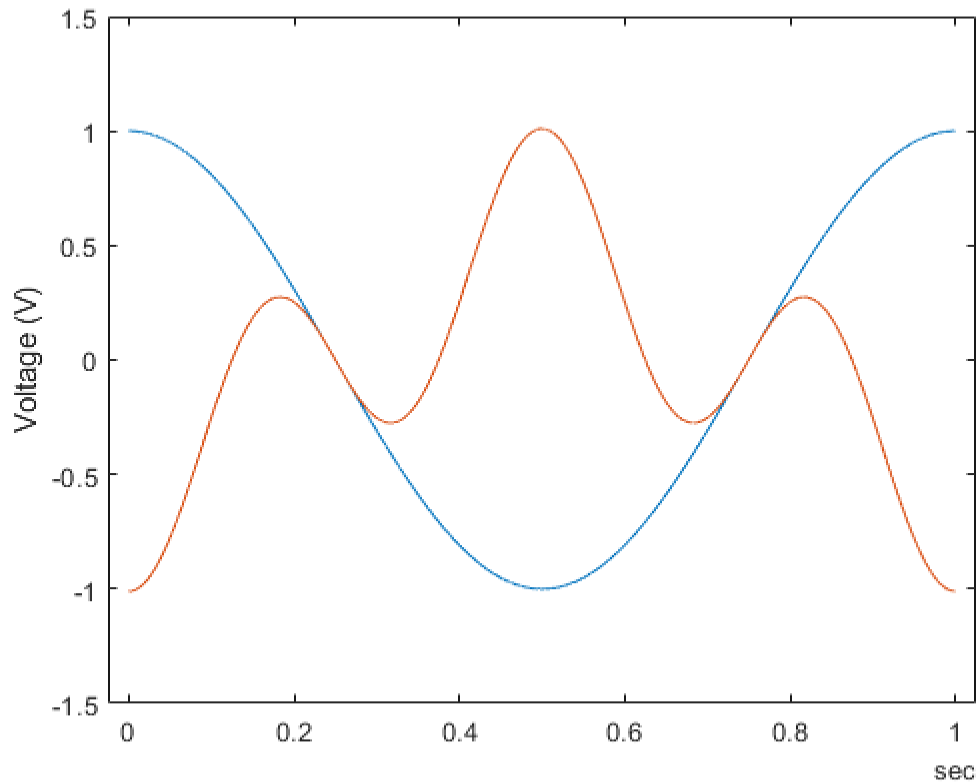
Acquire Data For a Specified Duration

Use `read` to acquire multiple scans, blocking MATLAB execution until all the data requested is acquired. The acquired data is returned as a timetable with width equal to the number of channels and height equal to the number of scans.

```
% Acquire data for one second at 8000 scans per second.
data = read(dq, seconds(1));
```

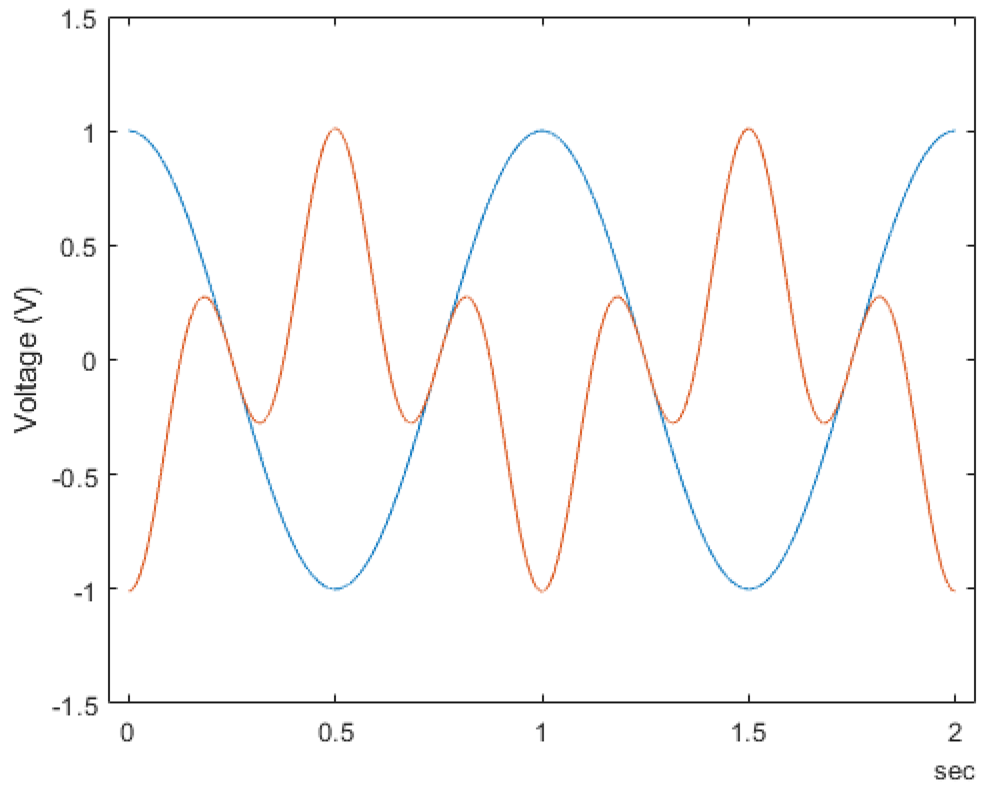
Plot the Acquired Data

```
plot(data.Time, data.Variables);
ylabel("Voltage (V)")
```



Acquire Specified Number of Scans

```
data = read(dq, 2*dq.Rate);
plot(data.Time, data.Variables);
ylabel("Voltage (V)")
```


1.5
1
0.5
0
-0.5
-1
-1.5
Voltage (V)
0
0.5
1
1.5
2
sec

Acquire Continuous and Background Data Using NI Devices

This example shows how to acquire analog input data using non-blocking commands. This allows you to continue working in the MATLAB command window during the acquisition. This is called **background acquisition**. Use **foreground acquisition** to cause MATLAB to wait for the entire acquisition to complete before you can execute your next command.

Create and Configure the DataAcquisition Object

Use `daq` to create a DataAcquisition object and `addinput` to add an input channel to it. This example uses an NI 9205 module in National Instruments® CompactDAQ Chassis NI cDAQ-9178. This is module 1 in the chassis.

```
dq = daq("ni");
addinput(dq, "cDAQ1Mod1", "ai0", "Voltage");
dq.Rate = 2000;
```

Plot Live Data as It Is Acquired

During a background acquisition, the DataAcquisition can handle acquired data in a specified way using the `ScansAvailableFcn` property.

```
dq.ScansAvailableFcn = @(src,evt) plotDataAvailable(src, evt);
```

Set ScansAvailableFcnCount

By default, the ScansAvailableFcn is called 10 times per second. Modify the `ScansAvailableFcnCount` property to decrease the call frequency. The ScansAvailableFcn will be called when the number of points accumulated exceeds this value. Set the ScansAvailableFcnCount to the rate, which results in one call to ScansAvailableFcn per second.

```
dq.ScansAvailableFcnCount = 2000;
```

Start the Background Acquisition

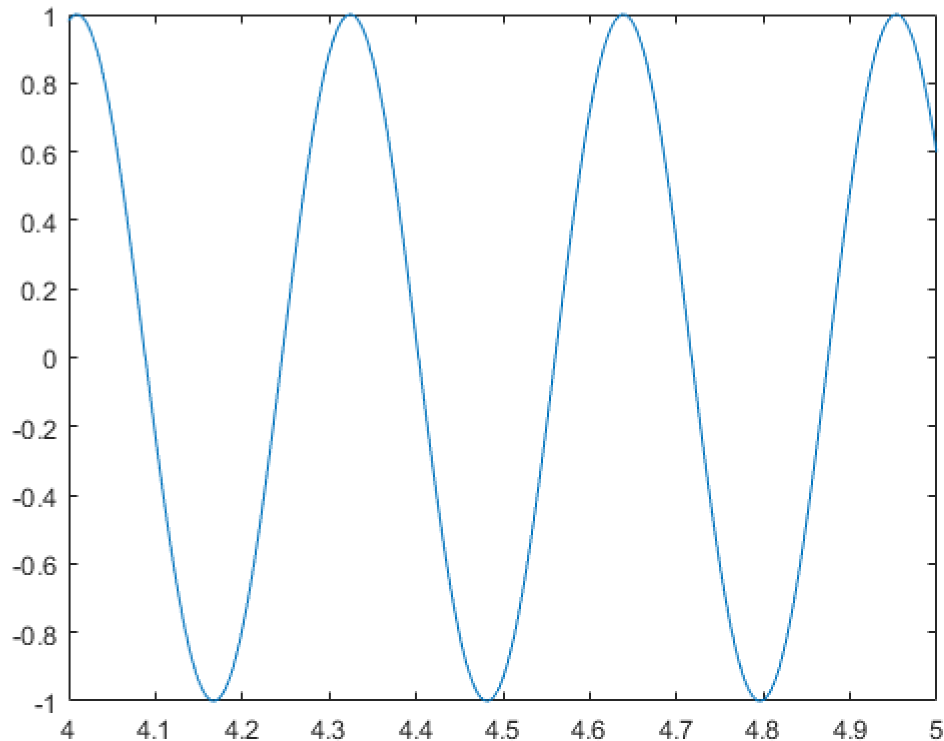
Use `start` to start the background acquisition.

```
start(dq, "Duration", seconds(5))
```

There are no other calculations to perform and the acquisition is set to run for the entire five seconds. Use `pause` in a loop to monitor the number of scans acquired for the duration of the acquisition.

```
while dq.Running
    pause(0.5)
    fprintf("While loop: Scans acquired = %d\n", dq.NumScansAcquired)
end

fprintf("Acquisition stopped with %d scans acquired\n", dq.NumScansAcquired);
```

```
While loop: Scans acquired = 1000
```



Capture a Unique Event in Incoming Data

Acquire continuously until a specific condition is met. In this example, acquire until the signal equals or exceeds 1 V.

```
dq.ScansAvailableFcn = @(src,evt) stopWhenEqualsOrExceedsOneV(src, evt);
```

Configure the DataAcquisition to acquire continuously. The listener detects the 1V event and calls `stop`.

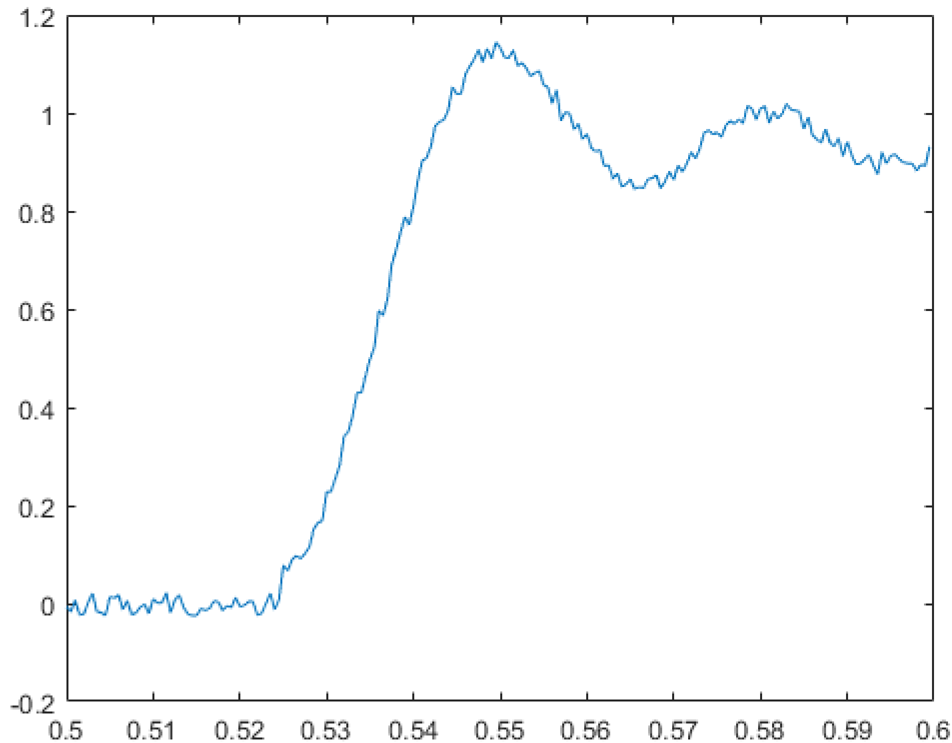
```
start(dq, "continuous");
```

Use `pause` in a loop to monitor the number of scans acquired for the duration of the acquisition. Note that the status string displayed by the `ScansAvailableFcn` may appear before the last status string displayed by the while loop.

```
while dq.Running
    pause(0.5)
    fprintf("While loop: Scans acquired = %d\n", dq.NumScansAcquired)
end

fprintf("Acquisition has terminated with %d scans acquired\n", dq.NumScansAcquired);

dq.ScansAvailableFcn = [];
```

```
While loop: Scans acquired = 1000
Detected voltage exceeds 1V: stopping acquisition
```

```
While loop: Scans acquired = 2000
Acquisition has terminated with 2000 scans acquired
```




```
function plotDataAvailable(src, ~)
    [data, timestamps, ~] = read(src, src.ScansAvailableFcnCount, "OutputFormat", "Matrix");
    plot(timestamps, data);
end

function stopWhenEqualsOrExceedsOneV(src, ~)
    [data, timestamps, ~] = read(src, src.ScansAvailableFcnCount, "OutputFormat", "Matrix");
    if any(data >= 1.0)
        disp('Detected voltage exceeds 1V: stopping acquisition')
        % stop continuous acquisitions explicitly
        src.stop()
        plot(timestamps, data)
    else
        disp('Continuing to acquire data')
    end
end
```

```
While loop: Scans acquired = 2200
While loop: Scans acquired = 3200
While loop: Scans acquired = 4200
While loop: Scans acquired = 5200
While loop: Scans acquired = 6200
While loop: Scans acquired = 7200
While loop: Scans acquired = 8200
While loop: Scans acquired = 9200
```

```
While loop: Scans acquired = 10000
Acquisition stopped with 10000 scans acquired
```

Acquire Data from Multiple Channels using an MCC Device

This example shows how to acquire data from multiple analog input channels with an MCC device.

Hardware Setup

This example uses a Measurement Computing USB-1608FS-Plus device to log data from analog input channels 0 and 9, which are connected to the outputs of a function generator.

Display a List of Available Devices

Discover devices connected to your system using `daqlist`.

```
d = daqlist("mcc")
```

```
d =

  1×4 table

    DeviceID                       Description                              Model              Devi
    ________    ____________________________________________    __________________    ___________

    "Board0"    "Measurement Computing Corp. USB-1608FS-Plus"    "USB-1608FS-Plus"    [1×1 daq.sc
```

Get Details About a Device

The `daqlist` function shows you the overview of devices available. You can find additional device details by reviewing the `DeviceInfo` field of the table.

```
deviceInfo = d{1, "DeviceInfo"}
```

```
deviceInfo =

mcc: Measurement Computing Corp. USB-1608FS-Plus (Device ID: 'Board0')
   Analog input supports:
      4 ranges supported
      Rates from 0.1 to 100000.0 scans/sec
      8 channels ('Ai0' - 'Ai7')
      'Voltage' measurement type
```

Create a DataAcquisition and Add Input Channels

The `daq` function creates a DataAcquisition object. The DataAcquisition contains information describing hardware, scan rate, and other properties associated with the acquisition.

```
dq = daq("mcc")

% The |addinput| function adds an analog input channel to
% the DataAcquisition.  You can add more than one channel to a
% DataAcquisition.
```

```
ch1 = addinput(dq, "Board0", 0, "Voltage");
ch2 = addinput(dq, "Board0", 1, "Voltage");
```

```
dq = 

DataAcquisition using Measurement Computing Corp. hardware:

                     Running: 0
                        Rate: 1000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

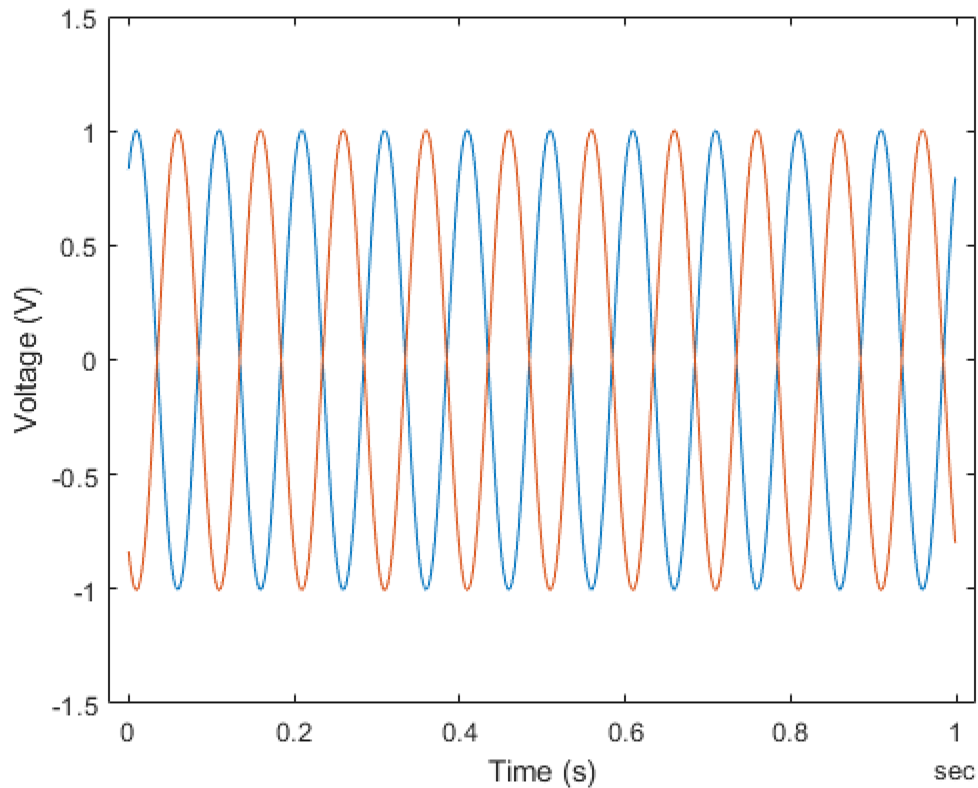
Acquire Timestamped Data

The `read` function starts the acquisition and returns the results as a timetable.

```
data = read(dq, seconds(1));
```

Plot Acquired Data

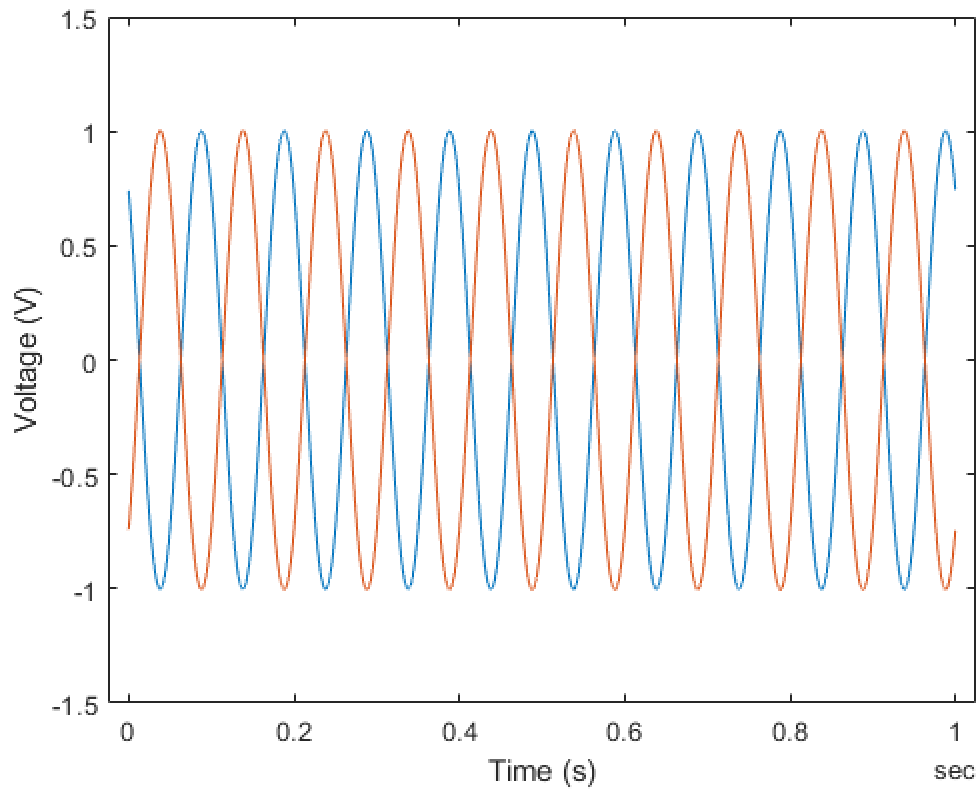
```
plot(data.Time, data.Board0_Ai0, data.Time, data.Board0_Ai1);
xlabel('Time (s)');
ylabel('Voltage (V)');
```



Change Default Properties of the Acquisition

By default, acquisitions run for one second at 1000 scans per second. To acquire at a different rate, change the `Rate` property.

```
dq.Rate = 10000;
[data, startTime] = read(dq, seconds(1));
plot(data.Time, data.Board0_Ai0, data.Time, data.Board0_Ai1);
xlabel('Time (s)');
ylabel('Voltage (V)');
```


1.5
1
0.5
0
-0.5
-1
-1.5
Voltage (V)
0
0.2
0.4
0.6
0.8
1
Time (s)
sec

Acquire Data from an Accelerometer

This example shows how to acquire and display data from an accelerometer attached to a vehicle driven under uneven road conditions.

Discover Devices that Support Accelerometers

To discover a device that supports accelerometers, access the device in the table returned by the `daqlist` command. This example uses National Instruments® CompactDAQ Chassis NI cDAQ-9178 and module NI 9234 with ID `cDAQ1Mod3`.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID               Description                    Model             DeviceInfo
    ___________    __________________________________    ____________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"       [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"       [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"       [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"       [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"       [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"       [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"       [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"       [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"     [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"   [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"     [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"     [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{3, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9234 (Device ID: 'cDAQ1Mod3')
   Analog input supports:
      -5.0 to +5.0 Volts range
      Rates from 1000.0 to 51200.0 scans/sec
      4 channels ('ai0','ai1','ai2','ai3')
      'Voltage','Accelerometer','Microphone','IEPE' measurement types

This module is in slot 3 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Add an Accelerometer Channel

Create a DataAcquisition, and add an analog input channel with `Accelerometer` measurement type.

```
dq = daq("ni");
ch = addinput(dq, "cDAQ1Mod3", "ai0", "Accelerometer");
```

Set the Scan Rate

Change the acquisition scan rate to 4000 scans per second.

```
dq.Rate = 4000;
```

Set the Sensitivity

You must set the `Sensitivity` value to the value specified in the accelerometer's data sheet. This example uses a ceramic shear accelerometer model 352C22 from PCB Piezotronics with 9.22 mV per gravity.

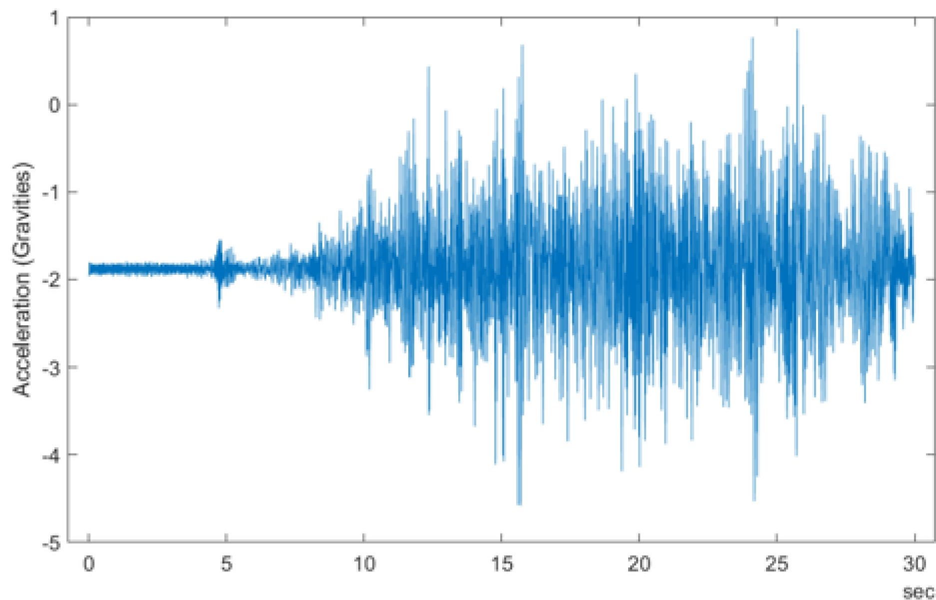
```
ch.Sensitivity = 0.00922;
ch
```

```
ch =

    Index    Type       Device       Channel      Measurement Type             Range
    _____    ____    ____________    _______    ________________________    ____________________

      1      "ai"    "cDAQ1Mod3"     "ai0"      "Accelerometer (Diff)"    "-5.0 to +5.0 Volts"
```

Acquire and Plot Data

Use the `read` command to acquire data for 30 seconds.

```
data = read(dq, seconds(30));
plot(data.Time, data.cDAQ1Mod3_ai0);
ylabel("Acceleration (Gravities)");
```



Measure Strain Using an Analog Bridge Sensor

This example shows how to acquire bridge circuit voltage ratio data using a CompactDAQ module, then compute and plot strain values. This example does not apply to USB devices such as the NI USB-9219.

Discover Devices that Support Bridge Sensor Measurements

To discover a device that supports bridge sensor measurements, access the device in the array returned by `daqlist` command. For this example use National Instruments® CompactDAQ Chassis NI cDAQ-9178 and module NI 9219 with ID `cDAQ1Mod7`.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID                 Description                    Model             DeviceInfo
    ___________    __________________________________    _____________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"        [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"    [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{7, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9219 (Device ID: 'cDAQ1Mod7')
   Analog input supports:
      9 ranges supported
      Rates from 0.1 to 100.0 scans/sec
      4 channels ('ai0','ai1','ai2','ai3')
      'Voltage','Current','Thermocouple','RTD','Bridge' measurement types

This module is in slot 7 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Create an Analog Input Channel

Create a DataAcquisition and add an analog input channel with the `Bridge` measurement type. There are two strain gauges connected to the NI 9219 in half bridge configuration.

```
dq = daq("ni");
dq.Rate = 10;
ch = addinput(dq, "cDAQ1Mod7", "ai0", "Bridge");
```

Set Channel Properties

You must set the bridge mode according to the bridge circuit configuration and the nominal resistance to the value specified by the strain gauge datasheet. In this example, the strain gauges used are the SGD-3/350-LY13 linear strain gauges from Omega®, with a nominal resistance of 350 ohms, and the bridge is configured as a half-bridge.

```
ch.BridgeMode = "Half";
ch.NominalBridgeResistance = 350;
```

Set ADCTimingMode

By default, the ADC timing mode `ADCTimingMode` of the channel is set to `'HighResolution'`. Set the `ADCTimingMode` to `'HighSpeed'`.

```
ch.ADCTimingMode = "HighSpeed";
```

Acquire Data

Use `read` to acquire 10 seconds of data.

```
data = read(dq, seconds(10));
```

Calculate Strain from Voltage Ratio

The acquired data is the ratio of measured voltage to excitation voltage.

This data is used to compute strain values using a conversion formula (as determined by your bridge configuration).

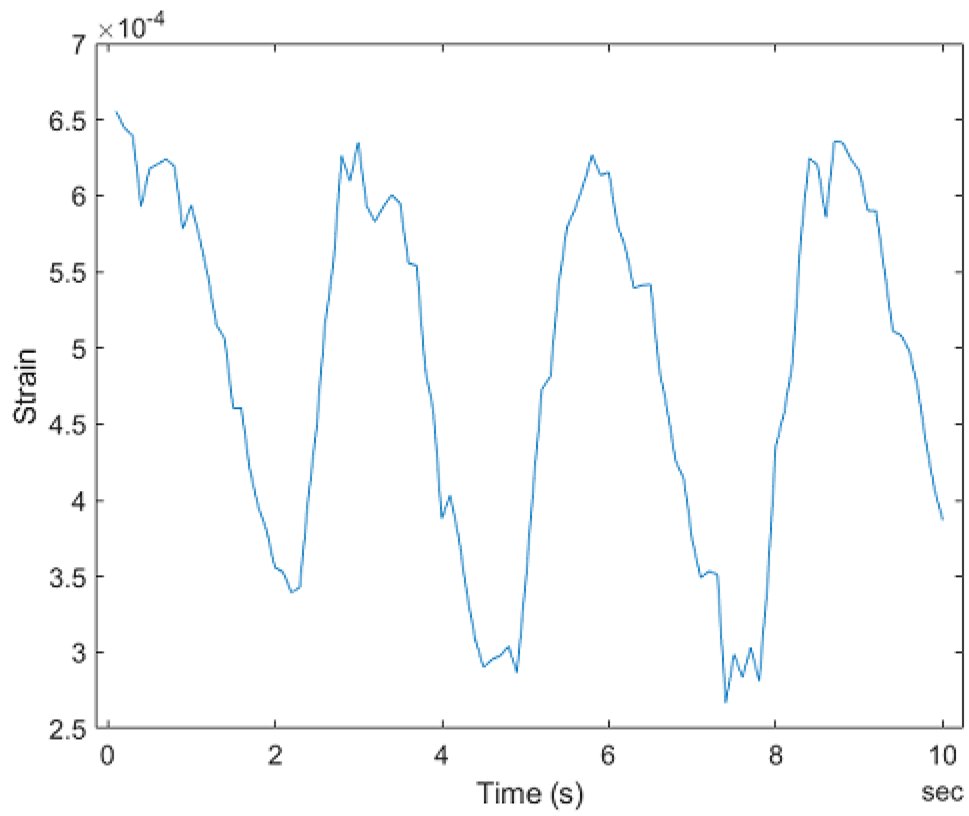
For half bridge configuration, use

```
strain = -2*Vr/GF
```

where GF is gauge factor provided in the sensor data sheet and Vr is the voltage ratio output as measured by your bridge channel.

Assume negligible lead wire resistance in this case. For the strain gauge used in this example, GF = 2.13.

```
GF = 2.13;
strain = -2*data.cDAQ1Mod7_ai0/GF;
plot(data.Time, strain);
xlabel('Time (s)');
ylabel('Strain');
```


×10^-4
7
6.5
6
5.5
5
4.5
4
3.5
3
2.5
Strain
0
2
4
6
8
10
Time (s)
sec

Acquire Temperature Data From a Thermocouple

This example shows how to read data from NI devices that support thermocouple measurements.

Discover Devices That Support Thermocouples

To discover a device that supports thermocouple measurements, access the device in the table returned by the `daqlist` command. This example uses an NI 9213 device. This is a 16 channel thermocouple module and is module 6 in the chassis.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID                Description                    Model              DeviceInfo
    ___________    ____________________________________    ______________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"          "NI 9205"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"          "NI 9263"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"          "NI 9234"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"          "NI 9201"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"          "NI 9402"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"          "NI 9213"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"          "NI 9219"         [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"          "NI 9265"         [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"        "PCIe-6363"       [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"      "NI ELVIS II"     [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"        "PCIe-6363"       [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"        "PCIe-6363"       [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{6, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9213 (Device ID: 'cDAQ1Mod6')
   Analog input supports:
      -0.078 to +0.078 Volts range
      Rates from 0.1 to 1351.4 scans/sec
      16 channels ('ai0' - 'ai15')
      'Voltage','Thermocouple' measurement types

This module is in slot 6 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Add a Thermocouple Channel

Create a DataAcquisition, change its scan `Rate` to four scans per second, and add an analog input channel with `Thermocouple` measurement type.

```
dq = daq("ni");
dq.Rate = 4;
ch = addinput(dq, "cDAQ1Mod6", "ai0", "Thermocouple");
```

Configure Channel Properties

Set the thermocouple type to K and units to Kelvin (the thermocouple type should match the sensor configuration).

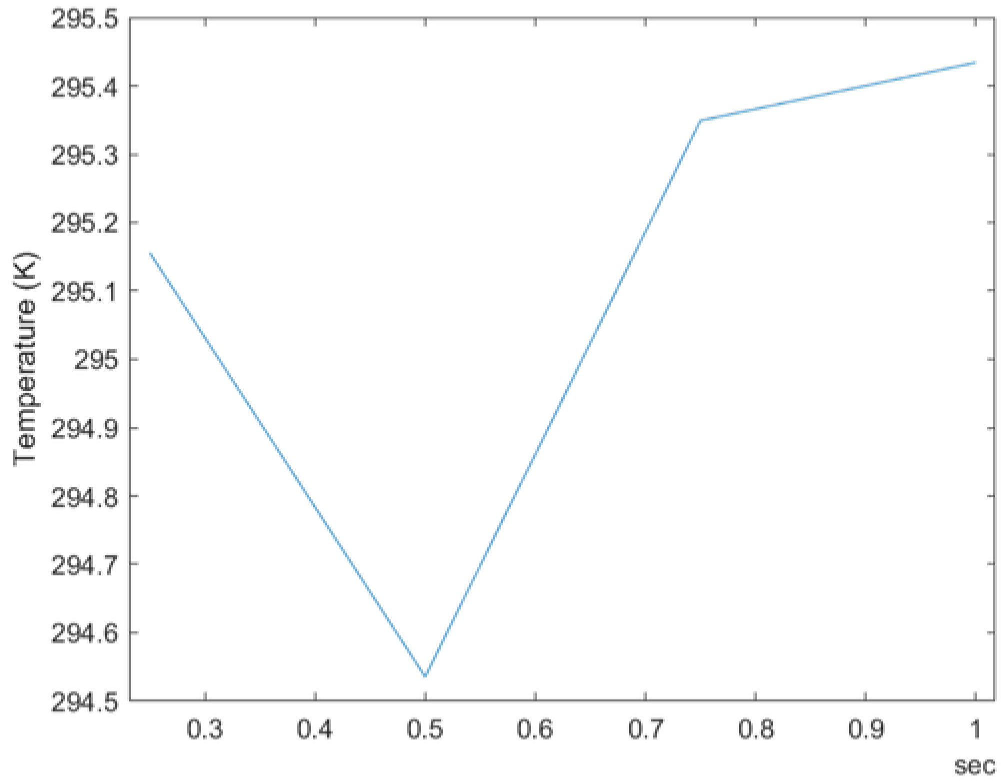
```
ch.ThermocoupleType = 'K';
ch.Units = 'Kelvin';
ch
```

```
ch = 

    Index    Type       Device       Channel    Measurement Type            Range                 
    _____    ____    ___________    _______    ________________    ____________________    ______

      1      "ai"    "cDAQ1Mod6"     "ai0"     "Voltage (Diff)"    "+73 to +1523 Kelvin"    "cDAQ
```

Acquire and Plot Data

Use the `read` command to acquire data.

```
data = read(dq, seconds(1));
plot(data.Time, data.cDAQ1Mod6_ai0);
ylabel('Temperature (K)');
```


295.5
295.4
295.3
295.2
295.1
295
294.9
294.8
294.7
294.6
294.5
Temperature (K)
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
sec

Acquire Temperature Data From an RTD

This example shows how to acquire temperature data from a Resistive temperature device (RTD) and display the readings. The device is attached inside a PC case to monitor the internal temperature changes.

Discover Devices That Support RTDs

To discover a device that supports bridge sensor measurements, access the device in the table returned by the `daqlist` command. This example uses an NI 9219 module in National Instruments® CompactDAQ Chassis NI cDAQ-9178. This is a 24-Bit Universal Analog Input module and is module 7 in the chassis.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID                Description                   Model             DeviceInfo
    ___________    ___________________________________    _____________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"         "NI 9205"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"         "NI 9263"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"         "NI 9234"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"         "NI 9201"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"         "NI 9402"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"         "NI 9213"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"         "NI 9219"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"         "NI 9265"        [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"       "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"     "NI ELVIS II"    [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"       "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"       "PCIe-6363"      [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{7, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9219 (Device ID: 'cDAQ1Mod7')
   Analog input supports:
      9 ranges supported
      Rates from 0.1 to 100.0 scans/sec
      4 channels ('ai0','ai1','ai2','ai3')
      'Voltage','Current','Thermocouple','RTD','Bridge' measurement types

This module is in slot 7 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Add an RTD Channel

Create a DataAcquisition, and add an analog input channel with `RTD` measurement type.

```
dq = daq("ni");
dq.Rate = 30;
ch = addinput(dq, "cDAQ1Mod7", "ai3", "RTD");
```

Set Sensor Properties

Refer to the sensor data sheet and match the values accordingly. In this example, an SA1-RTD series sensor from Omega® is used. Set units to `"Fahrenheit"`, RTD type to `"Pt3851"`, configure the RTD circuit as `"FourWire"`, and set the resistance to 100 ohms.

```
ch.Units = "Fahrenheit";
ch.RTDType = "Pt3851";
ch.RTDConfiguration = "FourWire";
ch.R0 = 100;
```

Set ADCTimingMode

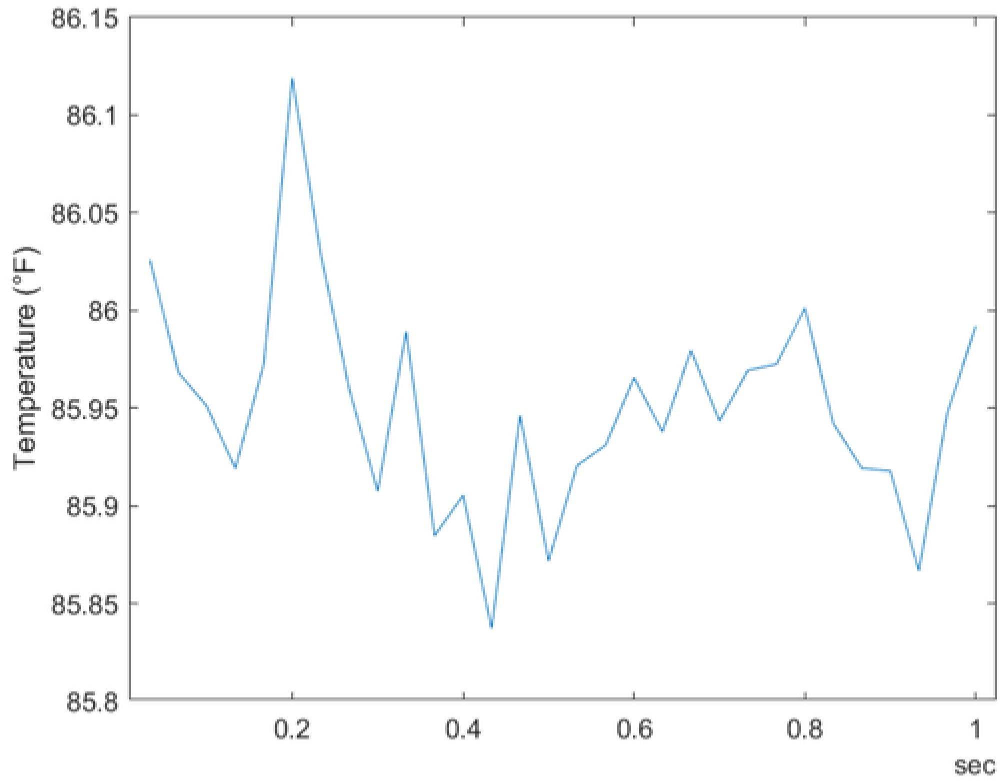
By default, the ADC timing mode `ADCTimingMode` of the channel is set to `"HighResolution"`. Set the `ADCTimingMode` to `"HighSpeed"`.

```
ch.ADCTimingMode = "HighSpeed";
```

Acquire and Plot Data

Use the `read` command to acquire data.

```
data = read(dq, seconds(1));
plot(data.Time, data.cDAQ1Mod7_ai3);
degreeSign = 176;
ylabel(sprintf("Temperature (%cF)", degreeSign));
```


86.15
86.1
86.05
86
85.95
85.9
85.85
85.8
Temperature (°F)
0.2
0.4
0.6
0.8
1
sec

Acquire and Analyze Sound Pressure Data From an IEPE Microphone

This example shows how to acquire and display sound pressure data from a PCB® IEPE array microphone, Model 130E20. The sensor is recording sound pressure generated by a tuning fork at Middle C (261.626 Hz) frequency.

Discover Devices That Support Microphones

To discover a device that supports microphone measurements, access the device in the table returned by the `daqlist` command. For this example, the microphone is connected on channel 0 of National Instruments® device NI 9234 on CompactDAQ Chassis NI cDAQ-9178 with ID `cDAQ1Mod3`.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID                 Description                    Model              DeviceInfo
    ___________    __________________________________    _____________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"        [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"    [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{3, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9234 (Device ID: 'cDAQ1Mod3')
   Analog input supports:
      -5.0 to +5.0 Volts range
      Rates from 1000.0 to 51200.0 scans/sec
      4 channels ('ai0','ai1','ai2','ai3')
      'Voltage','Accelerometer','Microphone','IEPE' measurement types

This module is in slot 3 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Create a DataAcquisition and Add Microphone Channel

Create a DataAcquisition and add a channel with `Microphone` measurement type.

```
dq = daq("ni");
ch = addinput(dq, "cDAQ1Mod3", "ai0", "Microphone");
```

Set Sensor Properties

Set the microphone channel `Sensitivity` property to the value specified in the sensor's data sheet. For this sensor, the `Sensitivity` value is 0.037 Volts/Pascal. Examine the channel properties to see the changes in the device configuration.

```
ch.Sensitivity = 0.037;
ch
```

```
ch =

    Index    Type      Device       Channel      Measurement Type             Range
    _____    ____    ___________    _______    ____________________    ________________________    _

      1      "ai"    "cDAQ1Mod3"     "ai0"     "Microphone (Diff)"     "-200 to +200 Pascals"     "
```

Configure and Start Acquisition

Set the acquisition scan rate to 51200 scans per second, then use `read` to acquire four seconds of data.

```
dq.Rate = 51200;
tt = read(dq, seconds(4));
t = tt.Time;
data = tt.cDAQ1Mod3_ai0;
```

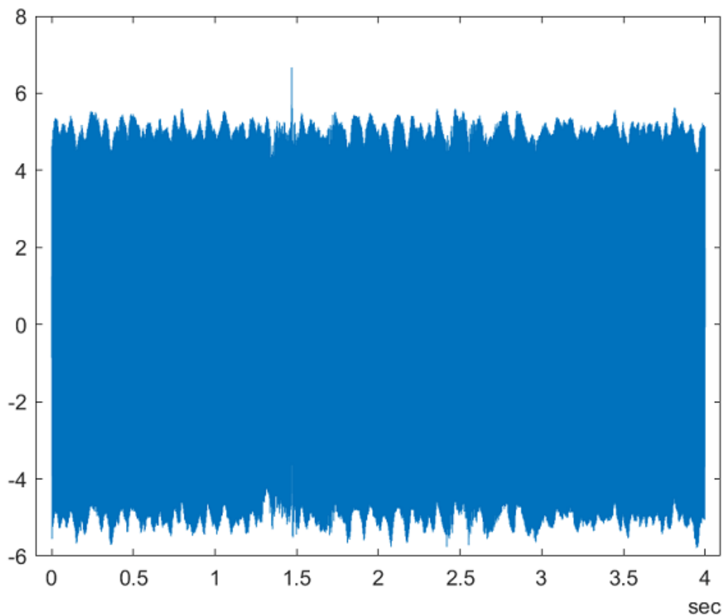
Analyze Data

Use `audioplayer` to play back the acquired microphone signal

```
p = audioplayer(data, dq.Rate);
play(p);
```

Examine the Data in the Time Domain

```
plot(t, data);
ylabel('Sound Pressure (pascals)');
```



Examine the Data in the Frequency Domain

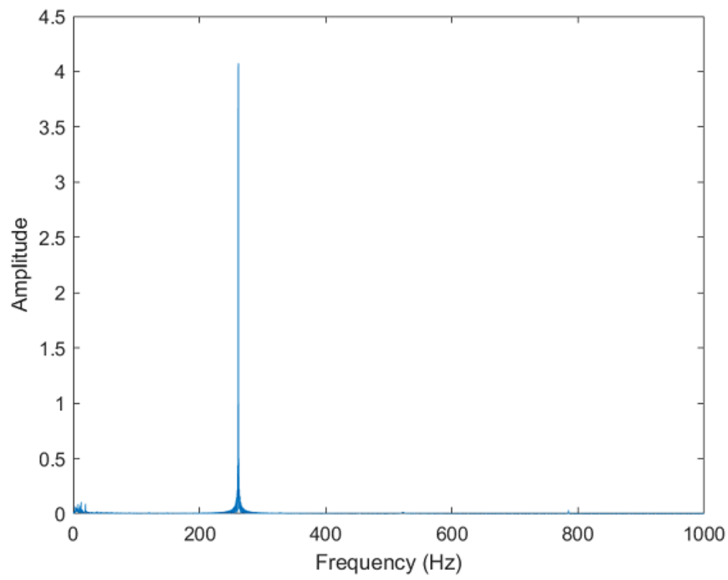
Use `fft` and the following parameters to calculate the single-sided amplitude spectrum of the incoming data:

- Calculate the length of signal (number of samples or entries in the table)
- Calculate the nfft
- Calculate amplitude and frequency

```
len = height(tt);
nfft = 2^nextpow2(len);
y = fft(data,nfft)/len;
f = dq.Rate/2*linspace(0,1,nfft/2+1);
A = 2*abs(y(1:nfft/2+1));
```

Plot the Single-Sided Amplitude Spectrum

```
plot(f,A);
xlim([0 1000]);
xlabel('Frequency (Hz)');
ylabel('Amplitude');
```



The plot shows a spike at 261.626 Hz. This matches the frequency of the tuning fork.

Acquire and Analyze Noisy Clock Signals

This example shows how to acquire clock signals and analyze transitions, pulses, and compute metrics including rise time, fall time, slew rate, overshoot, undershoot, pulse width, and duty cycle. This example uses Data Acquisition Toolbox in conjunction with the Signal Processing Toolbox.

Use Data Acquisition Toolbox to configure the acquisition. Use the statistics and measurement functions in Signal Processing Toolbox to analyze the data signal.

Create a DataAcquisition and Acquire a Clock Signal

Use `daq` to create a DataAcquisition and `addinput` to add a channel from the National Instruments® NI-9205 with ID of 'cDAQ1Mod1'.

```
dq = daq("ni");
addinput(dq,"cDAQ1Mod1","ai0","Voltage");
```

By default the DataAcquisition is configured to run at 1000 scans/second.

Change the scan rate to 250000 scans/second.

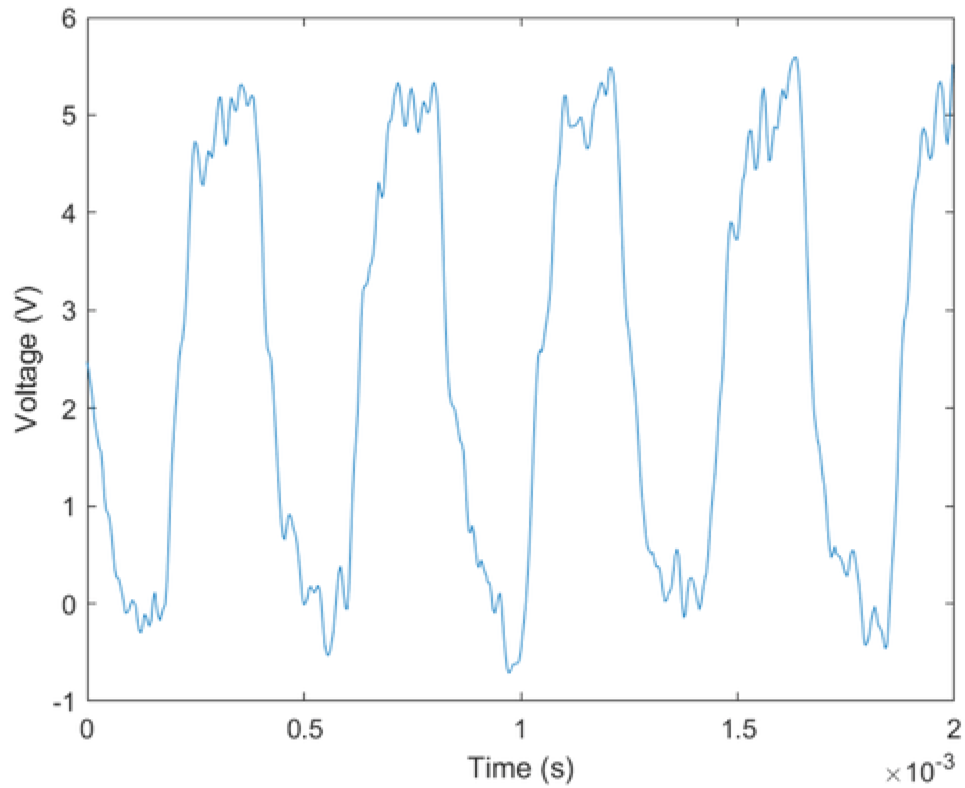
```
dq.Rate = 250000;
```

Use `read` to acquire multiple scans for 2 ms.

```
[data, time] = read(dq, milliseconds(2), "OutputFormat", "Matrix");
```

Plot the acquired clock signal (note that it is overdamped).

```
plot(time, data)
xlabel('Time (s)')
ylabel('Voltage (V)')
```

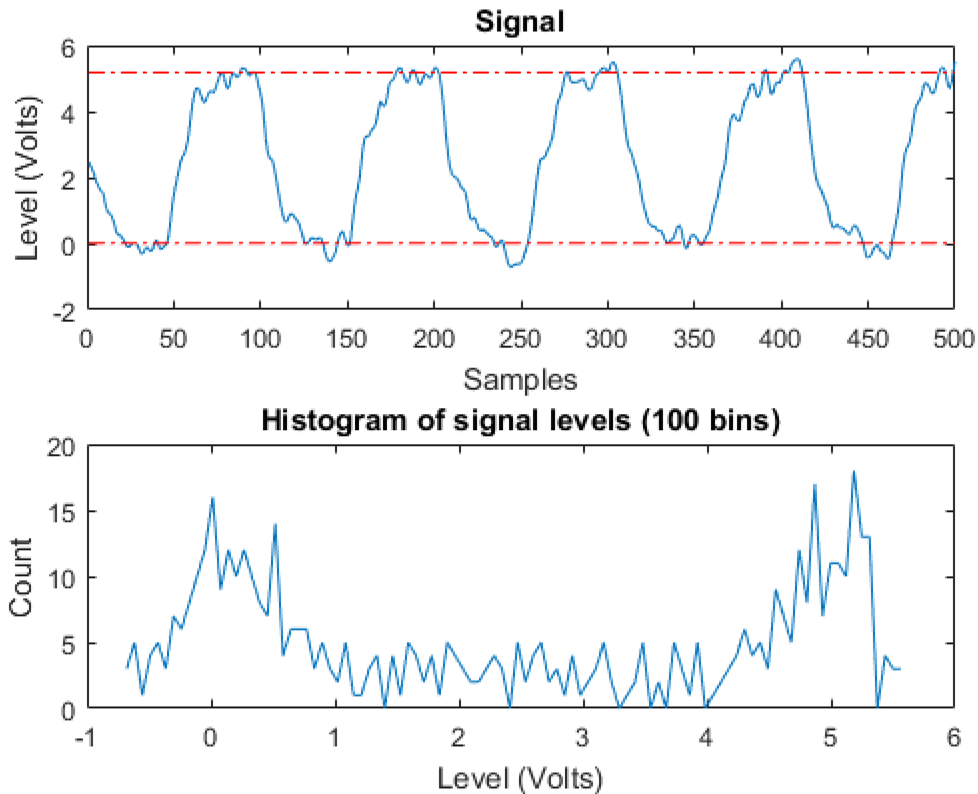


Estimate State Levels

Use `statelevels` with no output argument to visualize the state levels in a histogram.

```
statelevels(data)
```

```
ans =

    0.0138    5.1848
```



The computed histogram is divided into two equal sized regions between the first and last bin. The mode of each region of the histogram is returned as an estimated state level value in the command window.

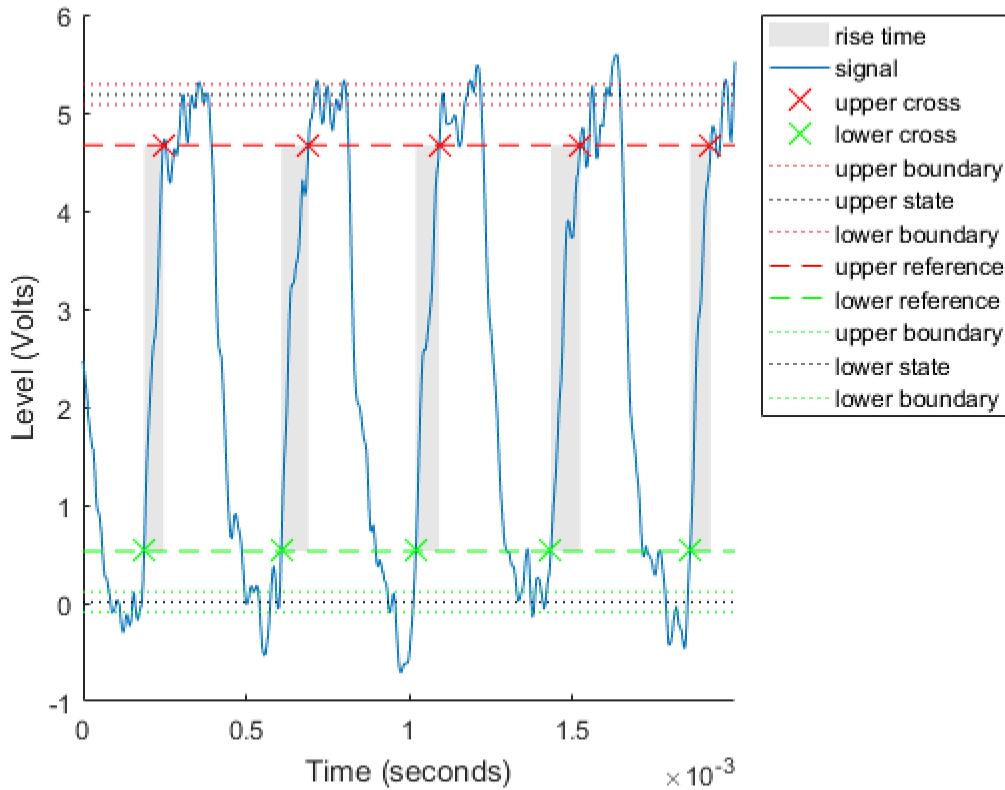
Use optional input arguments to specify the number of histogram bins, histogram bounds, and the state level estimation method.

Measure Rise Time, Fall Time, and Slew Rate

Use `risetime` with no output argument to visualize the rise time of positive edges.

```
risetime(data,time)
```

```
ans =

   1.0e-04 *

    0.5919
    0.8344
    0.7185
    0.8970
    0.6366
```

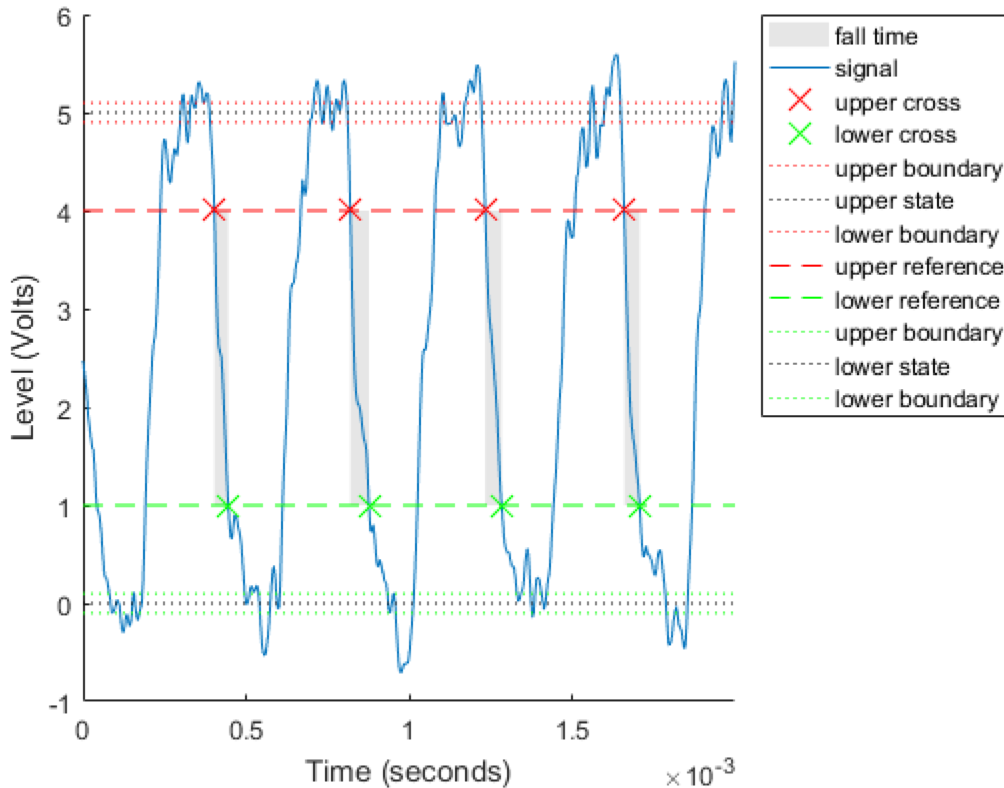


The default reference levels for computing rise time and fall time are set at 10% and 90% of the waveform amplitude.

Measure fall time by specifying custom reference and state levels.

```
falltime(data,time,'PercentReferenceLevels',[20 80],'StateLevels',[0 5])
```

```
ans =

   1.0e-04 *

    0.4294
    0.5727
    0.5032
    0.4762
```



Obtain measurements programmatically by calling functions with one or more output arguments. For uniformly sampled data, you can provide a sample rate in place of the time vector. Use `slewrate` to measure the slope of each positive or negative edge.

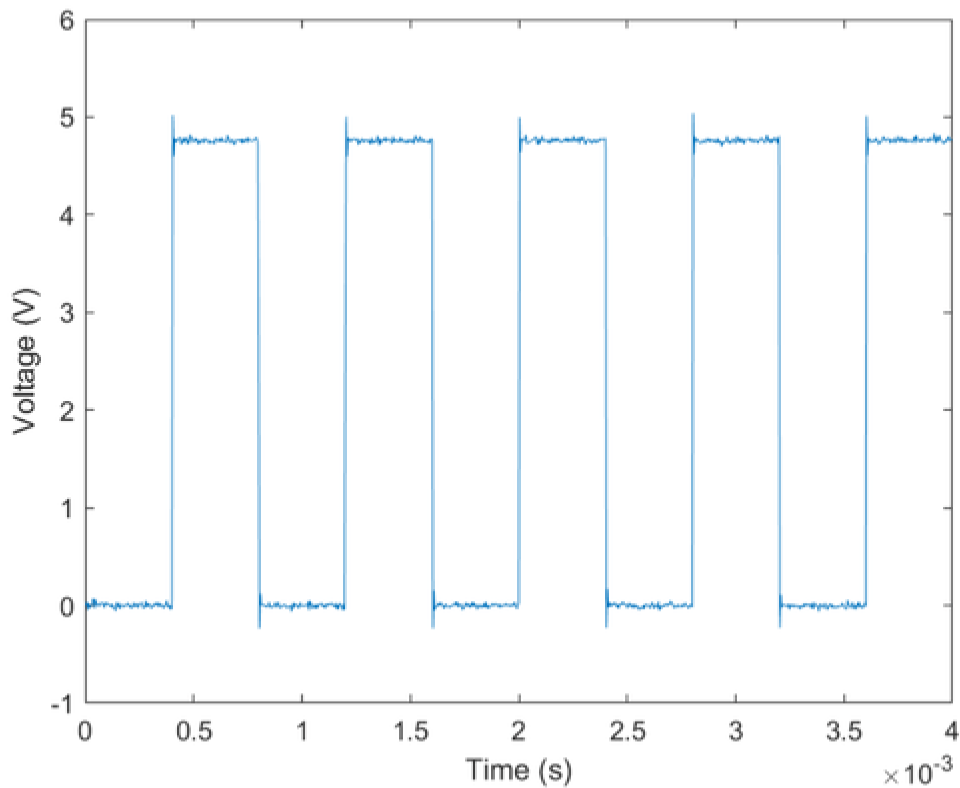
```
sr = slewrate(data(1:100), dq.Rate)
```

```
sr =

   7.0840e+04
```

Analyze Overshoot and Undershoot

Acquire a new underdamped clock signal.

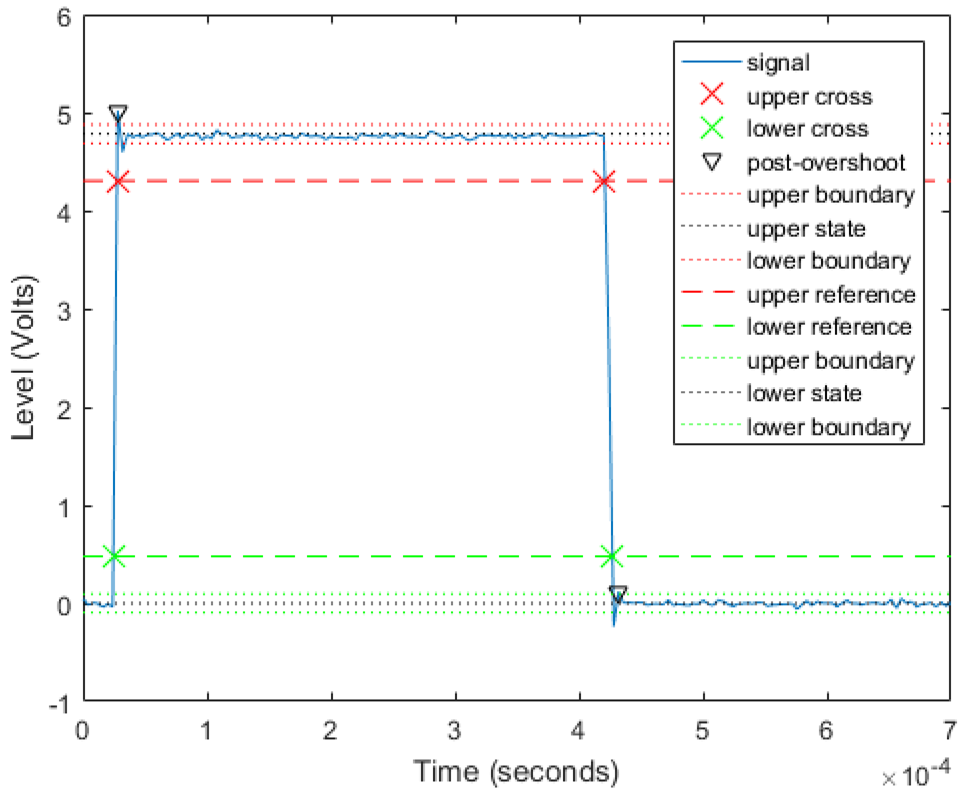
```
[data, time] = read(dq, milliseconds(4), "OutputFormat", "Matrix");
plot(time, data)
xlabel('Time (s)')
ylabel('Voltage (V)')
```



Underdamped clock signals exhibit overshoot. Overshoot is expressed as a percentage of the difference between state levels. Overshoot can occur just after an edge, at the start of the post-transition aberration region. This is called "postshoot" overshoot. Measure the overshoot using `overshoot`.

```
overshoot(data(95:270),dq.Rate)
legend('Location','NorthEast')
```

```
ans =

    4.9451
    2.5399
```

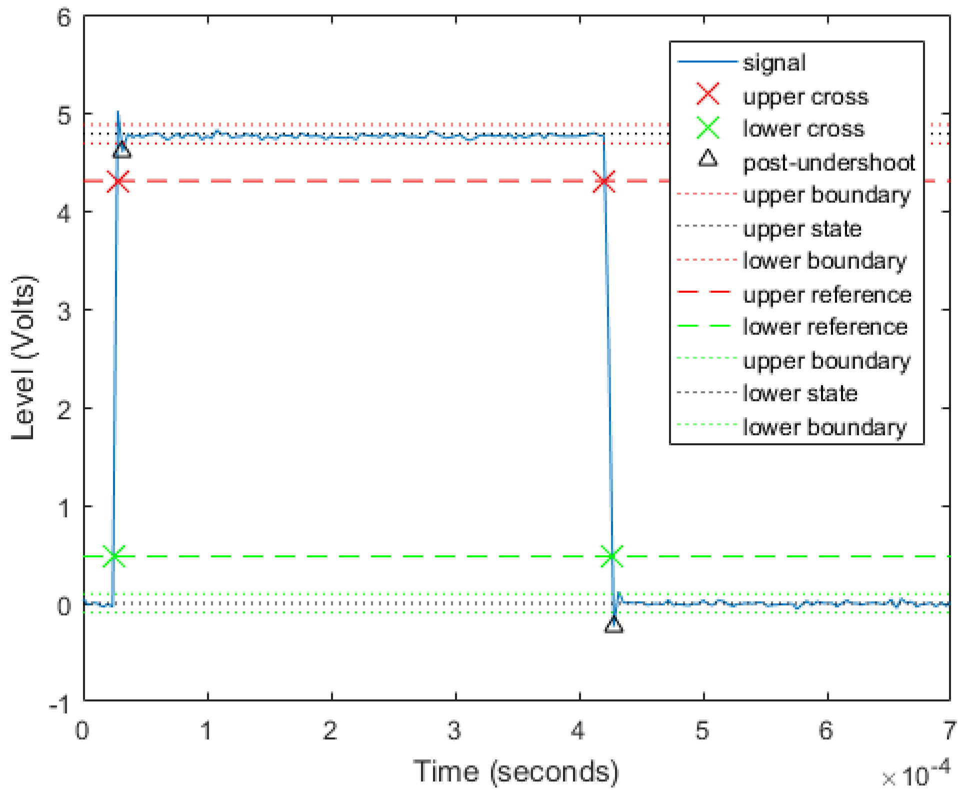


Overshoot can also occur just before an edge, at the end of the pre-transition aberration region. This is called "preshoot" overshoot.

Undershoot can occur in the pre-aberration and post-aberration regions and is expressed as a percentage of the difference between the state levels. Measure the undershoot with optional input arguments specifying the region to measure aberrations.

```
undershoot(data(95:270),dq.Rate,'Region','Postshoot')
legend('Location','NorthEast')
```

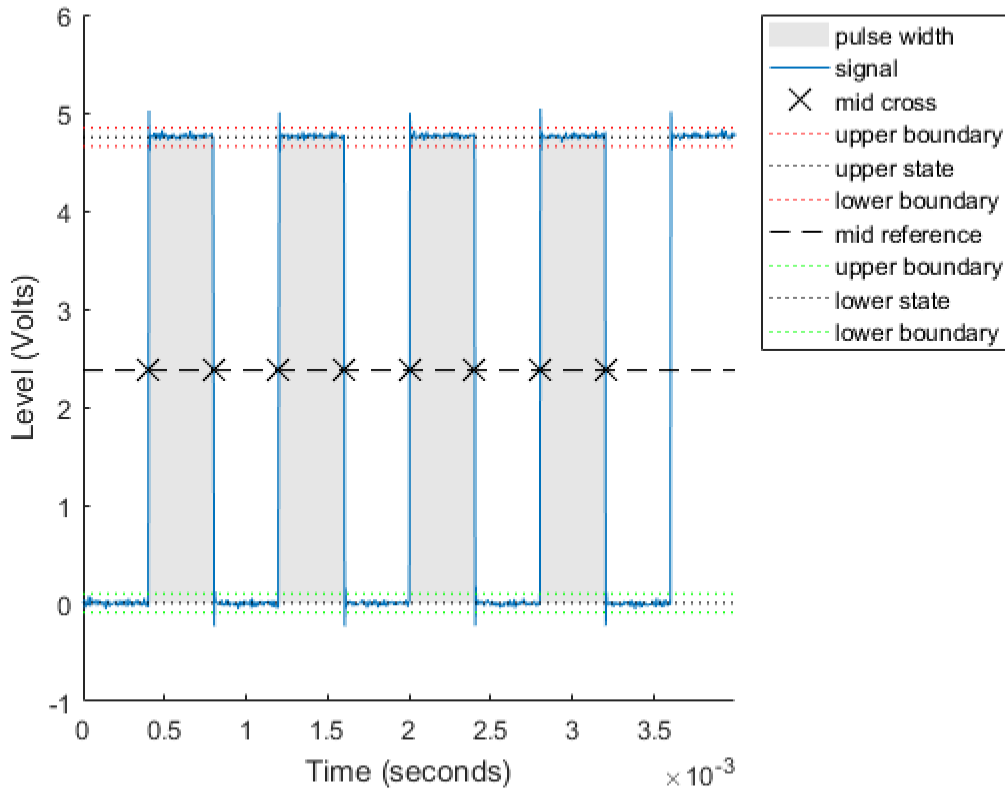
```
ans =

    3.8499
    4.9451
```



Measure Pulse Width and Duty Cycle

Use `pulsewidth` with no output argument to visualize highlighted pulse widths.

```
pulsewidth(data, time, 'Polarity', 'Positive');
```



This displays pulses of positive polarity. Select negative polarity to see the widths of negative polarity pulses.

Use `dutycycle` to compute the ratio of the pulse width to the pulse period for each positive-polarity or negative-polarity pulse. Duty cycles are expressed as a percentage of the pulse period.

```
d = dutycycle(data,time,'Polarity','negative')
```

```
d =

    0.4979
    0.5000
    0.5000
```

Use `pulseperiod` to obtain the periods of each cycle of the waveform. Use this information to compute other metrics such as the average frequency of the waveform or the total observed jitter.

```
pp = pulseperiod(data, time);

avgFreq = 1./mean(pp)
totalJitter = std(pp)
```

```
avgFreq =

    1.2500e+03
```

```
totalJitter =

   1.9866e-06
```

Generate Voltage Signals Using NI Devices

This example shows how to generate data using a National Instruments device.

Discover Devices That Can Output Voltage

To discover a device that supports analog outputs, access the device in the table returned by the `daqlist` command. This example uses an NI 9263 module in National Instruments® CompactDAQ Chassis NI cDAQ-9178. This is module 2 in the chassis.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID               Description                    Model             DeviceInfo
    ___________    __________________________________    _____________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"        [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"    [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{2, "DeviceInfo"}
```

```
deviceInfo = 

ni: National Instruments NI 9263 (Device ID: 'cDAQ1Mod2')
   Analog output supports:
      -10 to +10 Volts range
      Rates from 0.6 to 100000.0 scans/sec
      4 channels ('ao0','ao1','ao2','ao3')
      'Voltage' measurement type

This module is in slot 2 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Create a DataAcquisition and Add Analog Output Channels

Create a DataAcquisition, set the generation scan rate by setting the `Rate` property (the default is 1000 scans per second), and add analog output channels using `addoutput`.

```
dq = daq("ni");
dq.Rate = 8000;
```

```
addoutput(dq, "cDAQ1Mod2", "ao0", "Voltage");
addoutput(dq, "cDAQ1Mod2", "ao1", "Voltage");
```

Generate a Single Scan

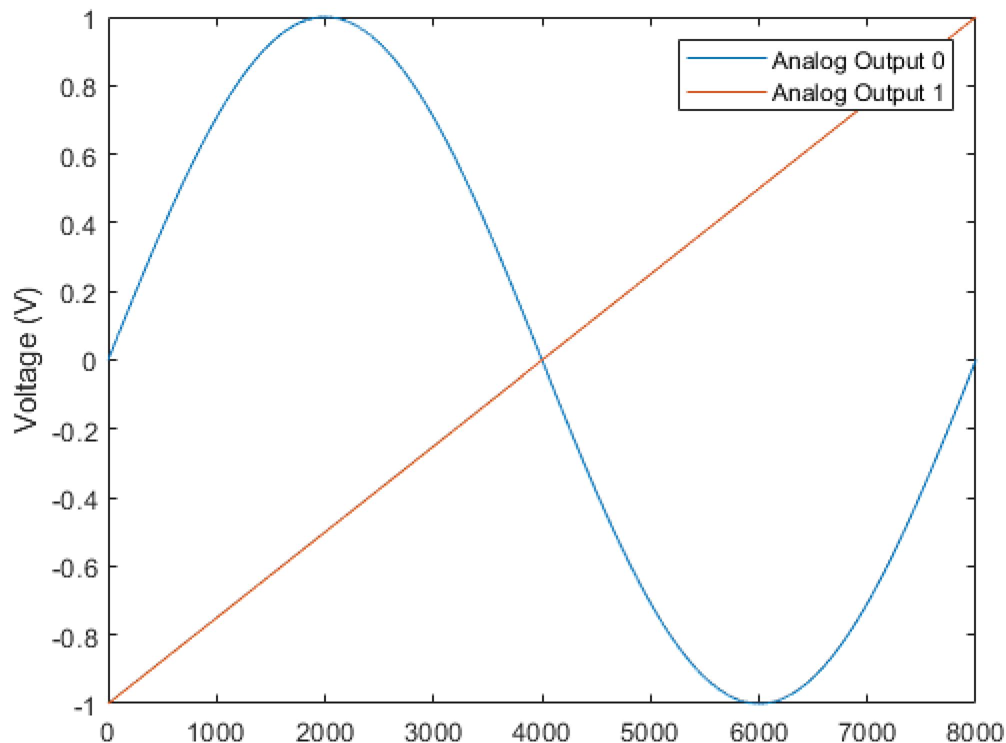
Use `write` to generate a single scan (2 V on each channel). The output scan data is a 1-by-N matrix where N corresponds to the number of output channels.

```
output = 2;
write(dq,[output output]);
```

Create and Plot the Output Data

Generate two output signals (a 1 Hz sine wave and a 1 Hz ramp) and plot them. The plot depicts the data generated on both channels for a device that supports simultaneous sampling.

```
n = dq.Rate;
outputSignal1 = sin(linspace(0,2*pi,n)');
outputSignal2 = linspace(-1,1,n)';
outputSignal = [outputSignal1 outputSignal2];
plot(1:n, outputSignal);
ylabel("Voltage (V)");
legend("Analog Output 0", "Analog Output 1");
```



Write Data

Use `write` to generate the output waveforms.

```
write(dq, outputSignal)
```

Generate Signals on NI Devices That Output Current

This example shows how to generate signals on an analog current output channel of an NI device.

Discover Devices That Can Output Current

To discover a device that outputs current, access the device in the table returned by the `daqlist` command. This example uses an NI 9265 module in a National Instruments® CompactDAQ Chassis NI cDAQ-9178. This is a 4-channel analog current output device and is module 8 in the chassis.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID                 Description                   Model              DeviceInfo
    ___________    __________________________________    _____________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"        [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"    [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{8, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments NI 9265 (Device ID: 'cDAQ1Mod8')
   Analog output supports:
      0 to +0.020 A range
      Rates from 0.6 to 100000.0 scans/sec
      4 channels ('ao0','ao1','ao2','ao3')
      'Current' measurement type

This module is in slot 8 of the 'cDAQ-9178' chassis with the name 'cDAQ1'.
```

Add an Output Current Channel

Create a DataAcquisition, and add two analog output channels.

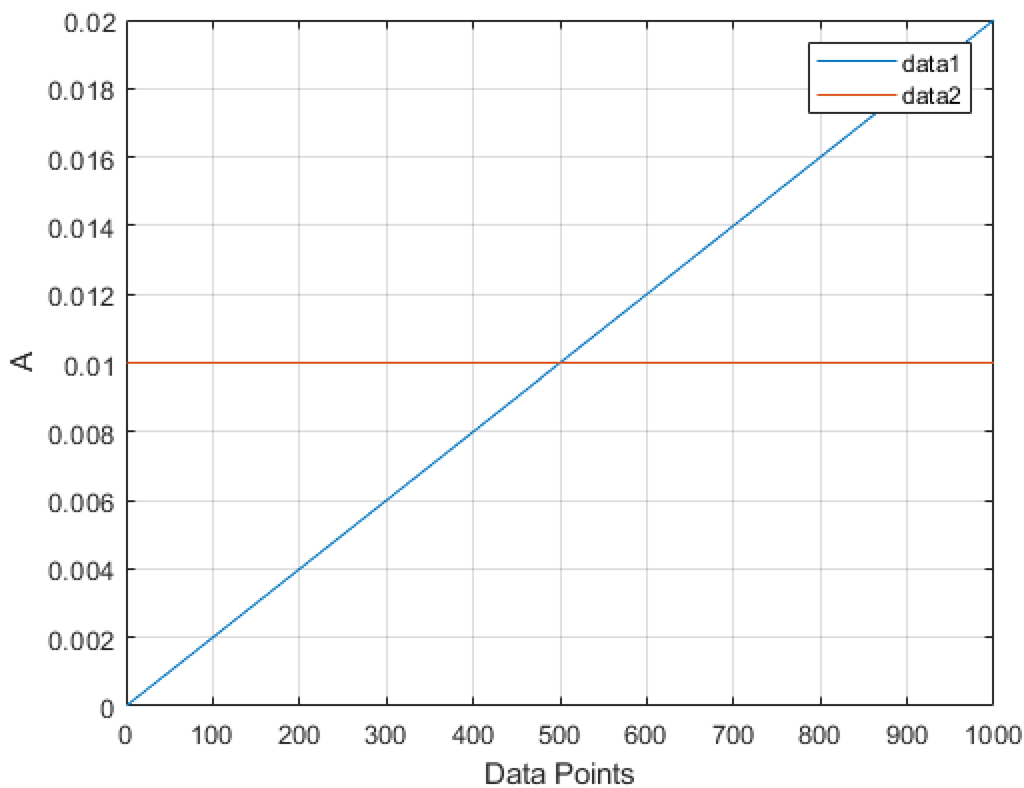
```
dq = daq("ni");
dq.Rate = 100;
```

```
ch1 = addoutput(dq, "cDAQ1Mod8", "ao0", "Current");
ch2 = addoutput(dq, "cDAQ1Mod8", "ao1", "Current");
```

Create and Plot the Output Data

The channels of the NI 9265 have a range of 0 to 20 mA. Produce a ramp from 0 to 20 mA on channel 1, and a constant 10 mA on channel 2. For each waveform, use enough points to generate 10 seconds of output data at the specified scan rate.

```
n = 10 * dq.Rate;
data1 = linspace(20e-6, 20e-3, n)';
data2 = repmat(10e-3, n, 1);
data = [data1 data2];
plot(1:n, data)
grid on
xlabel('Data Points')
ylabel('A')
legend('data1','data2')
```



Generate the Channel Output

Use `write` to generate the output waveforms.

```
write(dq, data)
```

Changing the Duration of the Output

To reduce the duration of the output, increase the generation scan rate. For one second of output, change the `Rate` to the number of samples in the scan.

```
dq.Rate = n;
write(dq, data)
```

Generate Continuous and Background Signals Using NI Devices

This example shows how to generate analog output data using non-blocking commands. This allows you to continue working in the MATLAB command window during the generation. This is called **background generation**. Use **foreground generation** to cause MATLAB to wait for the entire data generation to complete before you can execute your next command.

Create a DataAcquisition and Add Analog Output Channels

Use `daq` to create a DataAcquisition. This example uses an NI 9263 module in National Instruments® CompactDAQ Chassis NI cDAQ-9178. This is module 2 in the chassis. Output data on three channels at a rate of 10000 scans per second.

```
dq = daq("ni");
dq.Rate = 10000;
addoutput(dq, "cDAQ1Mod2", 0:2, "Voltage");
```

Create Synchronized Signals

Generate output signals by creating a pattern of data that is repeatedly written to the output device. The data for each channel is column based and the output signals are synchronized to a common clock.

Create 3 waveforms:

- `data0`: 1 cycle of a sine wave
- `data1`: 1 cycle of a sine wave with a 45 degree phase lag
- `data2`: 10 cycles of a sine wave

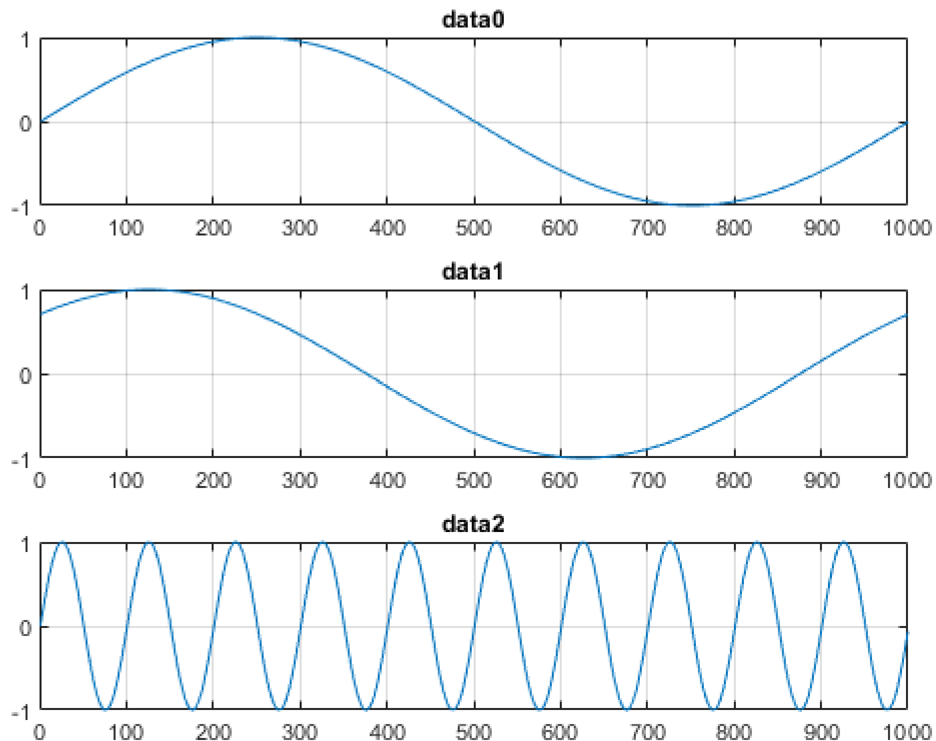
```
data0 = sin(linspace(0, 2*pi, 1001))';
data1 = sin(linspace(0, 2*pi, 1001) + pi/4)';
data2 = sin(linspace(0, 2*pi*10, 1001))';
```

The above waveform contains sin(0) and sin(2*pi). To repeat the waveform coherently, omit the final point.

```
data0(end) = [];
data1(end) = [];
data2(end) = [];
```

At a generation rate of 10000 scans per second, you can expect to observe `data0` and `data1` as 10 Hz sine waves and `data2` as a 100 Hz sine wave.

```
subplot(3,1,1)
plot(data0)
title('data0')
grid on
subplot(3,1,2)
plot(data1)
title('data1')
grid on;
subplot(3,1,3)
plot(data2)
title('data2')
grid on;
```



Queue the Output Data and Start Background Generation

Before starting a continuous generation, `preload` half a second of data. Use `start` to initiate the generation and return control to the command line immediately, allowing you to do other operations in MATLAB while the generation is running in the background.

```
preload(dq,repmat([data0, data1, data2], 5, 1));
start(dq, "repeatoutput")
```

Use `pause` in a loop to monitor the number of scans output by the hardware for the duration of the generation.

```
t = tic;
while toc(t) < 1.0
    pause(0.1)
    fprintf("While loop: scans output by hardware = %d\n", dq.NumScansOutputByHardware)
end

fprintf("Generation has terminated with %d scans output by hardware\n", dq.NumScansAcquired);
```

```
While loop: scans output by hardware = 1109
While loop: scans output by hardware = 2089
While loop: scans output by hardware = 3100
While loop: scans output by hardware = 4095
While loop: scans output by hardware = 5093
While loop: scans output by hardware = 6094
While loop: scans output by hardware = 7082
While loop: scans output by hardware = 8082
```

```
While loop: scans output by hardware = 9088
While loop: scans output by hardware = 10099
Generation has terminated with 0 scans output by hardware
```

Stop the Continuous Background Generation

Background generation runs simultaneously with other operations in MATLAB. Explicitly call `stop` to end the background generation.

```
stop(dq)
```

Generate Output Data Dynamically Using MATLAB Functions

To dynamically generate the output data using a MATLAB function, assign the function to the `ScansRequiredFcn` of the DataAcquisition. The code below is functionally equivalent to `'repeatoutput'`

```
dq.ScanRequiredFunction = (src,evt) write(src, repmat([data0, data1, data2],
5, 1));

start(dq, "continuous")
```

Simultaneously Acquire Data and Generate Signals

This example shows how to acquire and generate data using two National Instruments modules operating at the same time.

Create a DataAcquisition

Use `daq` to create a DataAcquisition

```
dq = daq("ni")
```

```
dq = 

DataAcquisition using National Instruments hardware:

                     Running: 0
                        Rate: 1000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Set up Hardware

This example uses a compactDAQ chassis NI c9178 with NI 9205 (cDAQ1Mod1 - 4 analog input channels) module and NI 9263 (cDAQ1Mod2 - 4 analog output channels) module. Use `daqlist` to obtain more information about connected hardware.

The analog output channels are physically connected to the analog input channels so that the acquired data is the same as the data generated from the analog output channel.

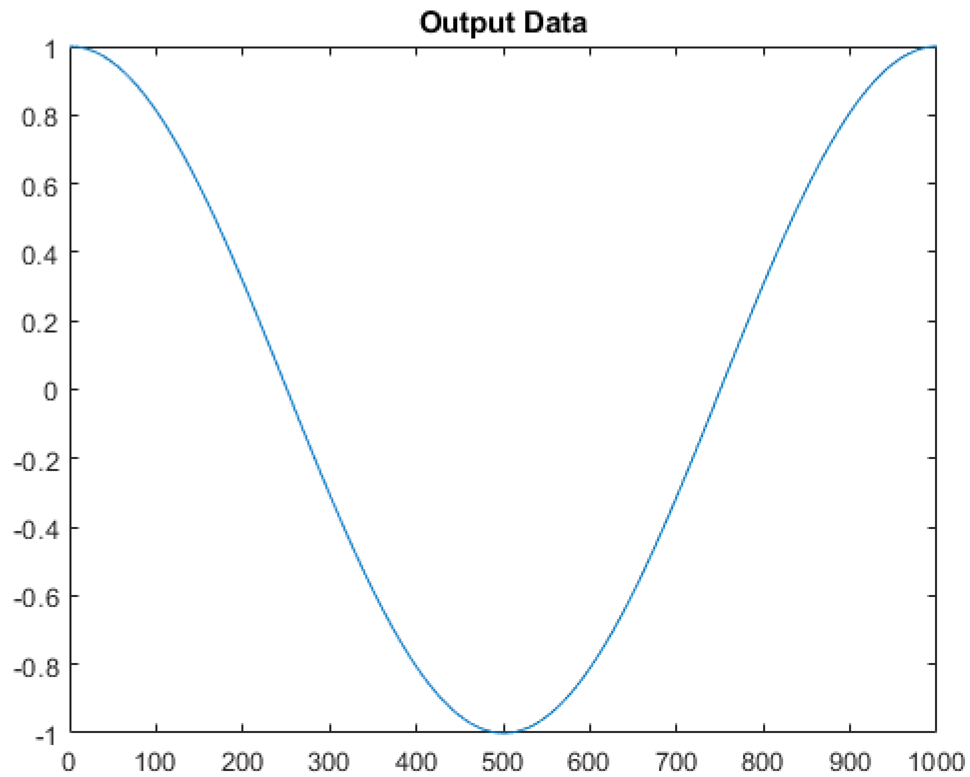
Add an Analog Input Channel and an Analog Output Channel

Use `addinput` to add an analog input voltage channel. Use `addoutput` to add an analog output voltage channel.

```
addinput(dq, "cDAQ1Mod1", "ai0", "Voltage")
addoutput(dq, "cDAQ1Mod2", "ao0", "Voltage")
```

Create and Plot the Output Signal

```
output = cos(linspace(0,2*pi,1000)');
plot(output);
title("Output Data");
```



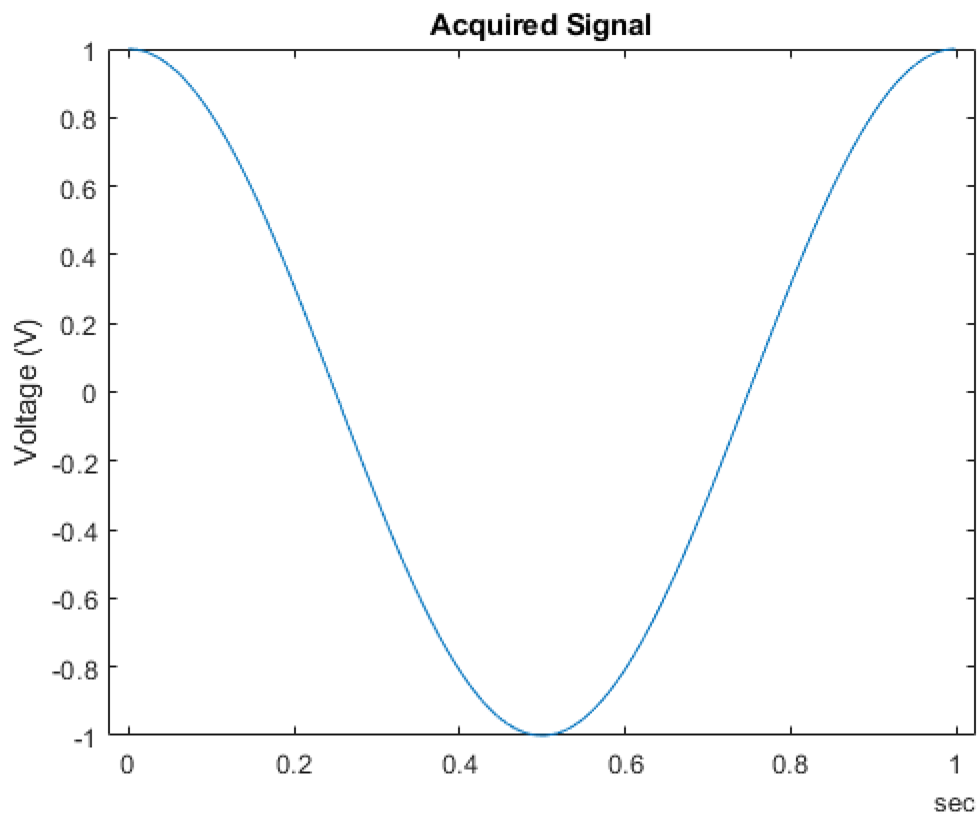
Generate and Acquire Data

Use `readwrite` to generate and acquire scans simultaneously.

```
data1 = readwrite(dq, output);
```

Plot the Acquired Data

```
plot(data1.Time, data1.Variables);
ylabel("Voltage (V)")
title("Acquired Signal");
```

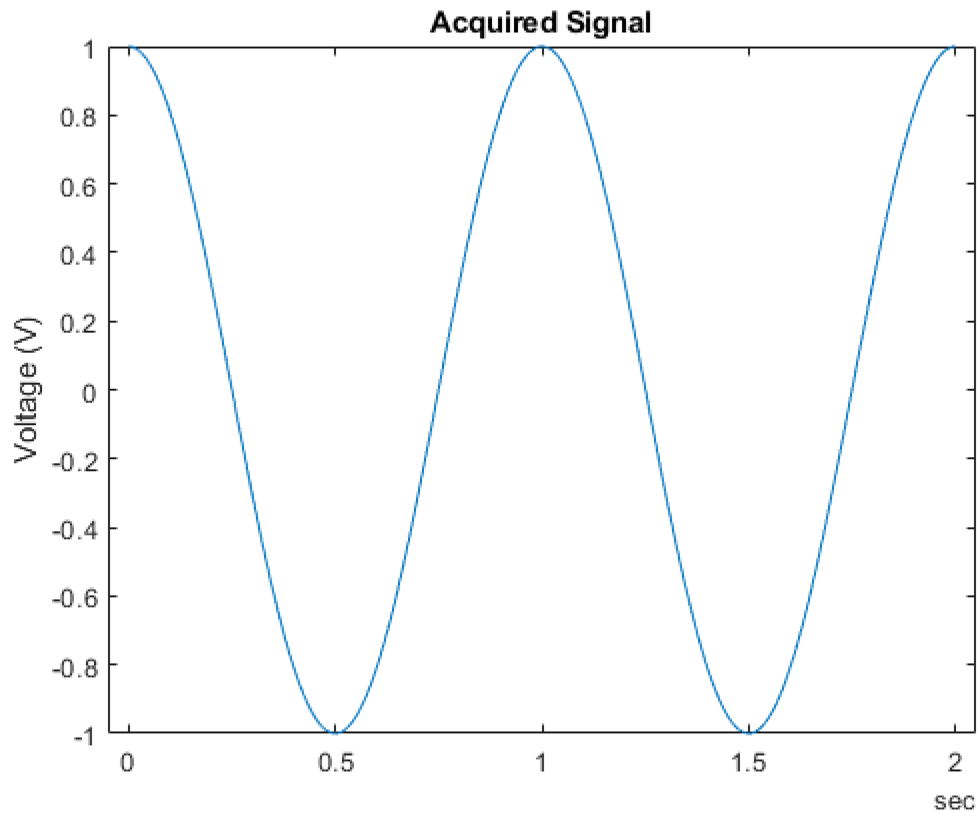


Generate and Acquire Date for Twice the Previous Duration

```
data2 = readwrite(dq, [output; output]);
```

Plot the Acquired Data

```
plot(data2.Time, data2.Variables);
ylabel("Voltage (V)")
title("Acquired Signal");
```


Acquired Signal
Voltage (V)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
0.5
1
1.5
2
sec

Log Analog Input Data to a File Using NI Devices

This example shows how to save data acquired in the background to a file.

Create a DataAcquisition with Analog Input Channels

Create a DataAcquisition and add two analog input channels with `Voltage` measurement type. For this example use a National Instruments® X Series data acquisition device, NI PCIe-6363 card with ID `Dev1`.

```
d = daqlist("ni")
```

```
d =

  12×4 table

     DeviceID             Description                   Model             DeviceInfo
    ___________    ___________________________________    ______________    ____________________

    "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"        [1×1 daq.DeviceInfo]
    "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"        [1×1 daq.DeviceInfo]
    "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"    [1×1 daq.DeviceInfo]
    "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
    "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"      [1×1 daq.DeviceInfo]
```

```
deviceInfo = d{9, "DeviceInfo"}
```

```
deviceInfo =

ni: National Instruments PCIe-6363 (Device ID: 'Dev1')
   Analog input supports:
      7 ranges supported
      Rates from 0.0 to 2000000.0 scans/sec
      32 channels ('ai0' - 'ai31')
      'Voltage' measurement type

   Analog output supports:
      -5.0 to +5.0 Volts,-10 to +10 Volts ranges
      Rates from 0.0 to 2000000.0 scans/sec
      4 channels ('ao0','ao1','ao2','ao3')
      'Voltage' measurement type

   Digital IO supports:
      39 channels ('port0/line0' - 'port2/line6')
      'InputOnly','OutputOnly','Bidirectional' measurement types

   Counter input supports:
```

```
      Rates from 0.1 to 100000000.0 scans/sec
      4 channels ('ctr0','ctr1','ctr2','ctr3')
      'EdgeCount','PulseWidth','Frequency','Position' measurement types

   Counter output supports:
      Rates from 0.1 to 100000000.0 scans/sec
      4 channels ('ctr0','ctr1','ctr2','ctr3')
      'PulseGeneration' measurement type
```

```
dq = daq("ni");
addinput(dq, "Dev1", 0:1, "Voltage");
dq.Channels
```

```
ans =

    Index    Type     Device     Channel      Measurement Type            Range              Name
    _____    ____    ______    _______    ________________    __________________    __________

      1      "ai"    "Dev1"     "ai0"      "Voltage (Diff)"    "-10 to +10 Volts"    "Dev1_ai0"
      2      "ai"    "Dev1"     "ai1"      "Voltage (Diff)"    "-10 to +10 Volts"    "Dev1_ai1"
```

Create a Log File

Create the file `log.bin` and open it. The file identifier is used to write to the file.

```
fid1 = fopen("log.bin","w");
```

Set the ScansAvailableFcn

During a background acquisition, the DataAcquisition can be directed to handle acquired data in a specified way using the `ScansAvailableFcn` property.

```
dq.ScansAvailableFcn = @(src, evt) logData(src, evt, fid1);
```

Acquire Data in the Background

Use `start` to acquire data for five seconds.

```
start(dq, "Duration", seconds(5))
```

During normal operation, other MATLAB commands can execute during this acquisition. For this example, use `pause` in a loop to monitor the number of scans acquired for the duration of the acquisition.

```
while dq.Running
    pause(0.5)
    fprintf("While loop: Scans acquired = %d\n", dq.NumScansAcquired)
end

fprintf("Acquisition has terminated with %d scans acquired\n", dq.NumScansAcquired);
```

Close the Log File

```
fclose(fid1);
```

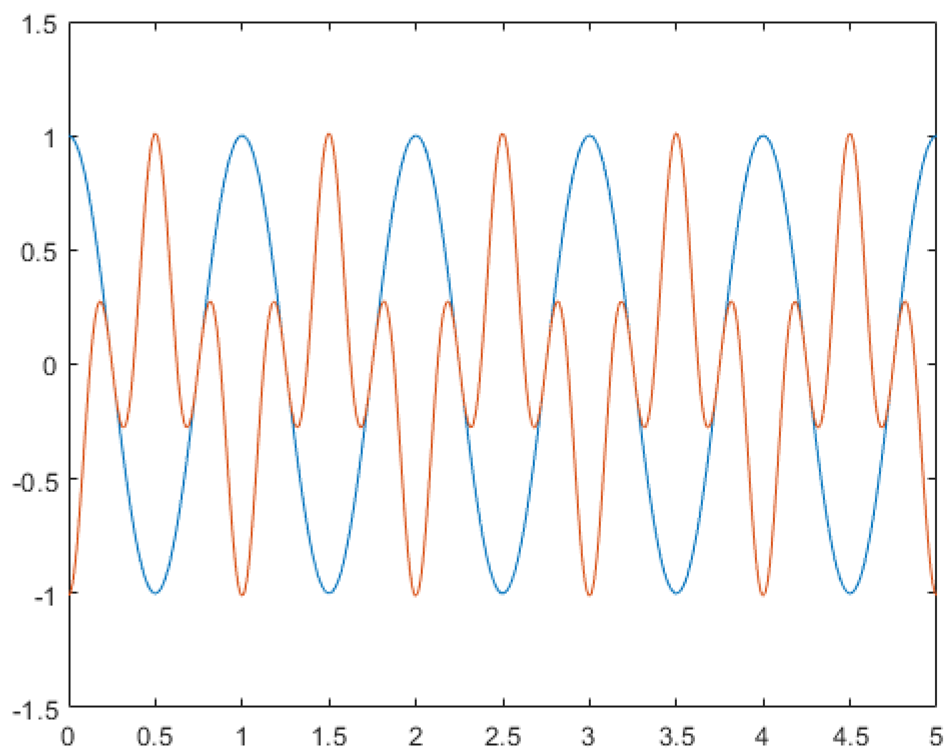
Load Data From the Log File

Load file contents as a 3-column matrix into `data`.

```
fid2 = fopen('log.bin','r');
[data,count] = fread(fid2,[3,inf],'double');
fclose(fid2);
```

Assign and Plot the Data

```
t = data(1,:);
ch = data(2:3,:);
plot(t, ch);
```




```
function logData(src, ~, fid)
[data, timestamps, ~] = read(src, src.ScansAvailableFcnCount, "OutputFormat", "Matrix");

data = [timestamps, data]' ;
fwrite(fid,data,'double');
end
```

```
While loop: Scans acquired = 500
While loop: Scans acquired = 1000
While loop: Scans acquired = 1500
While loop: Scans acquired = 2000
While loop: Scans acquired = 2500
While loop: Scans acquired = 3000
While loop: Scans acquired = 3500
```

```
While loop: Scans acquired = 4000
While loop: Scans acquired = 4500
While loop: Scans acquired = 5000
Acquisition has terminated with 5000 scans acquired
```

Getting Started Acquiring Data with Digilent Analog Discovery

This example shows you how to acquire voltage data at a rate of 300 kHz. The input waveform is a sine wave (10 Hz, 2 Vpp) provided by an external function generator.

Create a DataAcquisition for a Digilent Device

Discover Digilent devices connected to your system using `daqlist`.

```
daqlist("digilent")
dq = daq("digilent")
```

```
ans =

  1×4 table

    DeviceID                   Description                          Model                De
    ________    ____________________________________________    ____________________    _______

     "AD1"      "Digilent Inc. Analog Discovery 2 Kit Rev. C"    "Analog Discovery 2"    [1×1 dac

dq = 

DataAcquisition using Digilent Inc. hardware:

                     Running: 0
                        Rate: 10000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Add an Analog Input Channel

Add an analog input channel with device ID `AD1` and channel ID `1`. Set the measurement type to `Voltage`.

```
ch_in = addinput(dq, "AD1", "1", "Voltage");
```

Set DataAcquisition and Channel Properties

Set the acquisition rate to 300 kHz and the dynamic range of the incoming signal to -2.5 to 2.5 volts.

```
ch_in.Name = "AD1_1_in"
rate = 300e3;
dq.Rate = rate;
ch_in.Range = [-2.5 2.5];
```

```
ch_in =
```

```
    Index    Type    Device    Channel    Measurement Type          Range              Name
    _____    ____    ______    _______    ________________    __________________    __________

      1      "ai"    "AD1"       "1"      "Voltage (Diff)"    "-25 to +25 Volts"    "AD1_1_in"
```

Acquire a Single Sample

Acquire a single scan on-demand, displaying the data and trigger time.

```
[singleReading, startTime] = read(dq)
```

```
singleReading =

  timetable

    Time     AD1_1_in
    _____    ________

    0 sec    -0.37211


startTime =

  datetime

   21-Nov-2019 16:56:50.631
```

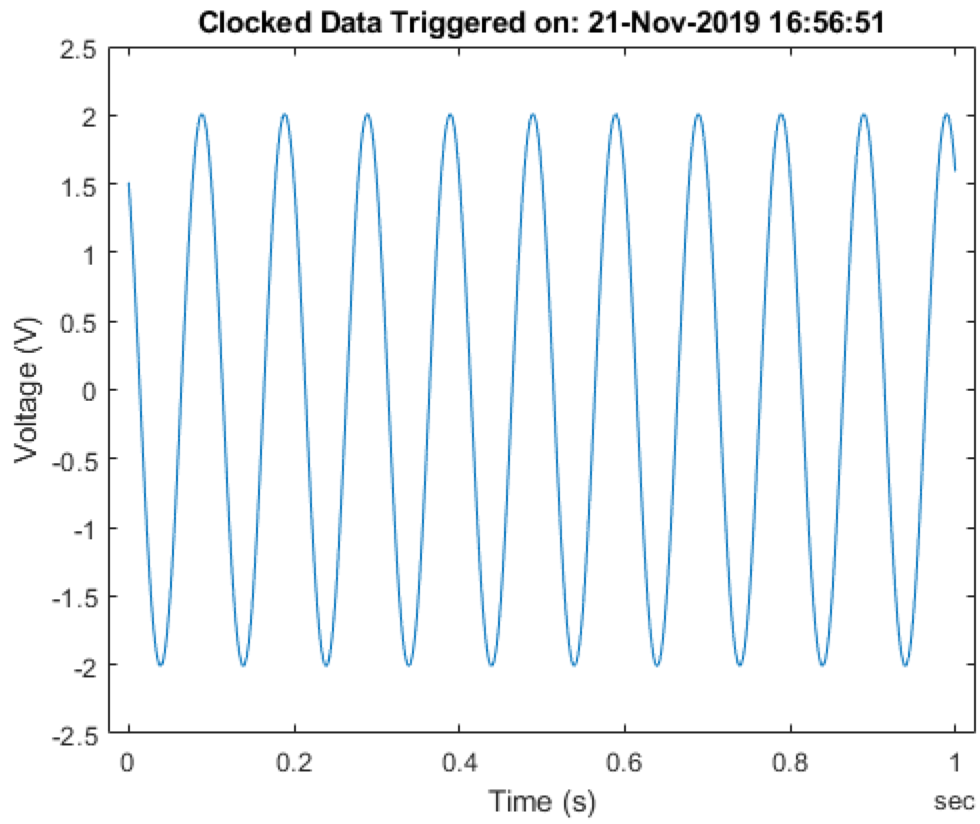
Acquire Timestamped Data

Acquire a set of clocked data for one second.

```
[data, startTime] = read(dq, seconds(1));
```

Plot Acquired Data

```
plot(data.Time, data.AD1_1_in);
xlabel('Time (s)');
ylabel('Voltage (V)');
title(['Clocked Data Triggered on: ' datestr(startTime)]);
```


Clocked Data Triggered on: 21-Nov-2019 16:56:51
Voltage (V)
Time (s)
sec
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
0.2
0.4
0.6
0.8
1

Getting Started Generating Data with Digilent Analog Discovery

This example shows you how to generate voltage data at a rate of 300 kHz.

Discovery Devices

Discover Digilent devices connected to your system using `daqlist`

```
daqlist("digilent")
```

```
ans =

  1×4 table

    DeviceID                 Description                            Model                De
    ________    ________________________________________    ______________________    ________

     "AD1"      "Digilent Inc. Analog Discovery 2 Kit Rev. C"    "Analog Discovery 2"    [1×1 dac
```

Create a DataAcquisition for a Digilent Device

```
dq = daq("digilent")
```

```
dq = 

DataAcquisition using Digilent Inc. hardware:

                     Running: 0
                        Rate: 10000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Add an Analog Output Channel

Add an analog output channel with device ID `AD1` and channel ID `1`. Set the measurement type to `Voltage`. By default, the voltage range of the output signal is -5.0 to +5.0 volts.

```
ch_out = addoutput(dq, "AD1", "1", "Voltage");
ch_out.Name = "AD1_1_out"
```

```
ch_out = 

    Index    Type    Device    Channel       Measurement Type              Range              Nar
    _____    ____    ______    _______    ______________________    ______________________    _____
```

```
     1      "ao"    "AD1"       "1"      "Voltage (SingleEnd)"    "-5.0 to +5.0 Volts"    "AD1_
```

Generate a Single Sample

Generate a single scan on-demand.

```
outVal = 2;
write(dq, outVal);
```

Set DataAcquisition Properties and Define the Output Waveform

Set the output scan rate to 300 kHz.

```
rate = 300e3;
dq.Rate = rate;

% Generate a 10 Hz sine-wave for half a second. The length of the
% output waveform and the specified output rate define the duration of
% the waveform (totalduration = numscans / rate).

f = 10;
totalduration = 1;
n = totalduration * rate;
t = (1:n)/rate;
output = sin(2*pi*f*t)';
```

Generate Data

```
write(dq, output);
```

Acquiring and Generating Data at the Same Time with Digilent Analog Discovery

This example shows you how to synchronously generate and acquire voltage data at a rate of 300 kHz.

Discovery Digilent Device

Discover Digilent devices connected to your system using `daqlist`

```
daqlist("digilent")
```

```
ans =

  1×4 table

    DeviceID                     Description                          Model               De
    ________    ____________________________________________    _____________________    _______

     "AD1"      "Digilent Inc. Analog Discovery 2 Kit Rev. C"    "Analog Discovery 2"    [1×1 da
```

Create a DataAcquisition for a Digilent Device

Discover Digilent devices connected to your system using `daqlist`

```
dq = daq("digilent")
```

```
dq = 

DataAcquisition using Digilent Inc. hardware:

                     Running: 0
                        Rate: 10000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Add an Analog Output Channel

Add an analog output channel using the listed Digilent device with ID `AD1`, channel ID `1`, and measurement type `Voltage`.

```
addoutput(dq, "AD1", "1", "Voltage");
addoutput(dq, "AD1", "2", "Voltage");
ch_out = dq.Channels(1:2);
ch_out(1).Name = "AD1_1_out";
ch_out(2).Name = "AD1_2_out"
```

```
ch_out =

    Index    Type    Device    Channel      Measurement Type              Range               Nar
    _____    ____    ______    _______    ______________________    ____________________    _____

      1      "ao"    "AD1"       "1"      "Voltage (SingleEnd)"    "-5.0 to +5.0 Volts"    "AD1_
      2      "ao"    "AD1"       "2"      "Voltage (SingleEnd)"    "-5.0 to +5.0 Volts"    "AD1_
```

Add an Analog Input Channel

Add an analog input channel with the same device and measurement type `Voltage`.

```
addinput(dq, "AD1", "1", "Voltage");
addinput(dq, "AD1", "2", "Voltage");
ch_in = dq.Channels(3:4);
ch_in(1).Name = "AD1_1_in";
ch_in(2).Name = "AD1_2_in"
```

```
ch_in =

    Index    Type    Device    Channel    Measurement Type          Range               Name
    _____    ____    ______    _______    ________________    __________________    __________

      1      "ai"    "AD1"       "1"      "Voltage (Diff)"    "-25 to +25 Volts"    "AD1_1_in"
      2      "ai"    "AD1"       "2"      "Voltage (Diff)"    "-25 to +25 Volts"    "AD1_2_in"
```

Set DataAcquisition Properties and Define an Output Waveform

Set the generation rate to 300 kHz.

```
rate = 300e3;
dq.Rate = rate;

% Specify a 10 Hz sine wave for 1 second.
f = 10;
totalduration = 1;
n = totalduration * rate;
t = (1:n)/rate;
output = sin(2*pi*f*t)';
```

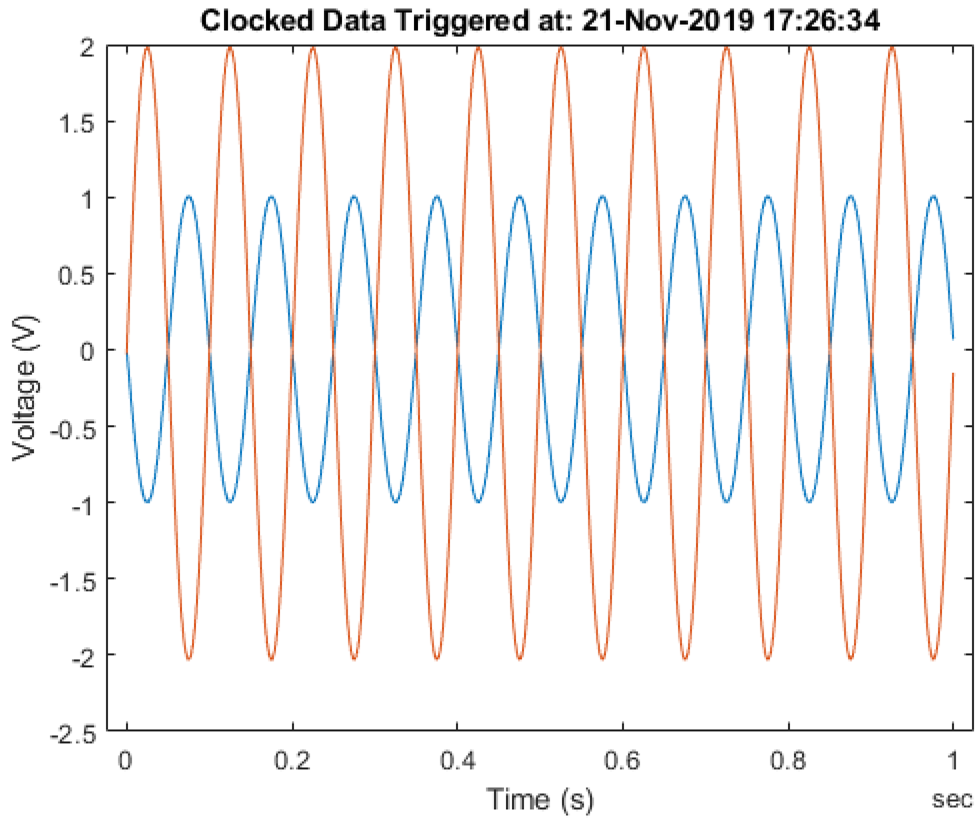
Generate and Acquire Data

Generate a sine wave with amplitude 1 V on channel 1 and amplitude 2 V on channel 2 and acquire timestamped data at the same rate.

```
[data, startTime] = readwrite(dq, [output 2*output]);
```

Plot Acquired Data

```
plot(data.Time, data.AD1_1_in, data.Time, data.AD1_2_in);
xlabel('Time (s)');
ylabel('Voltage (V)');
title(['Clocked Data Triggered at: ' datestr(startTime)])
```


Clocked Data Triggered at: 21-Nov-2019 17:26:34
Voltage (V)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
0.2
0.4
0.6
0.8
1
Time (s)
sec

Generate Standard Periodic Waveforms Using Digilent Analog Discovery

Use function generator channels to generate a 1 kHz sinusoidal waveform, and record data at the same time, using an analog input channel.

Discover Digilent Devices

Discover Digilent devices connected to your system using `daqlist`

```
daqlist("digilent")
dq = daq("digilent")
```

```
ans =

  1×4 table

    DeviceID                 Description                          Model              De
    ________    _____________________________________________    ____________________    ________

     "AD1"      "Digilent Inc. Analog Discovery 2 Kit Rev. C"    "Analog Discovery 2"    [1×1 dac

dq = 

DataAcquisition using Digilent Inc. hardware:

                     Running: 0
                        Rate: 10000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Add a Function Generator Channel

Add a function generator channel with device ID `AD1` and channel ID `1`. Set the waveform type to `Sine`.

```
ch_fgen = addoutput(dq, "AD1", "1", "Sine");
```

Set Channel Properties

Set channel gain to 5 (sets the amplitude of the sinusoid to 5 V). Assign the gain to a variable.

```
ch_fgen.Name = "AD1_1_fgen"
gain = 5;
ch_fgen.Gain = gain;
```

```
ch_fgen =
```

```
    Index     Type     Device     Channel     Measurement Type          Range              Name
    _____    ______    ______    _______    ________________    ____________________    _________

      1      "fgen"    "AD1"       "1"           "Sine"         "-5.0 to +5.0 Volts"    "AD1_1_fg
```

Set the signal frequency to 1 kHz

```
ch_fgen.Frequency = 1000;
```

Add an Analog Input Channel

Add an analog input channel with device ID `AD1` and channel ID `1`. Set the measurement type to `Voltage`.

```
ch_in = addinput(dq, "AD1", "1", "Voltage");
ch_in.Name = "AD1_1_in"
```

```
ch_in =

    Index    Type    Device    Channel    Measurement Type          Range              Name
    _____    ____    ______    _______    ________________    __________________    __________

      1      "ai"    "AD1"       "1"      "Voltage (Diff)"    "-25 to +25 Volts"    "AD1_1_in"
```

Set DataAcquisition Properties

Acquire data at a higher scan rate than the highest frequency in the generated waveform.

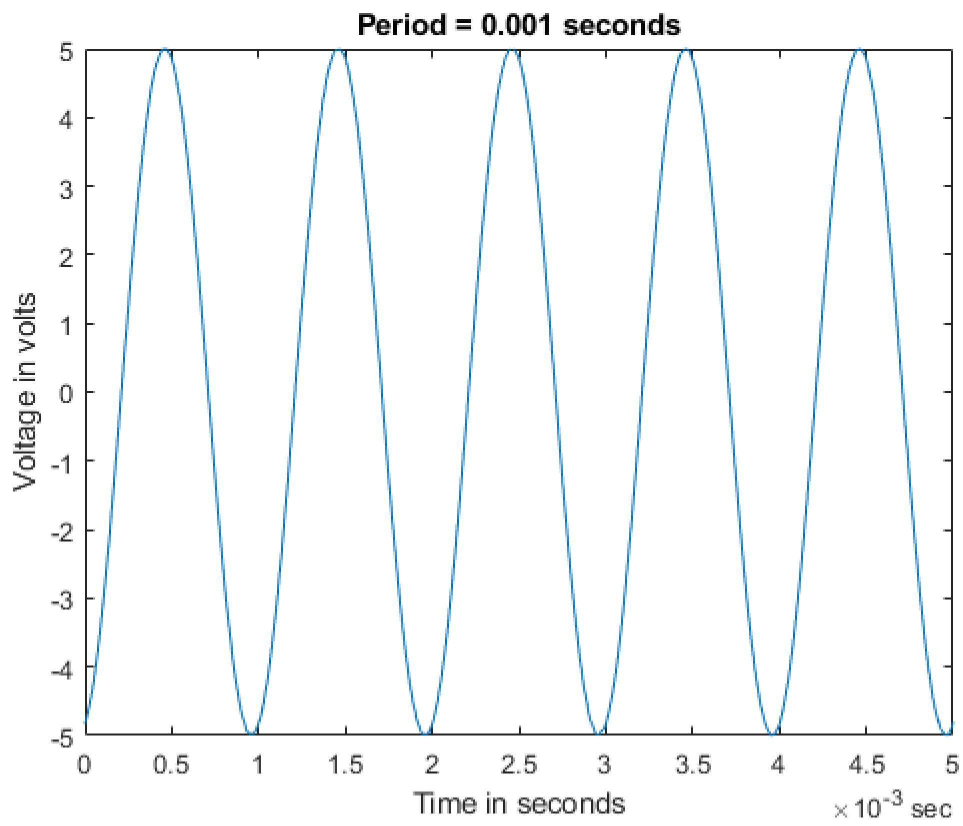
```
dq.Rate = 100 * ch_fgen.Frequency;
```

Generate a Periodic Waveform and Record Input

```
[data, startTime] = read(dq, seconds(1));
```

Plot Data

```
period = 1/ch_fgen.Frequency;
plot(data.Time, data.AD1_1_in);
xlabel('Time in seconds');
ylabel('Voltage in volts');
title(['Period = ', num2str(period), ' seconds'])
xlim([seconds(0) seconds(5*period)]);
ylim([-gain gain]);
```


Period = 0.001 seconds
Voltage in volts
Time in seconds
×10^-3 sec
5
4
3
2
1
0
-1
-2
-3
-4
-5
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Generate Arbitrary Periodic Waveforms Using Digilent Analog Discovery

Use function generator channels to generate an arbitrary 1kHz waveform function, and record data at the same time, using an analog input channel.

Discover Digilent Devices

Discover Digilent devices connected to your system using `daqlist`.

```
daqlist("digilent")
dq = daq("digilent")
```

```
ans =

  1×4 table

    DeviceID                  Description                          Model                 De
    ________    _____________________________________________    ______________________    ________

     "AD1"      "Digilent Inc. Analog Discovery 2 Kit Rev. C"    "Analog Discovery 2"    [1×1 dac
```

```
dq = 

DataAcquisition using Digilent Inc. hardware:

                     Running: 0
                        Rate: 10000
           NumScansAvailable: 0
            NumScansAcquired: 0
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Add a Function Generator Channel

Add a function generator channel with device ID `AD1` and channel ID `1`. Set the waveform type to `Arbitrary`. The voltage range of the output signal is -5.0 to +5.0 volts.

```
ch_fgen = addoutput(dq, "AD1", "1", "Arbitrary");
ch_fgen.Name = "AD1_1_fgen"
```

```
ch_fgen = 

    Index     Type      Device    Channel    Measurement Type            Range                Name
    _____    ______    ______    _______    ________________    ____________________    ________

      1      "fgen"    "AD1"      "1"        "Arbitrary"       "-5.0 to +5.0 Volts"    "AD1_1_fg
```

Define a Sum of Sinusoids as the Output Waveform

The function generator produces periodic outputs by repeatedly generating the contents of its buffer (4096 points). A waveform is constructed to fill this buffer without repetition.

```
buffersize = 4096;
len = buffersize + 1;

f0 = 1;
f1 = 1 * f0;
f2 = 2 * f0;
f3 = 3 * f0;

waveform  = sin(linspace(0, 2*pi*f1, len)) + ...
            sin(linspace(0, 2*pi*f2, len)) + ...
            sin(linspace(0, 2*pi*f3, len));

waveform(end) = [];
```

Assign the Waveform Data and Set Waveform Frequency

```
frequency = 1000;
ch_fgen.WaveformData = waveform;
ch_fgen.Frequency = frequency;
```

Add an Analog Input Channel

Add an analog input channel with device ID `AD1` and channel ID `1`. Set the measurement type to `Voltage`.

```
ch_in = addinput(dq, "AD1", "1", "Voltage");
ch_in.Name = "AD1_1_in"
```

```
ch_in = 

    Index    Type    Device    Channel    Measurement Type          Range             Name   
    _____    ____    ______    _______    ________________    __________________    __________

      1      "ai"    "AD1"       "1"      "Voltage (Diff)"    "-25 to +25 Volts"    "AD1_1_in"
```

Define the Acquisition Scan Rate

Acquire data at a higher scan rate than the highest frequency in the generated waveform (oversampling).

```
oversamplingratio = 50;
Fn = 2 * frequency;
Fs = oversamplingratio * Fn;
dq.Rate = Fs;
```

Generate a Periodic Waveform and Record Input

```
data = read(dq, seconds(3));
```

Define Plot Parameters

```
k = 5;
width = 750;
```

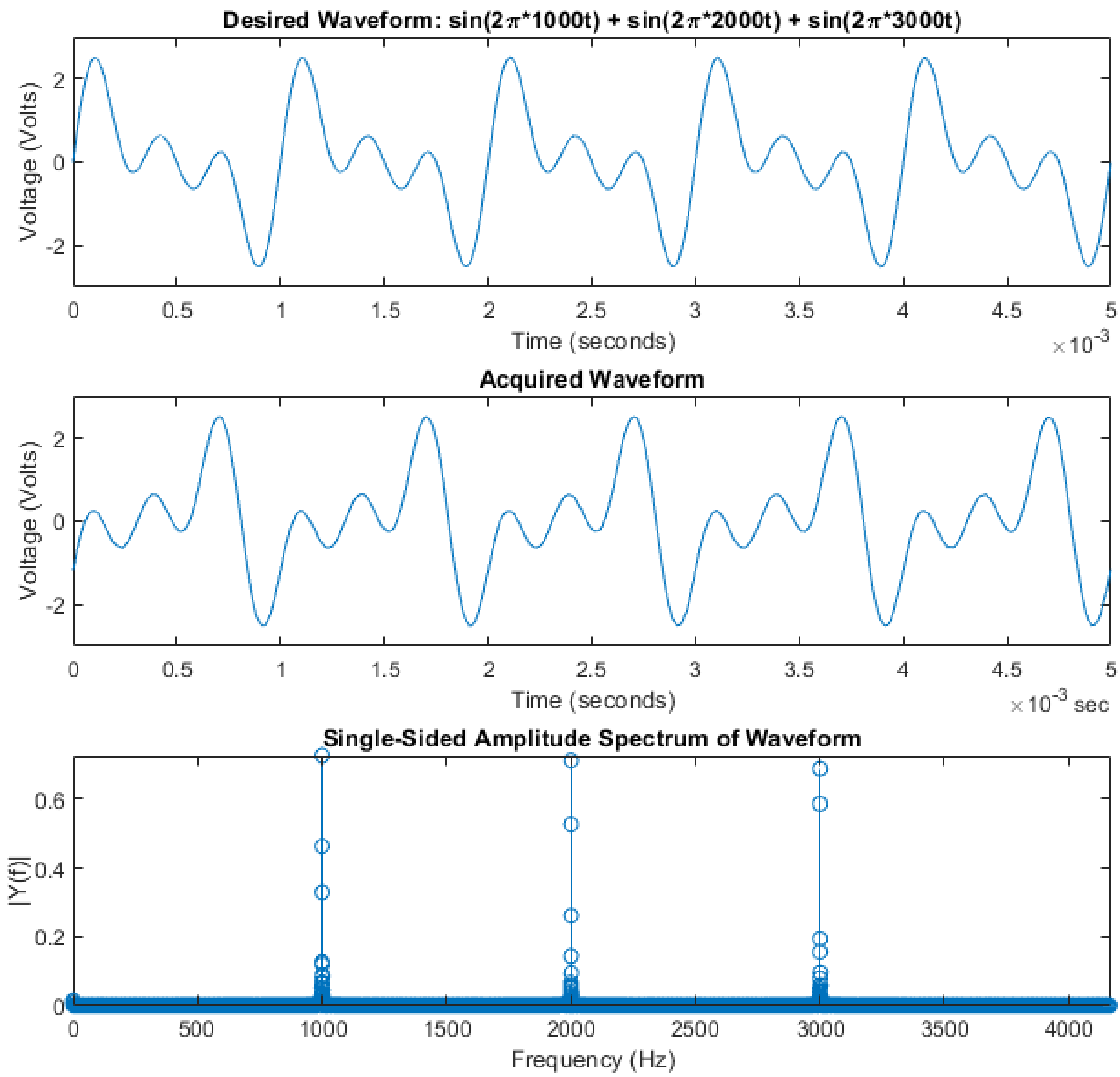
```
height = 750;
period = 1/frequency;
numperiods = k * period;
maxamplitude = 3*ch_fgen.Gain;

wavedesired = repmat(waveform', k, 1);
tsamples = linspace(0, numperiods, k * buffersize)';
```

Define FFT Parameters

```
L = 2 * oversamplingratio * buffersize;
NFFT = 2^nextpow2(L);
Y = fft(data.AD1_1_in, NFFT)/L;
f0 = (Fs/2) * linspace(0, 1, NFFT/2 + 1);
```

Plot Waveforms

```
plotScaleFactor = 12;
plotRange = NFFT/2; % Plot is symmetric about NFFT
plotRange = floor(plotRange / plotScaleFactor);

Yplot = Y(1:plotRange);
fplot = f0(1:plotRange);

fig = figure;

% Plot Desired Waveform
subplot(311)
plot(tsamples, wavedesired);
xlabel('Time (seconds)');
ylabel('Voltage (Volts)');
title('Desired Waveform: sin(2\pi*1000t) + sin(2\pi*2000t) + sin(2\pi*3000t)');
xlim([0 numperiods]);
ylim([-maxamplitude maxamplitude]);

% Plot Acquired Waveform
subplot(312)
plot(data.Time, data.AD1_1_in);
xlabel('Time (seconds)');
ylabel('Voltage (Volts)');
title('Acquired Waveform');
xlim([seconds(0) seconds(numperiods)]);
ylim([-maxamplitude maxamplitude]);

% Plot Single-Sided Amplitude Spectrum
subplot(313)
stem(fplot, 2*abs(Yplot));
title('Single-Sided Amplitude Spectrum of Waveform')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
axis tight

% Make Graph Larger
outpos = get(fig, 'OuterPosition');
set(fig, 'OuterPosition', [outpos(1)-125 outpos(2)-375 width height]);
```


Desired Waveform: sin(2π*1000t) + sin(2π*2000t) + sin(2π*3000t)
Voltage (Volts)
Time (seconds)
×10^-3
Acquired Waveform
Voltage (Volts)
Time (seconds)
×10^-3 sec
Single-Sided Amplitude Spectrum of Waveform
|Y(f)|
Frequency (Hz)

Acquire Continuous Audio Data

This example shows how to set up a continuous audio acquisition using a microphone.

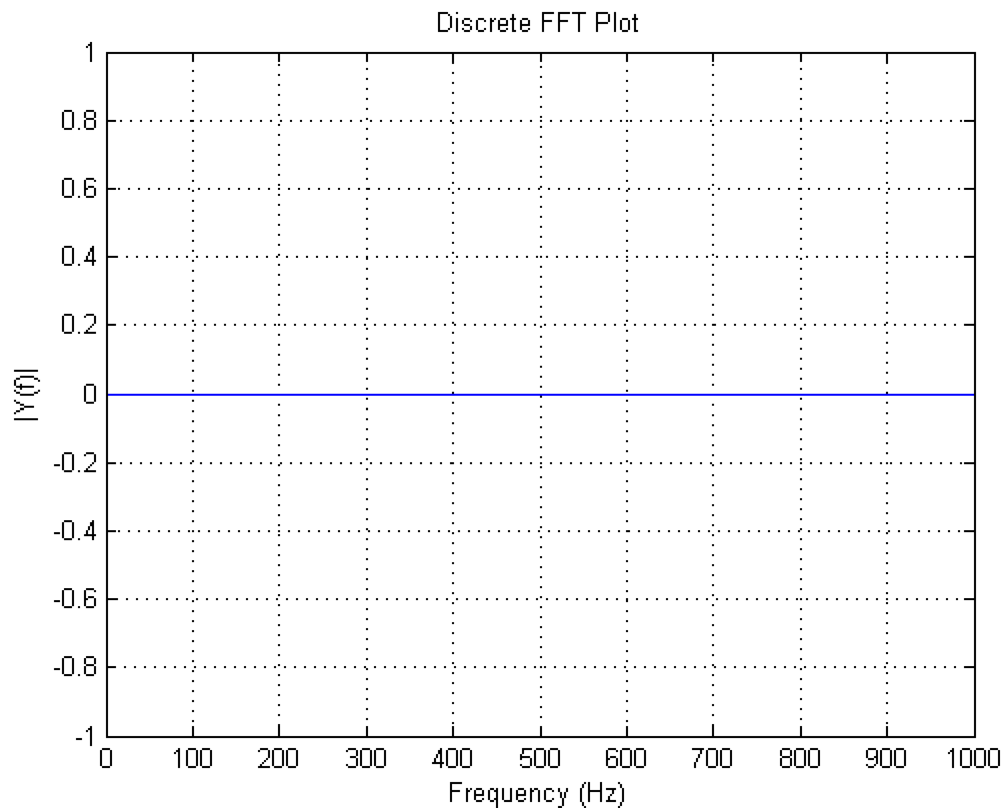
Create a DataAcquisition

Create a DataAcquisition with `directsound` as the vendor and add an audio input channel to it using `addinput`.

```
dq = daq("directsound");
addinput(dq,"Audio0",1,"Audio");
```

Set Up the FFT Plot

```
hf = figure;
hp = plot(zeros(1000,1));
T = title('Discrete FFT Plot');
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
grid on;
```



Set ScansAvailableFcn

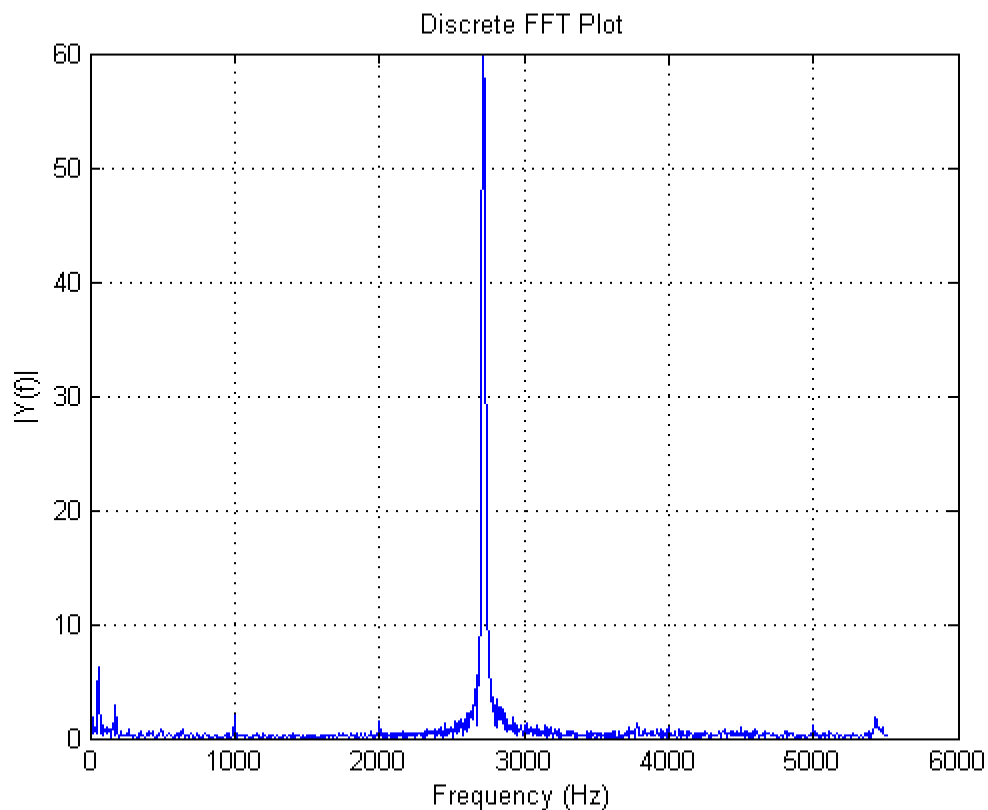
Update the figure with the FFT of the live input signal by setting the `ScansAvailableFcn`.

```
dq.ScansAvailableFcn = @(src, evt) continuousFFT(src, hp);
```

Start Acquisition

The figure updates, for 10 seconds, as the microphone is used.

```
start(dq,"Duration",seconds(10));
figure(hf);
```




```
function continuousFFT(daqHandle, plotHandle)
% Calculate FFT(data) and update plot with it.
data = read(daqHandle, daqHandle.ScansAvailableFcnCount, "OutputFormat", "Matrix");
Fs = daqHandle.Rate;

lengthOfData = length(data);
% next closest power of 2 to the length
nextPowerOfTwo = 2 ^ nextpow2(lengthOfData);

plotScaleFactor = 4;
% plot is symmetric about n/2
plotRange = nextPowerOfTwo / 2;
plotRange = floor(plotRange / plotScaleFactor);

yDFT = fft(data, nextPowerOfTwo);

h = yDFT(1:plotRange);
abs_h = abs(h);

% Frequency range
```

```
freqRange = (0:nextPowerOfTwo-1) * (Fs / nextPowerOfTwo);
% Only plot up to n/2 (as other half is the mirror image)
gfreq = freqRange(1:plotRange);

% Update the plot
set(plotHandle, 'ydata', abs_h, 'xdata', gfreq);
drawnow
end
```

Generate Audio Signals

This example shows how to generate audio signals using a 5.1 channel sound system.

Load Audio Signal

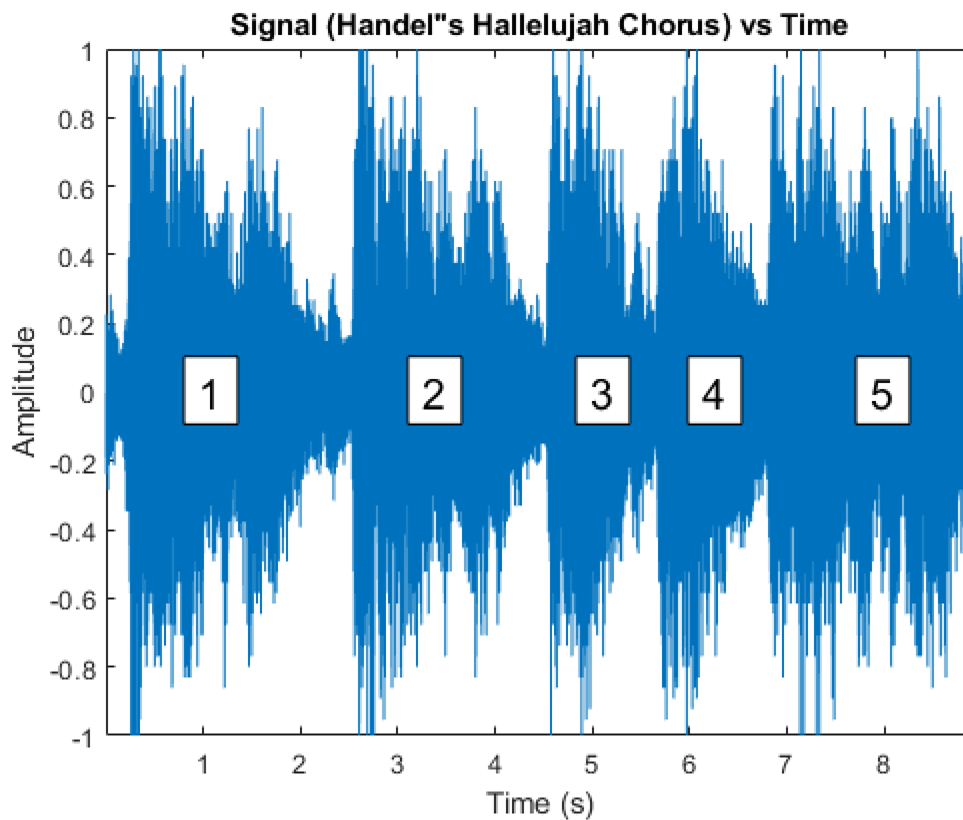
Load an audio file containing a sample of Handel's "Hallelujah Chorus."

```
load handel;
```

Plot Audio Signal

Plot the data to identify five distinct segments. Each segment represents a "Hallelujah" in the chorus. The segments are annotated as 1 to 5.

```
ly = length(y);
lspan = 1:ly;
t = lspan/Fs;

hf = figure;
plot(t,y./max(y))
axis tight;
title("Signal (Handel''s Hallelujah Chorus) vs Time");
xlabel("Time (s)");
ylabel("Amplitude");

markers = struct('xpos',[0.2,0.4,0.55,0.65,0.8],'string',num2str([1:5]'));
for i = 1:5,
    annotation(hf,'textbox',[markers.xpos(i) 0.48 0.048 0.080],'String', markers.string(i),'Backg
end
```



Create a DataAcquisition and Add Audio Output Channels

This example uses a 5.1 channel sound system with device ID `'Audio2'`.

1. Create a DataAcquisition with `directsound` as the vendor and add an audio output channel to it.

```
dd = daq("directsound");
nch = 6;
addoutput(dd, "Audio2", 1:nch, "Audio");
```

2. Update the generation scan rate to match the audio sampling rate.

```
dd.Rate = Fs;
```

3. Generate audio signals (Handel's "Hallelujah Chorus"). "Hallelujah" should be voiced five times, one for each segment depicted in the figure on all channels of the speaker system.

```
write(dd, repmat(y,1,nch));
```

4. Close the figure.

```
close(hf);
```

Generating Multichannel Audio

This example shows how to set up continuous audio generation using multiple audio channels. The signal, a sample of Handel's "Hallelujah Chorus", is broken up into contiguous segments and played back in two parts. The first part of the example plays each segment on a single speaker and a subwoofer. The second part plays each segment on a different set of speakers (a choir of voices).

Load Audio Data

Load Handel's "Hallelujah".

Load variables:

- `y` representing the Hallelujah waveform
- `Fs` representing the sampling frequency

```
load handel;
```

Create a Data Acquisition

Create a DataAcquisition object using `directsound` as the vendor ID.

```
dq = daq("directsound")
```

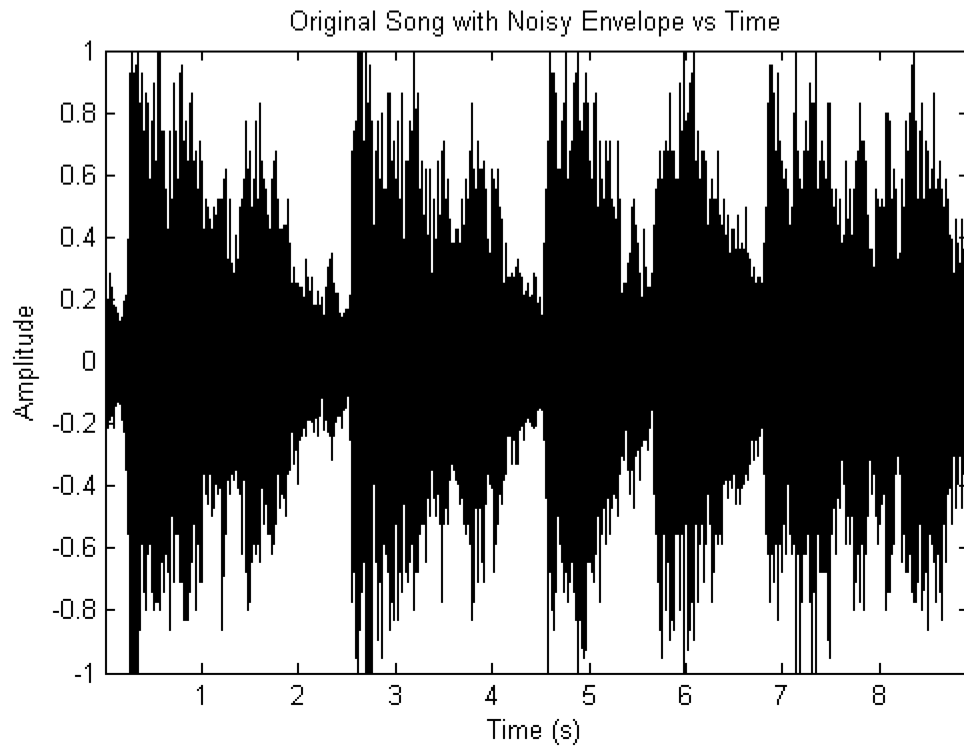
Add Channels and Adjust Generation Scan Rate to Match the Audio Sampling Frequency

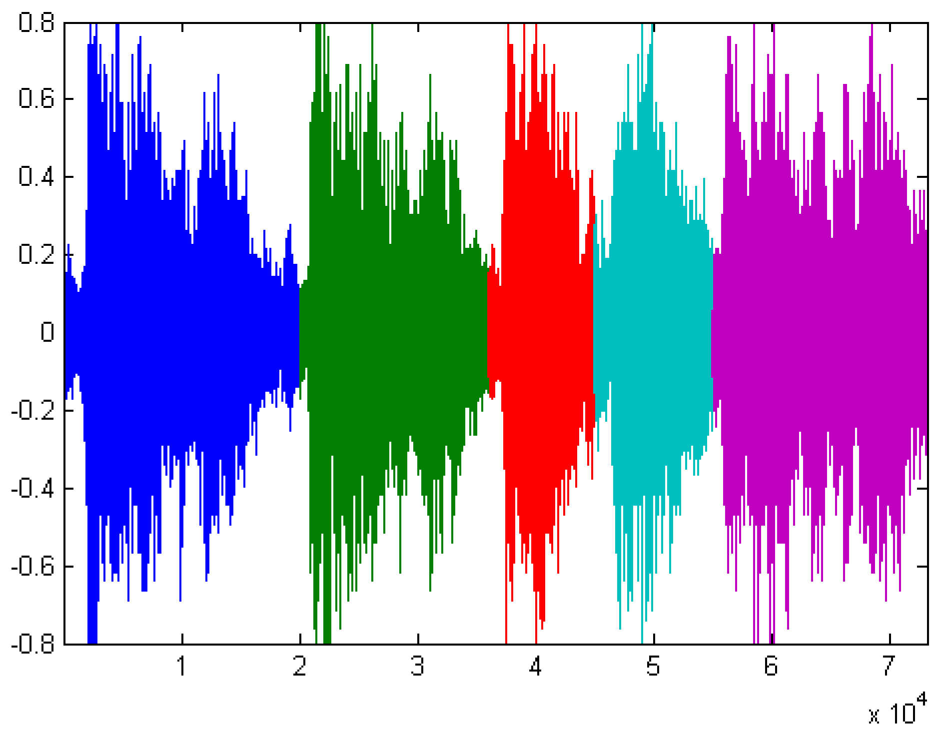
Add six audio output channels and set the generation scan rate to the audio sampling rate.

```
addoutput(dq,"Audio7",1:6,"Audio");
dq.Rate = Fs;
```

Plot Audio Data

Visually identify audio segments that correspond to each "Hallelujah" in the chorus and select sample numbers at which these segments start and stop. Each color in the plot corresponds to a different segment of the chorus.


Original Song with Noisy Envelope vs Time
Amplitude
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
1
2
3
4
5
6
7
8
Time (s)


0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
1
2
3
4
5
6
7
x 10^4

Identify the End of Each Segment

Visually identify the segment boundaries and mark them.

```
segmentEnd = [20000, 36000, 45000, 55000, length(y)];
```

Define Speaker Parameters

Set up a selection of speakers in a cell array named `speakerselection` to play five segments of "Hallelujah" on six different speakers.

```
nspeakers = 6;
nspeakergroups = 5;
speakerselection = cell(1, nspeakergroups);
```

Assign Speakers to Groups

Each speaker selection specifies which speakers from the 5.1 channel speaker system play each audio segment (these assignments may vary for your speaker system). For the first part of the example, use single speakers paired with the sub-woofer (4).

- Speaker 1: Left-Front
- Speaker 2: Right-Front
- Speaker 3: Center
- Speaker 4: Sub-Woofer
- Speaker 5: Left-Rear
- Speaker 6: Right-Rear

```
speakerselection{1} = [4, 6];
% Segment 1; speakers 4 and 6
speakerselection{2} = [4, 5];
% Segment 2; speakers 4 and 5
speakerselection{3} = [1, 4];
% Segment 3; speakers 1 and 4
speakerselection{4} = [2, 4];
% Segment 4; speakers 2 and 4
speakerselection{5} = [3, 4];
% Segment 5; speakers 3 and 4

[singleChannelOutputs] = ...
    surroundSoundVoices(y, segmentEnd, nspeakers, nspeakergroups, speakerselection);
```

Write Single Channel Outputs

Write a sequence of single channel outputs and then `pause` before proceeding to the next section.

```
write(dq, singleChannelOutputs);
pause(3);
```

Assign Speakers to Groups

Each speaker selection specifies which speakers from the 5.1 channel speaker system play each audio segment (these assignments may vary for your speaker system). For the second part of the example, use groups of speakers. Note that the sub-woofer (4) is included in all speaker selections

- Speaker 1: Left-Front
- Speaker 2: Right-Front
- Speaker 3: Center
- Speaker 4: Sub-Woofer
- Speaker 5: Left-Rear
- Speaker 6: Right-Rear

```
speakerselection{1} = [4, 5, 6];          % Segment 1; speakers 4, 5, 6
speakerselection{2} = [1, 2, 4];          % Segment 2; speakers 1, 2, 4
speakerselection{3} = [3, 4];             % Segment 3; speakers 3, 4
speakerselection{4} = [1, 2, 3, 4];       % Segment 4; speakers 1, 2, 3, 4
speakerselection{5} = [1, 2, 3, 4, 5, 6]; % Segment 5; all speakers

[multiChannelOutput] = ...
    surroundSoundVoices(y, segmentEnd, nspeakers, nspeakergroups, speakerselection);
```

Write Multichannel Outputs

```
write(dq, multiChannelOutput);

function [multiChannelOutput] = surroundSoundVoices(audioOut, segmentEnds, numSpeakers, numSpeake
% Distribute contiguous segments of an output waveform to multiple groups
% of speakers in a one-to-one relationship. The input waveform is broken up
% into contiguous segments. Each segment is output by one and only one
% group of speakers, with each group being visited in turn.

% Break up the input waveform into segments to be played by various groups
% of speakers. In this demonstration, we would like to slowly add "voices"
% by incrementally having more speakers generate the output waveform.
% In particular, we will regard the output waveform as a contiguous
% sequence of segments (one segment per group of speakers). For example, if
% we have 3 groups of speakers, we can think of breaking up the output
% waveform into 3 segments: output = [s1 s2 s3]
% Speaker group 1 outputs: s1 0  0
% Speaker group 2 outputs: 0  s2 0
% Speaker group 3 outputs: 0  0  s3

multiChannelOutput = repmat(0.01, length(audioOut), numSpeakers);
startOfSegment = [1 (segmentEnds(1:end-1)+1)];

for i = 1:numSpeakerGroups
    speakergroup = speakerGroups{i};
    n = numel(speakergroup);
    for j = 1:n
        range = startOfSegment(i):segmentEnds(i);
        multiChannelOutput(range, speakergroup(j)) = audioOut(range);
    end
end

end
```

Capture Data with Software-Analog Triggering

This example shows how to implement a triggered data capture based on a trigger condition defined in software. Data Acquisition Toolbox provides functionality for hardware triggering a data acquisition object, for example starting acquisition from a DAQ device based on an external digital trigger signal (rising or falling edge). For some applications however, it is desirable to start capturing or logging data based on the analog signal being measured, allowing for capturing only the signal of interest out of a continuous stream of digitized measurement data (such as an audio recording when the signal level passes a certain threshold).

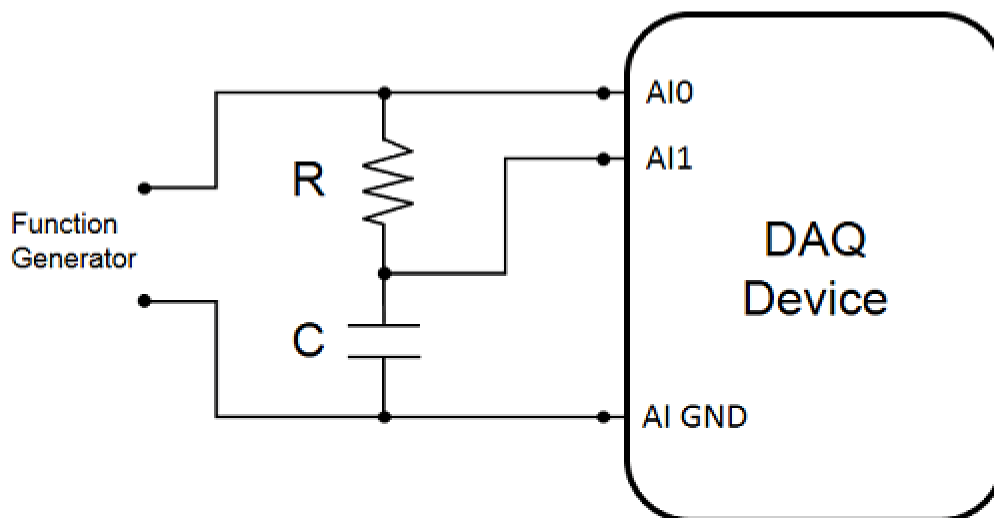
A custom graphical user interface (UI) is used to display a live plot of the data acquired in continuous mode, and allows you to input trigger parameters values for a custom trigger condition, which is based on the acquired analog input signal level and slope. Captured data is displayed in the interactive UI, and is saved to a MATLAB base workspace variable.

This example can be easily modified to instead use audio input channels with a DirectSound supported audio device.

The code is structured as a single program file, with a main function and several local functions.

Hardware Setup

- A DAQ device (such as NI USB-6218) with analog input channels, supported by the `DataAcquisition` interface in background acquisition mode.
- External signal connections to analog input channels. The data in this example represents measured voltages from a series resistor-capacitor (RC) circuit: total voltage across RC (in this example supplied by a function generator) is measured on channel AI0, and voltage across the capacitor is measured on channel AI1.



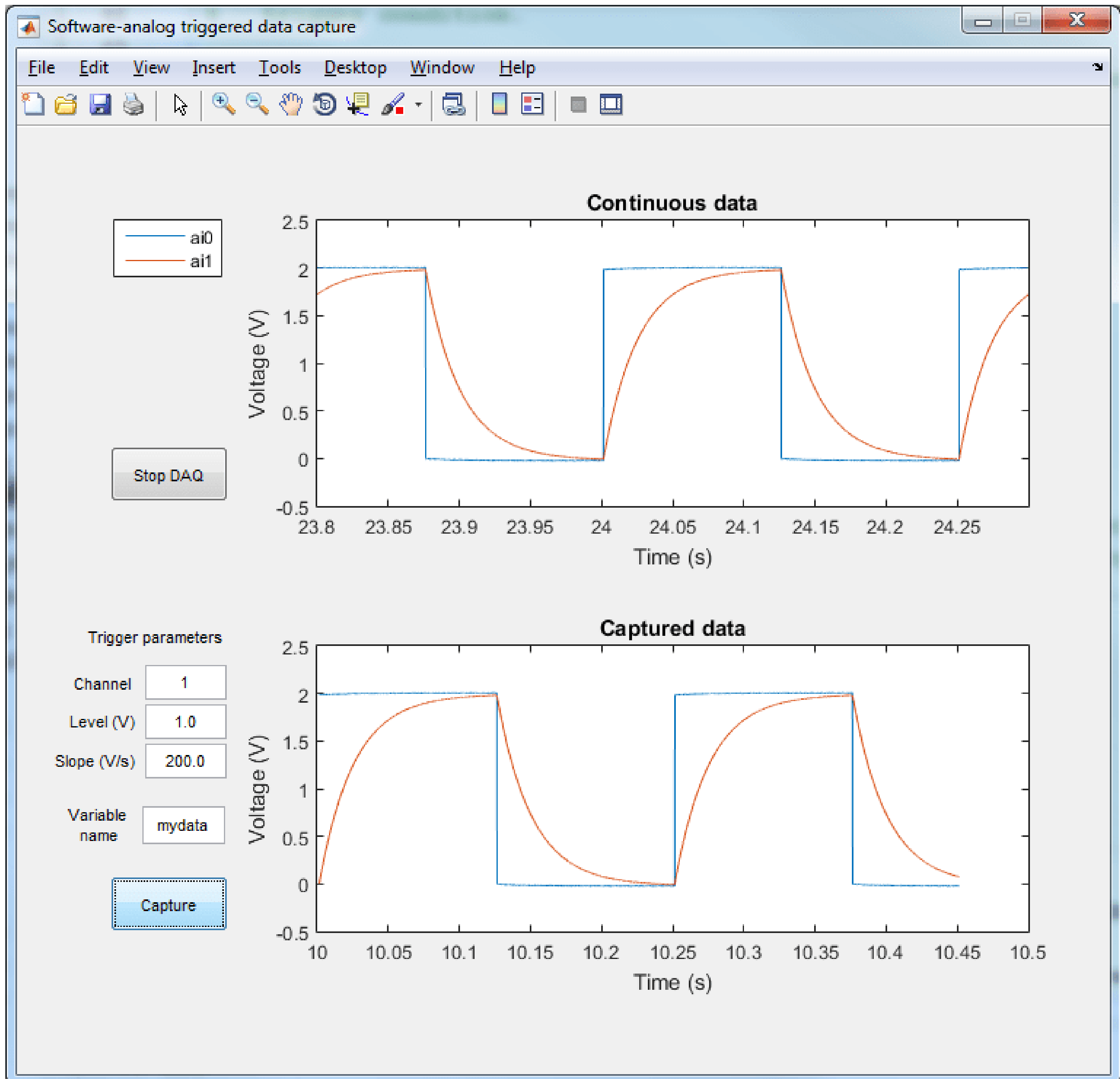
Configure Data Acquisition and Capture Parameters (Main Function)

Configure a data acquisition object with two analog input channels and set acquisition parameters. Background continuous acquisition mode provides the acquired data by calling a user defined

callback function (dataCapture) when ScansAvailable events occur. A custom graphical user interface (UI) is used for live acquired data visualization and for interactive data capture based on user specified trigger parameters.

```
function softwareAnalogTriggerCapture
%softwareAnalogTriggerCapture DAQ data capture using software-analog triggering
%   softwareAnalogTriggerCapture launches a user interface for live DAQ data
%   visualization and interactive data capture based on a software analog
%   trigger condition.

% Configure data acquisition object and add input channels
s = daq('ni');
ch1 = addinput(s, 'Dev1', 0, 'Voltage');
ch2 = addinput(s, 'Dev1', 1, 'Voltage');

% Set acquisition configuration for each channel
ch1.TerminalConfig = 'SingleEnded';
ch2.TerminalConfig = 'SingleEnded';
ch1.Range = [-10.0 10.0];
ch2.Range = [-10.0 10.0];

% Set acquisition rate, in scans/second
s.Rate = 10000;

% Specify the desired parameters for data capture and live plotting.
% The data capture parameters are grouped in a structure data type,
% as this makes it simpler to pass them as a function argument.

% Specify triggered capture timespan, in seconds
capture.TimeSpan = 0.45;

% Specify continuous data plot timespan, in seconds
capture.plotTimeSpan = 0.5;

% Determine the timespan corresponding to the block of samples supplied
% to the ScansAvailable event callback function.
callbackTimeSpan = double(s.ScansAvailableFcnCount)/s.Rate;
% Determine required buffer timespan, seconds
capture.bufferTimeSpan = max([capture.plotTimeSpan, capture.TimeSpan * 3, callbackTimeSpan * 3]);
% Determine data buffer size
capture.bufferSize =  round(capture.bufferTimeSpan * s.Rate);

% Display graphical user interface
hGui = createDataCaptureUI(s);

% Configure a ScansAvailableFcn callback function
% The specified data capture parameters and the handles to the UI graphics
% elements are passed as additional arguments to the callback function.
s.ScansAvailableFcn = @(src,event) dataCapture(src, event, capture, hGui);

% Configure a ErrorOccurredFcn callback function for acquisition error
% events which might occur during background acquisition
s.ErrorOccurredFcn = @(src,event) disp(getReport(event.Error));

% Start continuous background data acquisition
start(s, 'continuous')

% Wait until data acquisition object is stopped from the UI
```

```
while s.Running
    pause(0.5)
end

% Disconnect from hardware
delete(s)
end
```



Background Acquisition Callback Function

The dataCapture user-defined callback function is being called repeatedly, each time a ScansAvailable event occurs. With each callback function execution, the latest acquired data block and timestamps are added to a persistent FIFO data buffer, a continuous acquired data plot is updated, latest data is analyzed to check whether the trigger condition is met, and -- once capture is triggered and enough data has been captured for the specified timespan -- captured data is saved in a base workspace variable. The captured data is an N x M matrix corresponding to N acquired data scans, with the timestamps as the first column, and the acquired data corresponding to each channel as columns 2:M.

```
function dataCapture(src, ~, c, hGui)
%dataCapture Process DAQ acquired data when called by ScansAvailable event.
%  dataCapture processes latest acquired data and timestamps from data
%  acquisition object (src), and, based on specified capture parameters (c
%  structure) and trigger configuration parameters from the user interface
%  elements (hGui handles structure), updates UI plots and captures data.
%
%   c.TimeSpan        = triggered capture timespan (seconds)
%   c.bufferTimeSpan  = required data buffer timespan (seconds)
%   c.bufferSize      = required data buffer size (number of scans)
%   c.plotTimeSpan    = continuous acquired data timespan (seconds)
%

[eventData, eventTimestamps] = read(src, src.ScansAvailableFcnCount, ...
    'OutputFormat', 'Matrix');


% The read data is stored in a persistent buffer (dataBuffer), which is
% sized to allow triggered data capture.
% Since multiple calls to dataCapture will be needed for a triggered
% capture, a trigger condition flag (trigActive) and a corresponding
% data timestamp (trigMoment) are used as persistent variables.
% Persistent variables retain their values between calls to the function.

persistent dataBuffer trigActive trigMoment

% If dataCapture is running for the first time, initialize persistent vars
if eventTimestamps(1)==0
    dataBuffer = [];          % data buffer
    trigActive = false;       % trigger condition flag
    trigMoment = [];          % data timestamp when trigger condition met
    prevData = [];            % last data point from previous callback execution
else
    prevData = dataBuffer(end, :);
end

% Store continuous acquisition timestamps and data in persistent FIFO
% buffer dataBuffer
latestData = [eventTimestamps, eventData];
dataBuffer = [dataBuffer; latestData];
numSamplesToDiscard = size(dataBuffer,1) - c.bufferSize;
if (numSamplesToDiscard > 0)
    dataBuffer(1:numSamplesToDiscard, :) = [];
end
```

```
    % Update live data plot
    % Plot latest plotTimeSpan seconds of data in dataBuffer
    samplesToPlot = min([round(c.plotTimeSpan * src.Rate), size(dataBuffer,1)]);
    firstPoint = size(dataBuffer, 1) - samplesToPlot + 1;
    % Update x-axis limits
    xlim(hGui.Axes1, [dataBuffer(firstPoint,1), dataBuffer(end,1)]);
    % Live plot has one line for each acquisition channel
    for ii = 1:numel(hGui.LivePlot)
        set(hGui.LivePlot(ii), 'XData', dataBuffer(firstPoint:end, 1), ...
                               'YData', dataBuffer(firstPoint:end, 1+ii))
    end


    % If capture is requested, analyze latest acquired data until a trigger
    % condition is met. After enough data is acquired for a complete capture,
    % as specified by the capture timespan, extract the capture data from the
    % data buffer and save it to a base workspace variable.

    % Get capture toggle button value (1 or 0) from UI
    captureRequested = hGui.CaptureButton.Value;

    if captureRequested && (~trigActive)
        % State: "Looking for trigger event"

        % Update UI status
        hGui.StatusText.String = 'Waiting for trigger';

        % Get the trigger configuration parameters from UI text inputs and
        %   place them in a structure.
        % For simplicity, validation of user input is not addressed in this example.
        trigConfig.Channel = sscanf(hGui.TrigChannel.String, '%u');
        trigConfig.Level = sscanf(hGui.TrigLevel.String, '%f');
        trigConfig.Slope = sscanf(hGui.TrigSlope.String, '%f');

        % Determine whether trigger condition is met in the latest acquired data
        % A custom trigger condition is defined in trigDetect user function
        [trigActive, trigMoment] = trigDetect(prevData, latestData, trigConfig);


    elseif captureRequested && trigActive && ((dataBuffer(end,1)-trigMoment) > c.TimeSpan)
        % State: "Acquired enough data for a complete capture"
        % If triggered and if there is enough data in dataBuffer for triggered
        % capture, then captureData can be obtained from dataBuffer.

        % Find index of sample in dataBuffer with timestamp value trigMoment
        trigSampleIndex = find(dataBuffer(:,1) == trigMoment, 1, 'first');
        % Find index of sample in dataBuffer to complete the capture
        lastSampleIndex = round(trigSampleIndex + c.TimeSpan * src.Rate());
        captureData = dataBuffer(trigSampleIndex:lastSampleIndex, :);

        % Reset trigger flag, to allow for a new triggered data capture
        trigActive = false;

        % Update captured data plot (one line for each acquisition channel)
        for ii = 1:numel(hGui.CapturePlot)
            set(hGui.CapturePlot(ii), 'XData', captureData(:, 1), ...
                                      'YData', captureData(:, 1+ii))
        end
```

```
    % Update UI to show that capture has been completed
    hGui.CaptureButton.Value = 0;
    hGui.StatusText.String = '';

    % Save captured data to a base workspace variable
    % For simplicity, validation of user input and checking whether a variable
    % with the same name already exists are not addressed in this example.
    % Get the variable name from UI text input
    varName = hGui.VarName.String;
    % Use assignin function to save the captured data in a base workspace variable
    assignin('base', varName, captureData);

elseif captureRequested && trigActive && ((dataBuffer(end,1)-trigMoment) < c.TimeSpan)
    % State: "Capturing data"
    % Not enough acquired data to cover capture timespan during this callback execution
    hGui.StatusText.String = 'Triggered';

elseif ~captureRequested
    % State: "Capture not requested"
    % Capture toggle button is not pressed, set trigger flag and update UI
    trigActive = false;
    hGui.StatusText.String = '';
end

drawnow

end
```

Create a Graphical User Interface for Live Data Capture

Create a user interface programmatically, by creating a figure, one plot for live acquired data, one plot for captured data, buttons for starting capture and stopping acquisition, and text fields for entering trigger configuration parameters and status update.

For simplicity, the figure and all user interface components have a fixed size and position defined in pixels. For high DPI displays the position values might have to be adjusted for optimum dimensions and layout. Another option for creating a custom UI is to use App Designer.

```
function hGui = createDataCaptureUI(s)
%createDataCaptureUI Create a graphical user interface for data capture.
%   hGui = createDataCaptureUI(s) returns a structure of graphics
%   components handles (hGui) and creates a graphical user interface, by
%   programmatically creating a figure and adding required graphics
%   components for visualization of data acquired from a data acquisition
%   object (s).

% Create a figure and configure a callback function (executes on window close)
hGui.Fig = figure('Name','Software-analog triggered data capture', ...
    'NumberTitle', 'off', 'Resize', 'off', ...
    'Toolbar', 'None', 'Menu', 'None',...
    'Position', [100 100 750 650]);
hGui.Fig.DeleteFcn = {@endDAQ, s};
uiBackgroundColor = hGui.Fig.Color;

% Create the continuous data plot axes with legend
% (one line per acquisition channel)
hGui.Axes1 = axes;
```

```
hGui.LivePlot = plot(0, zeros(1, numel(s.Channels)));
xlabel('Time (s)');
ylabel('Voltage (V)');
title('Continuous data');
legend({s.Channels.ID}, 'Location', 'northwestoutside')
hGui.Axes1.Units = 'Pixels';
hGui.Axes1.Position = [207 391 488 196];
% Turn off axes toolbar and data tips for live plot axes
hGui.Axes1.Toolbar.Visible = 'off';
disableDefaultInteractivity(hGui.Axes1);

% Create the captured data plot axes (one line per acquisition channel)
hGui.Axes2 = axes('Units', 'Pixels', 'Position', [207 99 488 196]);
hGui.CapturePlot = plot(NaN, NaN(1, numel(s.Channels)));
xlabel('Time (s)');
ylabel('Voltage (V)');
title('Captured data');
hGui.Axes2.Toolbar.Visible = 'off';
disableDefaultInteractivity(hGui.Axes2);

% Create a stop acquisition button and configure a callback function
hGui.DAQButton = uicontrol('style', 'pushbutton', 'string', 'Stop DAQ',...
    'units', 'pixels', 'position', [65 394 81 38]);
hGui.DAQButton.Callback = {@endDAQ, s};

% Create a data capture button and configure a callback function
hGui.CaptureButton = uicontrol('style', 'togglebutton', 'string', 'Capture',...
    'units', 'pixels', 'position', [65 99 81 38]);
hGui.CaptureButton.Callback = {@startCapture, hGui};

% Create a status text field
hGui.StatusText = uicontrol('style', 'text', 'string', '',...
    'units', 'pixels', 'position', [67 28 225 24],...
    'HorizontalAlignment', 'left', 'BackgroundColor', uiBackgroundColor);

% Create an editable text field for the captured data variable name
hGui.VarName = uicontrol('style', 'edit', 'string', 'mydata',...
    'units', 'pixels', 'position', [87 159 57 26]);
% Create an editable text field for the trigger channel
hGui.TrigChannel = uicontrol('style', 'edit', 'string', '1',...
    'units', 'pixels', 'position', [89 258 56 24]);
% Create an editable text field for the trigger signal level
hGui.TrigLevel = uicontrol('style', 'edit', 'string', '1.0',...
    'units', 'pixels', 'position', [89 231 56 24]);
% Create an editable text field for the trigger signal slope
hGui.TrigSlope = uicontrol('style', 'edit', 'string', '200.0',...
    'units', 'pixels', 'position', [89 204 56 24]);
% Create text labels
hGui.txtTrigParam = uicontrol('Style', 'text', 'String', 'Trigger parameters', ...
    'Position', [39 290 114 18], 'BackgroundColor', uiBackgroundColor);
hGui.txtTrigChannel = uicontrol('Style', 'text', 'String', 'Channel', ...
    'Position', [37 261 43 15], 'HorizontalAlignment', 'right', ...
    'BackgroundColor', uiBackgroundColor);
hGui.txtTrigLevel = uicontrol('Style', 'text', 'String', 'Level (V)', ...
    'Position', [35 231 48 19], 'HorizontalAlignment', 'right', ...
    'BackgroundColor', uiBackgroundColor);
hGui.txtTrigSlope = uicontrol('Style', 'text', 'String', 'Slope (V/s)', ...
    'Position', [17 206 66 17], 'HorizontalAlignment', 'right', ...
```

```
    'BackgroundColor', uiBackgroundColor);
hGui.txtVarName = uicontrol('Style', 'text', 'String', 'Variable name', ...
    'Position', [35 152 44 34], 'BackgroundColor', uiBackgroundColor);

end

function startCapture(obj, ~, hGui)
if obj.Value
    % If button is pressed clear data capture plot
    for ii = 1:numel(hGui.CapturePlot)
        set(hGui.CapturePlot(ii), 'XData', NaN, 'YData', NaN);
    end
end
end

function endDAQ(~, ~, s)
if isvalid(s)
    if s.Running
        stop(s);
    end
end
end
```

```
Background operation has started.
To stop the background operation, use stop.
To read acquired scans, use read.
```

Detect Trigger Condition in Acquired Data

In this example, the trigger condition is defined by the signal level on the trigger channel and the corresponding slope. Depending on the application and actual data being acquired, data filtering or more complex trigger conditions can be implemented.

```
function [trigDetected, trigMoment] = trigDetect(prevData, latestData, trigConfig)
%trigDetect Detect if trigger condition is met in acquired data
%   [trigDetected, trigMoment] = trigDetect(prevData, latestData, trigConfig)
%   Returns a detection flag (trigDetected) and the corresponding timestamp
%   (trigMoment) of the first data point which meets the trigger condition
%   based on signal level and slope specified by the trigger parameters
%   structure (trigConfig).
%   The input data (latestData) is an N x M matrix corresponding to N acquired
%   data scans, with the timestamps as the first column, and channel data
%   as columns 2:M. The previous data point prevData (1 x M vector of timestamp
%   and channel data) is used to determine the slope of the first data point.
%
%   trigConfig.Channel = index of trigger channel in data acquisition object channels
%   trigConfig.Level   = signal trigger level (V)
%   trigConfig.Slope   = signal trigger slope (V/s)

% Condition for signal trigger level
trigCondition1 = latestData(:, 1+trigConfig.Channel) > trigConfig.Level;

data = [prevData; latestData];

% Calculate slope of signal data points
% Calculate time step from timestamps
dt = latestData(2,1)-latestData(1,1);
slope = diff(data(:, 1+trigConfig.Channel))/dt;
```

```
% Condition for signal trigger slope
trigCondition2 = slope > trigConfig.Slope;

% If first data block acquired, slope for first data point is not defined
if isempty(prevData)
    trigCondition2 = [false; trigCondition2];
end

% Combined trigger condition to be used
trigCondition = trigCondition1 & trigCondition2;

trigDetected = any(trigCondition);
trigMoment = [];
if trigDetected
    % Find time moment when trigger condition has been met
    trigTimeStamps = latestData(trigCondition, 1);
    trigMoment = trigTimeStamps(1);
end
end
```

Count Pulses on a Digital Signal Using NI Devices

This example shows how to determine the rate of rotation of an Anaheim Automation motor controller by counting the number of rising edges in the signal. The controller returns hall effect pulses (square waves) that serve as frequency feedback for motor rotation speeds.

Create a Counter Input Channel

Use `daq` to create a DataAcquisition and `addinput` to add a counter input channel with `EdgeCount` measurement type. For this example, use CompactDAQ chassis NI c9178 and module NI 9402 with ID cDAQ1Mod5.

```
dq = daq("ni");
ch = addinput(dq,"cDAQ1Mod5", "ctr0", "EdgeCount");
ch
```

```
ch = 

    Index    Type      Device       Channel    Measurement Type    Range          Name
    _____    ____    ___________    _______    ________________    _____    ________________

      1      "ci"    "cDAQ1Mod5"    "ctr0"       "EdgeCount"       "n/a"    "cDAQ1Mod5_ctr0"
```

Determine the Terminal of the Counter Input Channel

To connect the input signal to the correct terminal, examine the `Terminal` property of the channel. The terminal is determined by the hardware.

```
ch.Terminal
```

```
ans = 

    'PFI0'
```

Read the Counter Channel

To determine if the counter is operational, input a single scan, pause while the motor rotates, then read the counter again.

```
read(dq)
```

```
ans =

  timetable

    Time     cDAQ1Mod5_ctr0
    _____    ______________

    0 sec          3
```

```
pause(0.1);
read(dq)
```

```
ans =

  timetable

    Time     cDAQ1Mod5_ctr0
    _____    ______________

    0 sec          14
```

```
pause(0.1);
read(dq)
```

```
ans =

  timetable

    Time     cDAQ1Mod5_ctr0
    _____    ______________

    0 sec          27
```

Measure Revolutions per Second

Count the number of pulses by resetting the counter to zero, pause for one second, and read the counter. The hall effects are oriented every 120 degrees and generate three square wave pulses for every rotation.

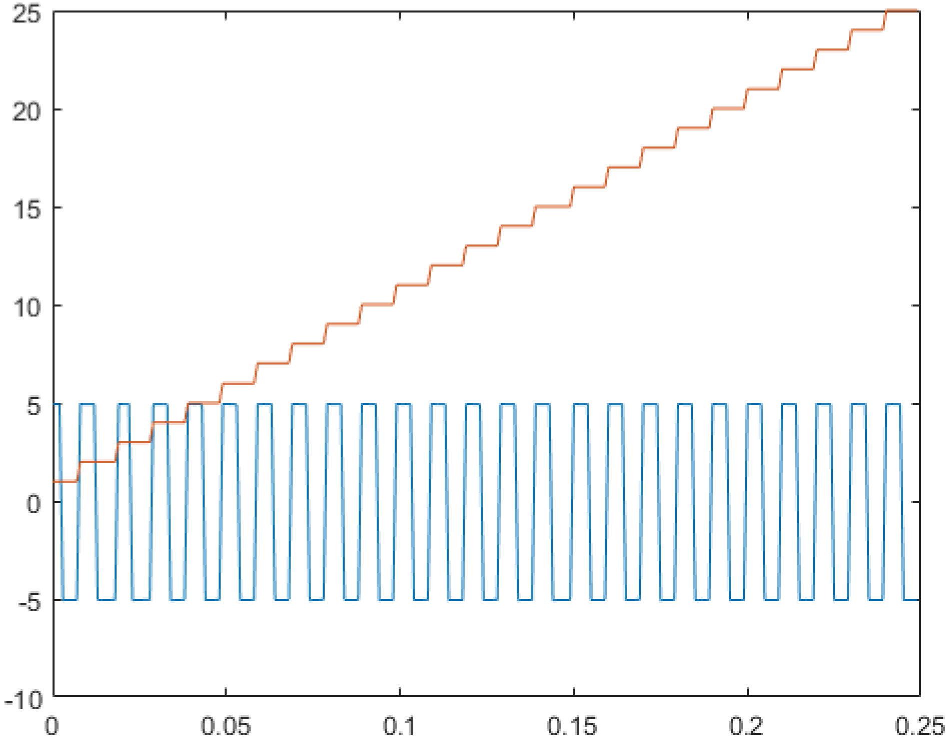
```
resetcounters(dq);
pause(1);
read(dq, "OutputFormat", "Matrix")/3
```

```
ans =

   33.6667
```

Use Hardware Clock for Higher Accuracy

The hardware clock is highly accurate. Use the hardware clock to acquire multiple counter measurements. NI counter devices require an external clock. By adding an analog input channel for a module on the same chassis, the DataAcquisition shares an internal clock with both modules.

```
dq = daq("ni");
addinput(dq,"cDAQ1Mod1", "ai0", "Voltage");
addinput(dq,"cDAQ1Mod5", "ctr0", "EdgeCount");
data = read(dq, seconds(0.25));
plot(data.Time, data.Variables);
```


25
20
15
10
5
0
-5
-10
0
0.05
0.1
0.15
0.2
0.25

Measure Frequency Using NI Devices

This example shows how to measure frequency to determine rate of flow of fluid using a flow sensor. The sensor generates a digital signal with frequency that correlates to the rate of flow of fluid.

Create a Counter Input Channel

Use `daq` to create a DataAcquisition and `addinput` to add a counter input channel with `Frequency` measurement type. For this example, use CompactDAQ chassis NI c9178 and module NI 9402 with ID cDAQ1Mod5.

```
dq = daq("ni");
ch = addinput(dq,"cDAQ1Mod5", "ctr0", "Frequency");
ch
```

```
ch =

    Index    Type      Device       Channel    Measurement Type    Range          Name
    _____    ____    ___________    _______    ________________    _____    ________________

      1      "ci"    "cDAQ1Mod5"    "ctr0"       "Frequency"       "n/a"    "cDAQ1Mod5_ctr0"
```

Determine the Terminal of the Counter Input Channel

To connect the input signal to the correct terminal, examine the `Terminal` property of the channel. The terminal is determined by the hardware.

```
ch.Terminal
```

```
ans =

    'PFI1'
```

Measure Frequency

To determine if the counter is operational, input a single scan while the motor is rotating.

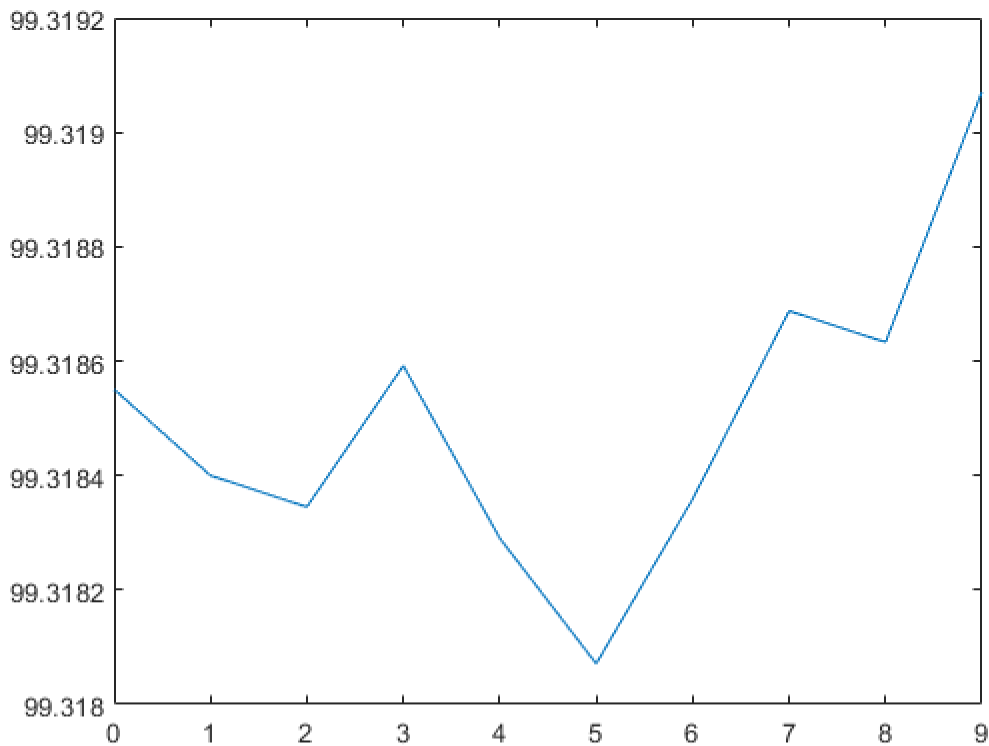
```
read(dq)
```

```
ans =

  timetable

    Time     cDAQ1Mod5_ctr0
    _____    ______________

    0 sec         100
```

Monitor Frequency over Time

Use the hardware clock to acquire multiple counter measurements over time. NI counter devices require an external clock. By adding an analog input channel for a module on the same chassis, the session shares an internal clock with both modules.

```
dq = daq("ni");
dq.Rate = 1;
addinput(dq,"cDAQ1Mod1", "ai0", "Voltage");
addinput(dq,"cDAQ1Mod5", "ctr0", "Frequency");

data = read(dq, seconds(10));
plot(data.Time, data.cDAQ1Mod5_ctr0);
```



Measure Pulse Width Using NI Devices

This example shows how to measure the width of an active high pulse. A sensor is used to measure distance from a point: the width of the pulse is correlated with the measured distance.

Create a Counter Input Channel

Create a DataAcquisition, and add a counter input channel with `PulseWidth` measurement type. For this example, use CompactDAQ chassis NI c9178 and module NI 9402 with ID cDAQ1Mod5.

```
dq = daq("ni");
ch = addinput(dq, "cDAQ1Mod5", "ctr0", "PulseWidth");
```

Determine the Terminal of the Counter Input Channel

To connect the input signal to the correct terminal, examine the `Terminal` property of the channel. The terminal is determined by the hardware.

```
ch.Terminal
```

```
ans =

    'PFI1'
```

Measure Distance

To determine if the counter is operational, acquire a single scan. The sensor generates a high pulse of width 0.0010 seconds corresponding a distance of one meter.

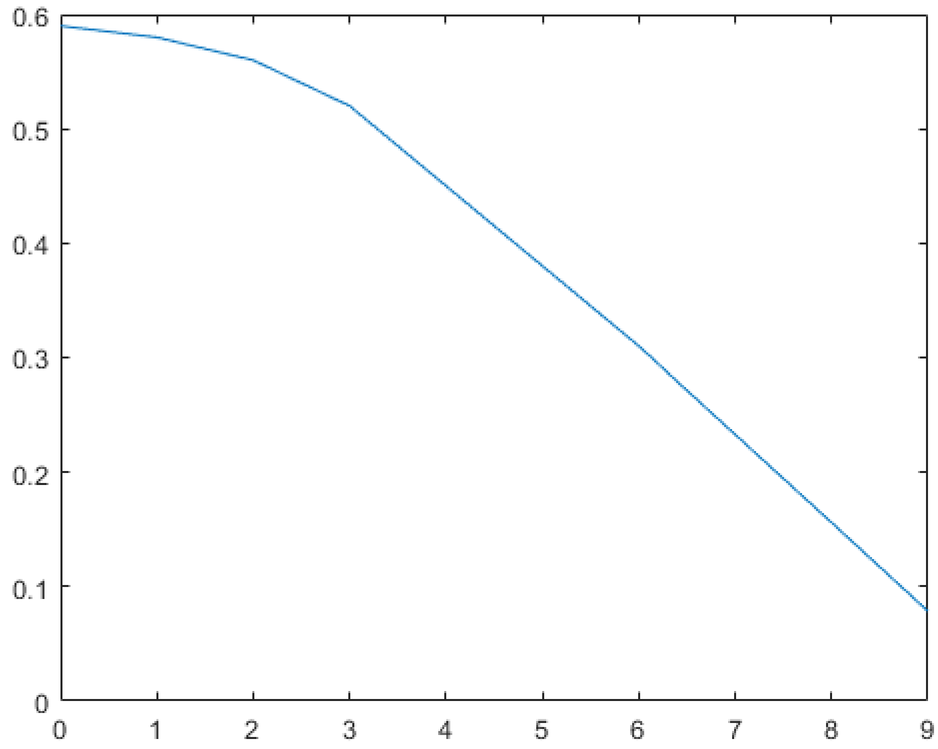
```
1000*read(dq, "OutputFormat", "Matrix")
```

```
ans =

     5
```

Measure Distance over Time

Use the hardware clock to acquire multiple counter measurements over time. NI counter devices require an external clock. By adding an analog input channel for a module on the same chassis, the internal clock is shared with both modules.

```
dq = daq("ni");
addinput(dq, "cDAQ1Mod1", "ai0", "Voltage");
addinput(dq, "cDAQ1Mod5", "ctr0", "PulseWidth");

dq.Rate = 1;
data = read(dq, seconds(10));
plot(data.Time, 1000*data.cDAQ1Mod5_ctr0);
```


0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
6
7
8
9

Generate Pulse Width Modulated Signals Using NI Devices

This example shows how to generate a pulse width modulated signal to drive a stepper motor.

Create a Counter Output Channel

Use `daq` to create a DataAcquisition. Use `addoutput` to add a counter output channel with `PulseGeneration` measurement type, and `addinput` to add an analog input channel to monitor the pulse generated by the counter output channel. For this example, use CompactDAQ chassis NI c9178 and module NI 9402 with ID cDAQ1Mod5 for the pulse generation and NI 9205 with ID cDAQ1Mod1 for the voltage input.

```
dq = daq("ni");
addinput(dq,"cDAQ1Mod1", "ai0", "Voltage");
ctr = addoutput(dq,"cDAQ1Mod5", "ctr0", "PulseGeneration");
dq.Channels
```

```
ans =

    Index    Type       Device       Channel    Measurement Type            Range            N
    _____    ____    ___________    _______    _________________    ___________________    ______

      1      "ai"    "cDAQ1Mod1"    "ai0"      "Voltage (Diff)"     "-10 to +10 Volts"     "cDAQ1M
      2      "co"    "cDAQ1Mod5"    "ctr0"     "PulseGeneration"    "n/a"                  "cDAQ1M
```

Determine the Terminal of the Counter Output Channel

To connect the output signal to the correct terminal, examine the `Terminal` property of the counter channel. The terminal is determined by the hardware.

```
ctr.Terminal
```

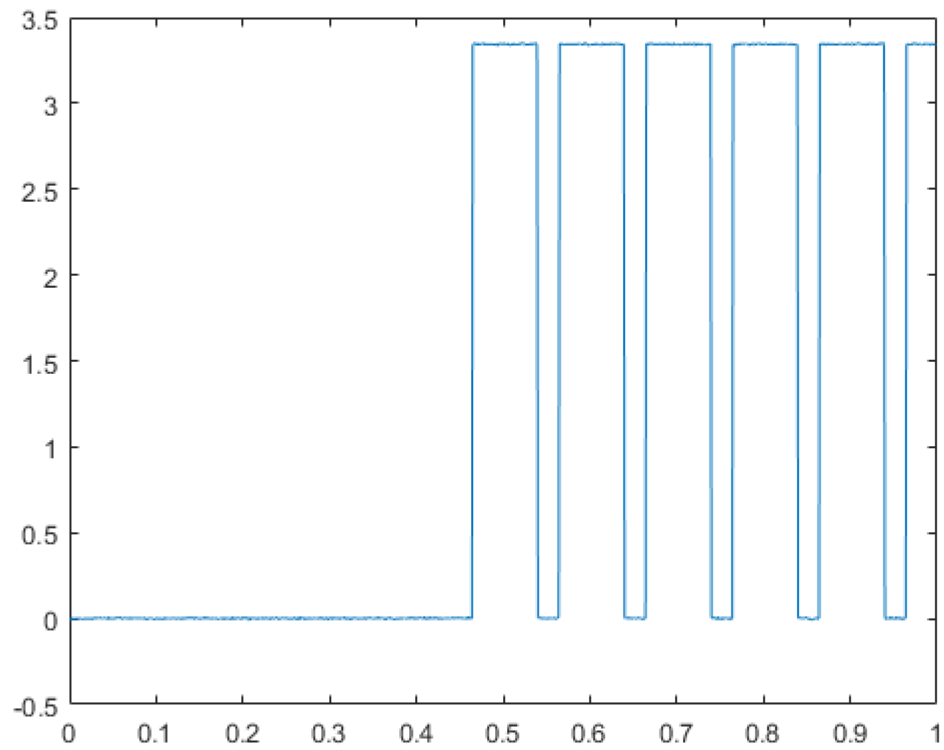
```
ans =

    'PFI0'
```

Clocked Counter Output

Use counter output channel 0 to generate a fixed pulse width modulated signal on terminal PFI0. Trigger the motor after 0.5 seconds, with a 75% duty cycle.

```
ctr.Frequency = 10;
ctr.InitialDelay = 0.5;
ctr.DutyCycle = 0.75;

% StartForeground returns data for input channels only. The data variable
% will contain one column of data.
start(dq, "Duration", seconds(1));

while dq.Running
    pause(0.1);
end
```

```
data = read(dq, seconds(1));
plot(data.Time, data.Variables);
```



Measure Angular Position with an Incremental Rotary Encoder

This example shows how to acquire angular position data using an incremental rotary encoder and a multifunction data acquisition (DAQ) device with the Data Acquisition Toolbox quadrature encoder measurement functionality.

An incremental rotary encoder is typically mounted on the shaft of a mechanical system, such as a wind turbine or a robotic arm, to provide motion or position information. The encoder outputs two quadrature signals, which provide information on the relative change in position and the direction of rotation. The counter subsystem of the DAQ device uses the signals output by the encoder to calculate the change in position, and keep track of the most recent position value. In MATLAB, an input channel with a `Position` measurement type is used to read the position values.

This example uses an optical shaft encoder (US Digital H6-2500-IE-S) and a multifunction DAQ device (NI USB-6255) with counter channels which have quadrature encoder capability.

Create a Data Acquisition Object

Create a data acquisition object and add an input channel with `Position` measurement type.

```
s = daq('ni');
ch1 = addinput(s, 'Dev1', 'ctr0', 'Position');
```

Configure Hardware

A rotary quadrature encoder outputs two quadrature signals, A and B, which provide information on the relative change in position and the direction of rotation. Optionally, some models also output an index or reference signal, Z, which is active once per revolution. You can use the Z signal to reset the counter position to a known reference value.

Connect A, B, and Z signal outputs from the encoder device to the proper DAQ input terminals specified by the DAQ device datasheet (PFI8, PFI10, and PFI9 for NI USB-6255). The correct terminals depend on the device model and the counter channel used, and can be listed by reading the following properties:

```
ch1.TerminalA
```

```
ans =

    'PFI8'
```

```
ch1.TerminalB
```

```
ans =

    'PFI10'
```

```
ch1.TerminalZ
```

```
ans =
```

```
    'PFI9'
```

Configure quadrature cycle encoding type (X1, X2, or X4). This corresponds to the number of counts (counter value increments or decrements) output by the encoder for each quadrature cycle (1, 2, or 4.), as specified in the encoder datasheet.

```
ch1.EncoderType = 'X1';
```

Read Encoder Position on Demand

The DAQ device counter hardware keeps track of the relative position changes signaled by the encoder. Use `read` to read an updated position from the counter input channel.

```
encoderPosition = read(s, 1, 'OutputFormat', 'Matrix')
```

```
encoderPosition =

     0
```

This example uses an optical encoder model with a resolution of 2500 quadrature cycles per shaft revolution, as specified in the encoder datasheet.

Convert counter values to angular position (in degrees) using the encoder resolution and the encoding type ('X1' in this case).

```
encoderCPR = 2500;
encoderPositionDeg = encoderPosition * 360/encoderCPR
```

```
encoderPositionDeg =

     0
```

Acquire Hardware-Timed Encoder Position Data

For applications where high time-resolution is required the data acquisition must be hardware-timed (clocked). As proof of concept, this example characterizes the motion of a swinging pendulum by measuring its angular position vs. time.

To acquire hardware-timed data from a counter input channel, NI DAQ devices require the use of an external clock or the use of a clock from another subsystem.

Add an analog input channel to the data acquisition object to automatically share this system's scan clock.

```
addinput(s, 'Dev1', 'ai0', 'Voltage');
```

Configure acquisition rate (samples/s) and acquisition duration in seconds.

```
s.Rate = 10000;
daqDuration = seconds(35);
```

Acquire data in the foreground.

```
[positionData, timestamps] = read(s, daqDuration, 'OutputFormat', 'Matrix');
```

By default, counter position readings are unsigned integer values. The counter channels of the DAQ device used in this example are 32-bit, so any counter value read will be in the range 0 to 2^32-1. Depending on the application, you may want to obtain signed position values (positive or negative) as decrementing the counter value past zero is a discontinuous wraparound to 2^32-1.

For 32-bit counter channels, use 2^31 as the threshold counter value for conversion to signed position values. The result is valid if the actual position value is in the range -2^31+1 to 2^31.

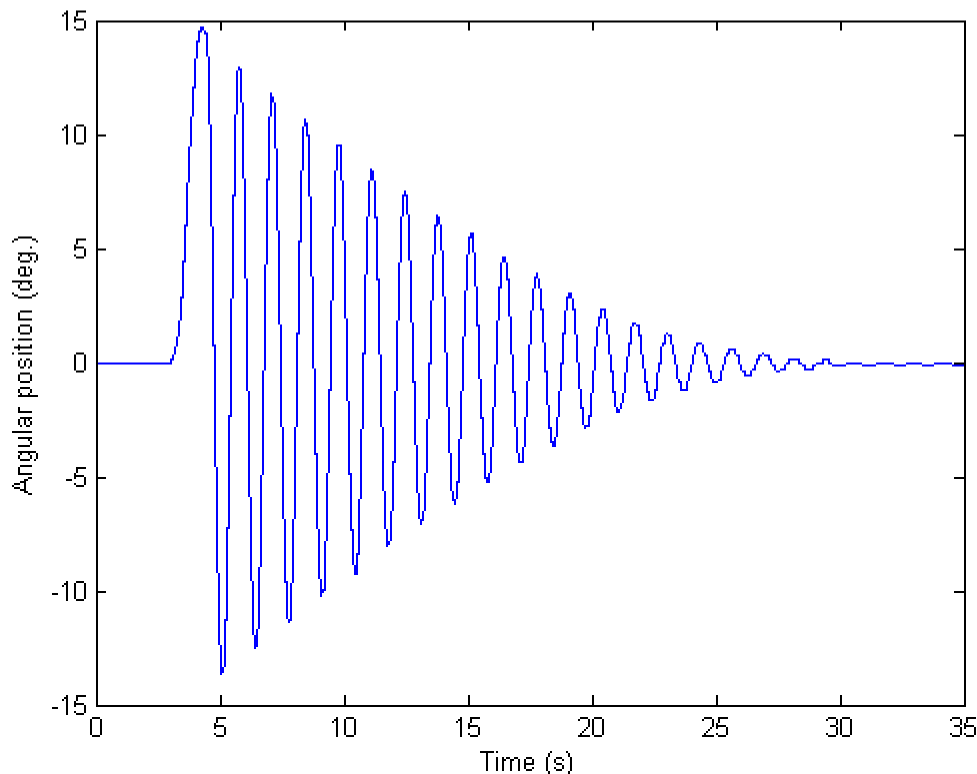
```
counterNBits = 32;
signedThreshold = 2^(counterNBits-1);
signedData = positionData(:,1);
signedData(signedData > signedThreshold) = signedData(signedData > signedThreshold) - 2^counterNB
```

Calculate encoder position data in degrees.

```
positionDataDeg = signedData * 360/encoderCPR;
```

Plot the signed angular position data acquired for the oscillatory motion of a pendulum.

```
figure
plot(timestamps, positionDataDeg);
xlabel('Time (s)');
ylabel('Angular position (deg.)');
```



Use the Z Signal to Reference the Relative Position to a Known Absolute Position

The A and B quadrature signals output by incremental rotary encoders provide only relative position information (direction of motion and changes in position). The optional reference signal Z is a single

pulse output once per encoder shaft revolution at a predefined absolute location. Referencing the relative position value to the known absolute position reference allows an incremental rotary encoder to function as a pseudo-absolute position encoder. This is useful in accurate positioning applications (such as industrial automation, robotics, solar tracking, radar antenna or telescope positioning).

For incremental rotary encoders that provide a Z index signal output, the counter position value can be configured to be reset automatically to the known reference value.

Set the `ZResetEnable` and `ZResetCondition` properties.

```
ch1.ZResetEnable = true;
```

Configure the `ZResetCondition`, which is based on the A and B phase signals.

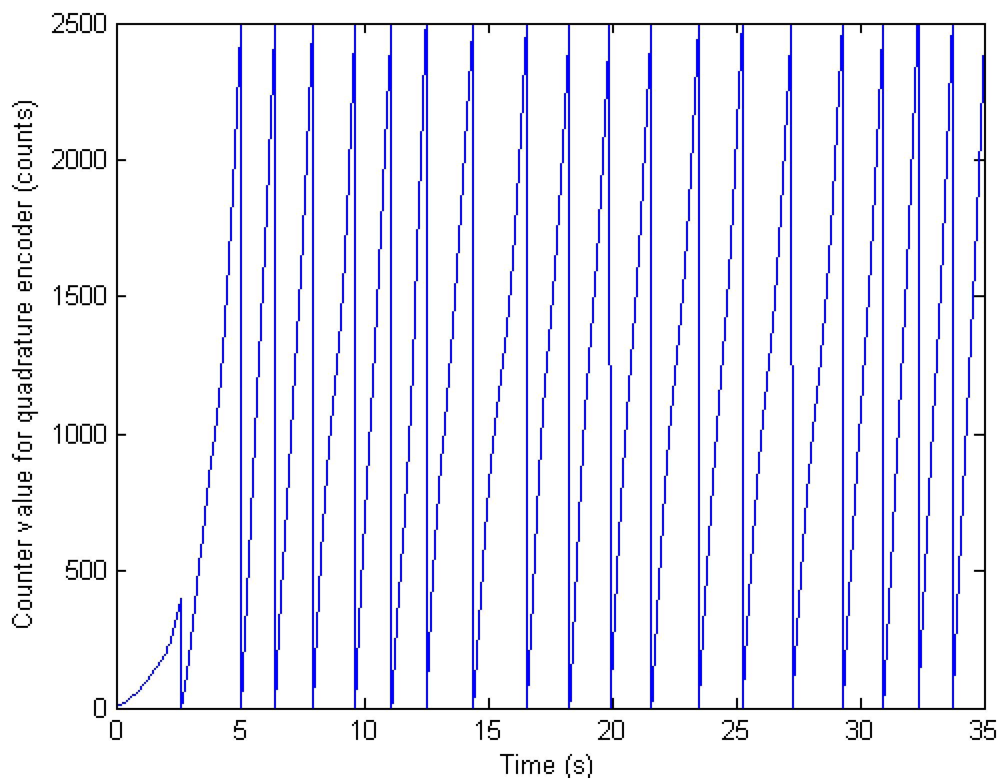
```
ch1.ZResetCondition = 'BothLow';
```

Specify the absolute reference position value `ZResetValue` to which the counter value will be reset.

```
ch1.ZResetValue = 0;
```

Acquire and plot a set of hardware-timed counter position data to show how you can use the encoder Z index signal to automatically reset the counter value to a known reference value.

```
[positionData2, timestamps2] = read(s, daqDuration, 'OutputFormat', 'matrix');

figure
plot(timestamps2, positionData2(:,1));
xlabel('Time (s)');
ylabel('Counter value for quadrature encoder (counts)');
```



The acquired position data corresponds to a rotary encoder shaft that is rotating continuously. Notice that before the first time the counter value is reset the position value is not referenced to an absolute position, whereas the other counter reset events occur when the counter value is 2500 (the encoder CPR value).

Control Stepper Motor Using Digital Outputs

This example shows how to control a stepper motor using digital output ports.

Discover Devices Supporting Digital Output

Use `daqlist` to discover devices. This example uses a National Instruments® ELVIS II with ID `Dev2`.

```
d = daqlist
```

```
d =

  12×5 table

    VendorID     DeviceID               Description                     Model          Device
    ________    ___________    __________________________________    ____________    ___________

      "ni"      "cDAQ1Mod1"    "National Instruments NI 9205"        "NI 9205"       [1×1 daq.De
      "ni"      "cDAQ1Mod2"    "National Instruments NI 9263"        "NI 9263"       [1×1 daq.De
      "ni"      "cDAQ1Mod3"    "National Instruments NI 9234"        "NI 9234"       [1×1 daq.De
      "ni"      "cDAQ1Mod4"    "National Instruments NI 9201"        "NI 9201"       [1×1 daq.De
      "ni"      "cDAQ1Mod5"    "National Instruments NI 9402"        "NI 9402"       [1×1 daq.De
      "ni"      "cDAQ1Mod6"    "National Instruments NI 9213"        "NI 9213"       [1×1 daq.De
      "ni"      "cDAQ1Mod7"    "National Instruments NI 9219"        "NI 9219"       [1×1 daq.De
      "ni"      "cDAQ1Mod8"    "National Instruments NI 9265"        "NI 9265"       [1×1 daq.De
      "ni"      "Dev1"         "National Instruments PCIe-6363"      "PCIe-6363"     [1×1 daq.De
      "ni"      "Dev2"         "National Instruments NI ELVIS II"    "NI ELVIS II"   [1×1 daq.De
      "ni"      "Dev3"         "National Instruments PCIe-6363"      "PCIe-6363"     [1×1 daq.De
      "ni"      "Dev4"         "National Instruments PCIe-6363"      "PCIe-6363"     [1×1 daq.De
```

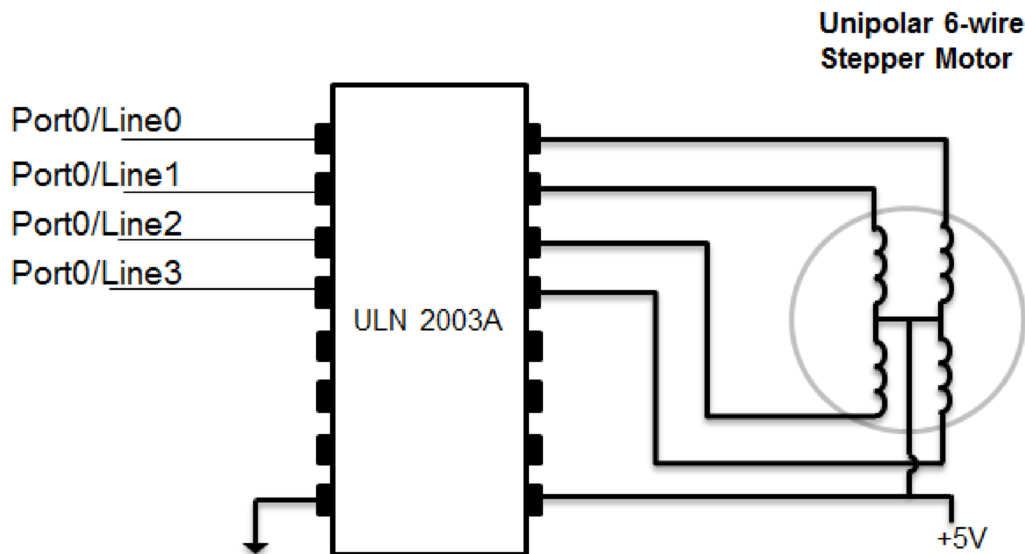
```
d{10, "DeviceInfo"}
```

```
ans =

ni: National Instruments NI ELVIS II (Device ID: 'Dev2')
   Analog input supports:
      7 ranges supported
      Rates from 0.0 to 1250000.0 scans/sec
      16 channels ('ai0' - 'ai15')
      'Voltage' measurement type

   Analog output supports:
      -5.0 to +5.0 Volts,-10 to +10 Volts ranges
      Rates from 0.0 to 2857142.9 scans/sec
      2 channels ('ao0','ao1')
      'Voltage' measurement type

   Digital IO supports:
      39 channels ('port0/line0' - 'port2/line6')
      'InputOnly','OutputOnly','Bidirectional' measurement types

   Counter input supports:
      Rates from 0.1 to 80000000.0 scans/sec
      2 channels ('ctr0','ctr1')
```

```
      'EdgeCount' measurement type

   Counter output supports:
      Rates from 0.1 to 80000000.0 scans/sec
      2 channels ('ctr0','ctr1')
      'PulseGeneration' measurement type
```

Hardware Setup Description

This example uses a Portescap 20M020D1U motor (5 V, 18 degree unipolar stepper). The TTL signals produced by the digital I/O system are amplified by a Texas Instruments ULN2003AIN (high voltage, high current Darlington transistor array), as shown in this schematic:



Add Digital Output-Only Channels

Create a DataAcquisition and add 4 digital channels on port 0, lines 0-3. Set the measurement type to `OutputOnly`. These are connected to the 4 control lines for the stepper motor.

```
dq = daq("ni");
addoutput(dq,"Dev2","port0/line0:3","Digital")
```

```
Warning: Added channel does not support clocked sampling: clocked operations are
disabled. Only on-demand operations are allowed.
```

Define Motor Steps

Refer to the Portescap motor wiring diagram describing the sequence of 4-bit patterns. Send this pattern sequentially to the motor to produce counterclockwise motion. Each step turns the motor 18

degrees. Each cycle of 4 steps turns the motor 72 degrees. Repeat this cycle five times to rotate the motor 360 degrees.

```
step1 = [1 0 1 0];
step2 = [1 0 0 1];
step3 = [0 1 0 1];
step4 = [0 1 1 0];
```

Rotate Motor

Use `write` to output the sequence to turn the motor 72 degrees counterclockwise.

```
write(dq,step1);
write(dq,step2);
write(dq,step3);
write(dq,step4);
```

Repeat sequence 50 times to rotate the motor 10 times counterclockwise.

```
for motorstep = 1:50
    write(dq,step1);
    write(dq,step2);
    write(dq,step3);
    write(dq,step4);
end
```

To turn the motor 72 degrees clockwise, reverse the order of the steps.

```
write(dq,step4);
write(dq,step3);
write(dq,step2);
write(dq,step1);
```

Turn Off All Outputs

After you use the motor, turn off all the lines to allow the motor to rotate freely.

```
write(dq,[0 0 0 0]);
```

Communicate with I2C Devices and Analyze Bus Signals Using Digital IO

Communicate with instruments and devices at the protocol layer as well as the physical layer. Use the I2C feature of Instrument Control Toolbox to communicate with a TMP102 temperature sensor, and simultaneously analyze the physical layer I2C bus communications using the clocked digital IO feature of Data Acquisition Toolbox.

Data Acquisition Toolbox and Instrument Control Toolbox are required.

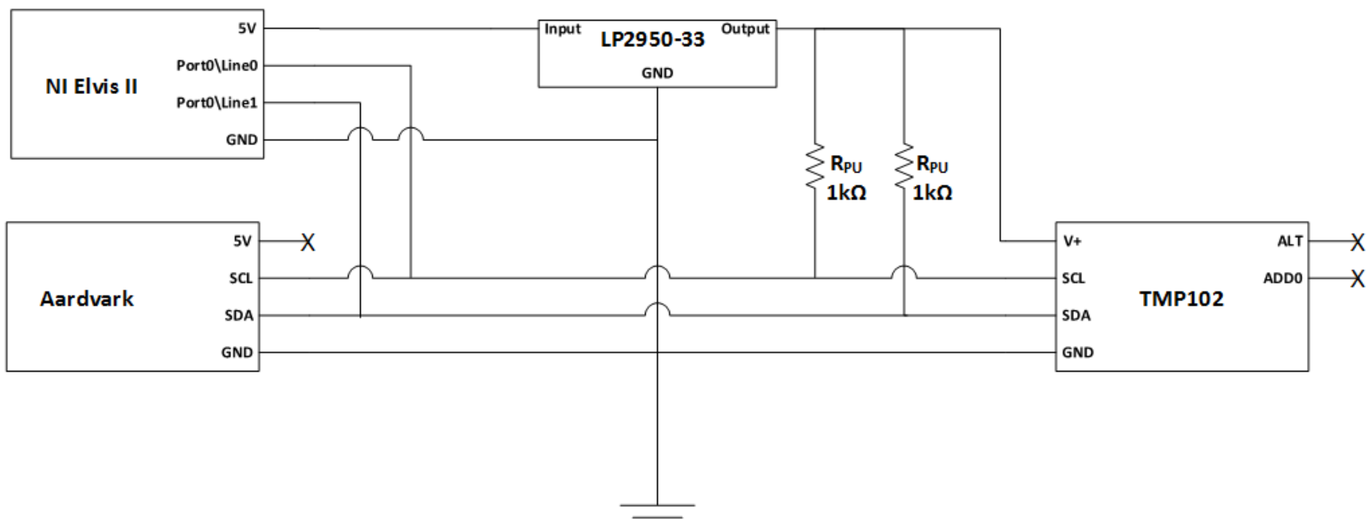
Hardware Configuration and Schematic

- Any supported National Instruments™ DAQ device with clocked DIO channels can be used (e.g., NI Elvis II)
- TotalPhase Aardvark I2C/SPI Host Adaptor
- TMP102 Digital Temperature Sensor with two-wire serial interface

The TMP102 requires a 3.3 V supply. Use a linear LDO (LP2950-33) to generate the 3.3 V supply from the DAQ device's 5 V supply line.

Alternative options include:

- Use an external power supply.
- Use an analog output channel from your DAQ device.



Connect to a TMP102 Sensor Using I2C Host Adaptor and Read Temperature Data

Wire up the sensor and verify communication to it using the I2C object from Instrument Control Toolbox.

```
aa = instrhwinfo('i2c', 'aardvark');      % Get information about connected I2C hosts
tmp102 = i2c('aardvark',0,hex2dec('48')); % Create an I2C object to connect to the TMP102
tmp102.PullupResistors = 'both';          % Use host adaptor pull-up resistors
fopen(tmp102);                            % Open the connection
data8 = fread(tmp102, 2, 'uint8');        % Read 2 byte data
% One LSB equals 0.0625 deg. C
temperature = ...
    (double(bitshift(int16(data8(1)), 4)) +...
     double(bitshift(int16(data8(2)), -4))) * 0.0625; % Refer to TMP102 data sheet to calculate
fprintf('The temperature recorded by the TMP102 sensor is: %s deg. C\n',num2str(temperature));
fclose(tmp102);
```

```
The temperature recorded by the TMP102 sensor is: 27.625 deg. C
```

Acquire the Corresponding I2C Physical Layer Signals Using a DAQ Device

Use oversampled clocked digital channels from the NI Elvis (`Dev4`) to acquire and analyze the physical layer communications on the I2C bus.

Acquire SDA data on port 0, line 0 of your DAQ device. Acquire SCL data on port 0, line 1 of your DAQ device.

```
dd = daq("ni");
addinput(dd,"Dev4","port0\line0","Digital"); % sda
addinput(dd,"Dev4","port0\line1","Digital"); % scl
```

Generate a Clock Signal for Use with the Digital Subsystem

Digital subsystems on NI DAQ devices do not have their own clock; they must share a clock with the analog subsystem or import a clock from an external subsystem. Generate a 50% duty cycle clock at 1 MHz using a `PulseGeneration` counter output, and set the input scan rate to match.

```
pgChan = addoutput(dd,"Dev4","ctr1","PulseGeneration");
dd.Rate = 1e6;
pgChan.Frequency = dd.Rate;
```

The clock is generated on the 'pgChan.Terminal' pin, allowing synchronization with other devices and viewing the clock on an oscilloscope. The counter output pulse signal is imported as a clock signal.

```
disp(pgChan.Terminal);
addclock(dd,"ScanClock","External",["Dev4/" pgChan.Terminal]);
```

```
PFI13
```

Acquire the I2C Signals Using Clocked Digital Channels

Acquire data in the background from the SDA and SCL digital lines.

- Start the DataAcquisition in background mode
- Start the I2C operations
- Stop the DataAcquisition after I2C operations are complete

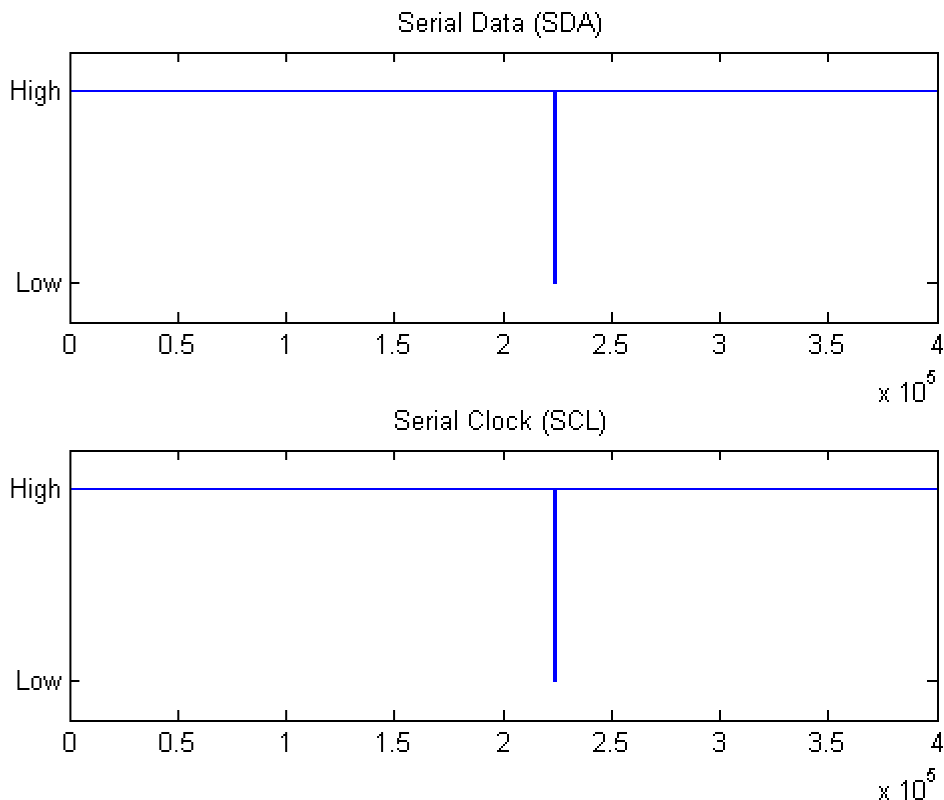
```
start(dd, "continuous");
fopen(tmp102);
data8 = fread(tmp102, 2, "uint8");
% One LSB equals 0.0625 deg. C
temperature = (double(bitshift(int16(data8(1)), 4)) +...
```

```
    double(bitshift(int16(data8(2)), -4))) * 0.0625;
fclose(tmp102);
pause(0.1);
stop(dd);
myData = read(dd, "all");
```

```
Warning: Triggers and Clocks will not affect counter output channels.
```

Plot the raw data to see the acquired signals. Notice that lines are held high during idle periods. The next section shows how to find the start/stop condition bits and use them to isolate areas of interest in the I2C communication.

```
figure("Name", "Raw Data");
subplot(2,1,1);

plot(myData(:,1));
ylim([-0.2, 1.2]);
ax = gca;
ax.YTick = [0,1];
ax.YTickLabel = {'Low','High'};
title("Serial Data (SDA)");
subplot(2,1,2);
plot(myData(:,2));
ylim([-0.2, 1.2]);
ax = gca;
ax.YTick = [0,1];
ax.YTickLabel = {'Low','High'};
title("Serial Clock (SCL)");
```



Analyze the I2C Physical Layer Bus Communications

Extract I2C physical layer signals on the SDA and SCL lines.

```
sda = myData(:,1)';
scl = myData(:,2)';
```

Find all rising and falling clock edges.

```
sclFlips = xor(scl(1:end-1), scl(2:end));
sclFlips = [1 sclFlips 1];
sclFlipIndexes = find(sclFlips==1);
```

Calculate the clock periods from the clock indices

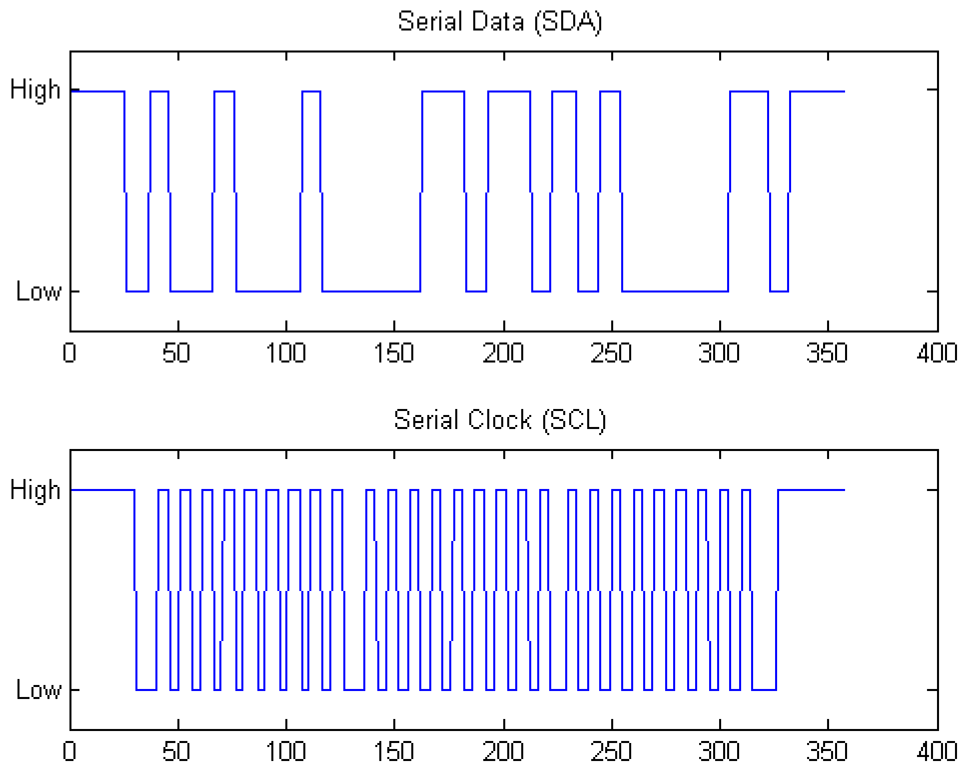
```
sclFlipPeriods = sclFlipIndexes(1:end)-[1 sclFlipIndexes(1:end-1)];
```

Through inspection, observe that idle periods have SCL high for longer than 100 us. Since scan rate = 1MS/s, each sample represents 1 us. `idlePeriodIndices` indicate periods periods of activity within the I2C communication.

```
idlePeriodIndices = find(sclFlipPeriods>100);
```

Zoom into the first period of activity on the I2C bus. For ease of viewing, include 30 samples of idle activity to the front and end of each plot.

```
range1 = sclFlipIndexes(idlePeriodIndices(1)) - 30 : sclFlipIndexes(idlePeriodIndices(2) - 1) + 3
figure("Name", "I2C Communication Data");
subplot(2,1,1);
plot(sda(range1));
ylim([-0.2, 1.2]);
ax = gca;
ax.YTick = [0,1];
ax.YTickLabel = {'Low','High'};
title("Serial Data (SDA)");
subplot(2,1,2);
plot(scl(range1));
ylim([-0.2, 1.2]);
ax = gca;
ax.YTick = [0,1];
ax.YTickLabel = {'Low','High'};
title("Serial Clock (SCL)");
```



Analyze Bus Performance Metrics

As a simple example analyze start and stop condition metrics, and I2C bit rate calculation.

- Start condition duration is defined as the time it takes for SCL to go low after SDA goes low.
- Stop condition duration is defined as the time it takes for SDA to go high after SCL goes high.
- Bit rate is calculated by taking the inverse of the time between 2 rising clock edges.

Start Condition: First SDA low, then SCL low

```
sclLowIndex = sclFlipIndexes(idlePeriodIndices(1));
sdaLowIndex = find(sda(1:sclLowIndex)==1, 1, "last") + 1; % +1, flip is next value after last hig
startConditionDuration = (sclLowIndex - sdaLowIndex) * 1/s.Rate;

fprintf('sda: %s\n', sprintf('%d ', sda(sdaLowIndex-1:sclLowIndex))); % Indexes point to next cha
fprintf('scl: %s\n', sprintf('%d ', scl(sdaLowIndex-1:sclLowIndex))); % subtract 1 from sdaLowInd
fprintf('Start condition duration: %d sec.\n\n', startConditionDuration); % count 5 pulses, 5 us
```

```
sda: 1 0 0 0 0 0 0
scl: 1 1 1 1 1 1 0
Start condition duration: 5.000000e-06 sec.
```

Stop Condition: First SCL high, then SDA high

```
% flip prior to going into idle is the one we want
sclHighIndex = sclFlipIndexes(idlePeriodIndices(2)-1);
```

```
sdaHighIndex = find(sda(sclHighIndex:end)==1, 1, 'first') + sclHighIndex - 1;
stopConditionDuration = (sdaHighIndex - sclHighIndex) * 1/s.Rate;

fprintf('sda: %s\n', sprintf('%d ',sda(sclHighIndex-1:sdaHighIndex)));
fprintf('scl: %s\n', sprintf('%d ',scl(sclHighIndex-1:sdaHighIndex)));
fprintf('Stop condition duration: %d sec.\n\n', stopConditionDuration);
```

```
sda: 0 0 0 0 0 0 1
scl: 0 1 1 1 1 1 1
Stop condition duration: 5.000000e-06 sec.
```

Bit Rate: Inverse of time between 2 rising edges on the SCL line

```
startConditionIndex = idlePeriodIndices(1);
firstRisingClockIndex = startConditionIndex + 2;
secondRisingClockIndex = firstRisingClockIndex + 2;
clockPeriodInSamples = sclFlipIndexes(secondRisingClockIndex) - sclFlipIndexes(firstRisingClockIn
clockPeriodInSeconds = clockPeriodInSamples * 1/s.Rate;
bitRate = 1/clockPeriodInSeconds;

fprintf('DAQ calculated bit rate = %d; Actual I2C object bit rate = %dKHz\n', ...
    bitRate,...
    tmp102.BitRate);
```

```
DAQ calculated bit rate = 1.000000e+05; Actual I2C object bit rate = 100KHz
```

Find the Bit Stream by Sampling on the Rising Edges

The `sclFlipIndexes` vector was created using XOR and hence contains both rising and falling edges. Start with a rising edge and use a step of two to skip falling edges.

```
% idlePeriodIndices(1)+1 is first rising clock edge after start condition.
% Use a step of two to skip falling edges and only look at rising edges.
% idlePeriodIndices(2)-1 is the index of the rising edge of the stop condition.
% idlePeriodIndices(2)-3 is the last rising clock edge in the bit stream to be
% decoded.
bitStream = sda(sclFlipIndexes(idlePeriodIndices(1)+1:2:idlePeriodIndices(2)-3));
fprintf('Raw bit stream extracted from I2C physical layer signal: %s\n\n', sprintf('%d ', bitStre
```

```
Raw bit stream extracted from I2C physical layer signal: 1 0 0 1 0 0 0 1 0 0 0 0 1 1 0 1 1 0 1 0
```

Decode the Acquired Bit Stream

```
ADR_RW = {'W', 'R'};
ACK_NACK = {'ACK', 'NACK'};
address = bitStream(1:7); % 7 bit address
fprintf('\nDecoded Address: %d%d%d%d%d%d%d(0x%s) %d(%s) %d(%s)\n', ...
    address,...
    binaryVectorToHex(address),...
    bitStream(8),...
    ADR_RW{bitStream(8)+1},...
    bitStream(9),...
    ACK_NACK{bitStream(9)+1});
for iData = 0:1
    startBit = 10 + iData*9;
    endBit = startBit + 7;
    ackBit = endBit + 1;
```

```
    data = bitStream(startBit:endBit);
    fprintf('Decoded Data%d: %s(0x%s) %d(%s)\n', ...
        iData+1,...
        sprintf('%d', data),...
        binaryVectorToHex(data),...
        bitStream(ackBit),...
        ACK_NACK{bitStream(ackBit)+1});
end
```

```
Decoded Address: 1001000(0x48) 1(R) 0(ACK)
Decoded Data1: 00011011(0x1B) 0(ACK)
Decoded Data2: 10100000(0xA0) 1(NACK)
```

Verify That the Decoded Data Using DAQ Matches the Data Read Using ICT

Two `uint8` bytes were read, using `fread`, from the I2C bus into variable `data8`. The hex conversion of these values should match the results of the bus decode shown above.

```
fprintf('Data acquired from I2C object: 0x%s\n', dec2hex(data8)');
fprintf('Temperature: %2.2f deg. C\n\n', temperature);
```

```
Data acquired from I2C object: 0x1BA0
Temperature: 27.63 deg. C
```

Synchronize NI PCI Devices Using RTSI

This example shows how to acquire synchronized data from two PCI devices. A sine wave is connected to channel 0 of NI PCI-6251 and to channel 0 of NI PCIe-6363. Synchronized operation is verified by demonstrating zero phase lag between the acquired signals.

Create DataAcquisition and Add Analog Input Channel

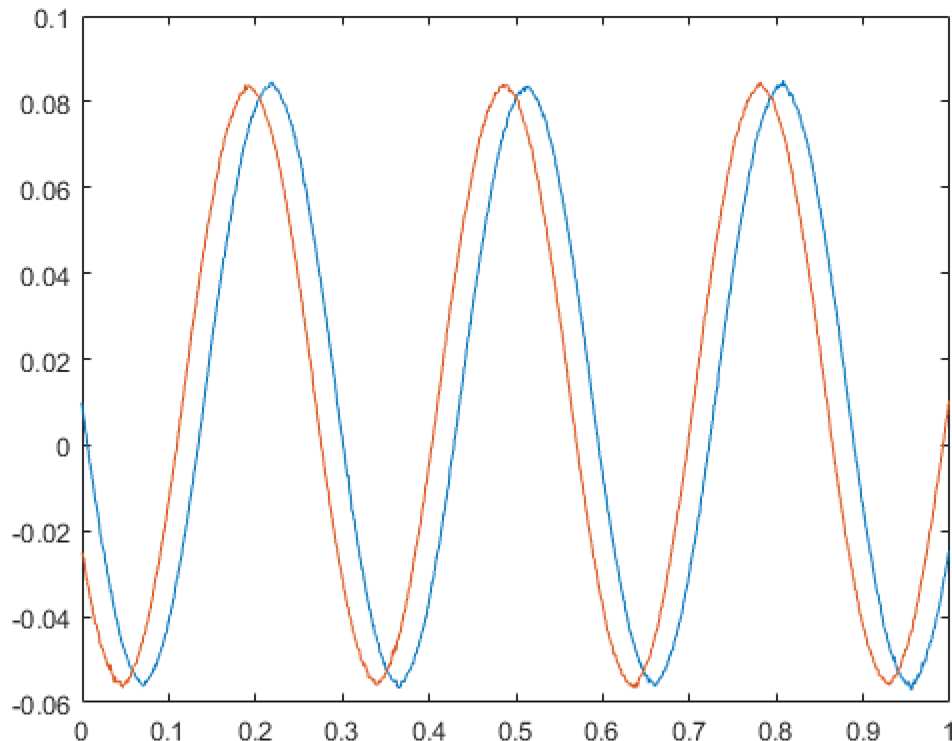
Create a DataAcquisition and add analog input voltage input channels from NI PCI-6251 and NI PCIe-6363 devices.

```
dd = daq("ni");
addinput(dd,"Dev3","ai0","Voltage");
addinput(dd,"Dev4","ai0","Voltage");
```

Acquire Unsynchronized Data

Use the `read` command to start the acquisition.

```
[data,time] = read(dd,seconds(1),"OutputFormat","Matrix");
plot(time, data)
```



There is a small phase lag between the two channel inputs. The DataAcquisition starts the two channels close together, but the devices do not share any clock and trigger information and therefore are not fully synchronized.

Set Up Hardware Connections

Connect PCI devices using a RTSI® (Real-Time System Integration) cable and register it in Measurement & Automation Explorer®. To synchronize the acquisition, share a scan clock and start trigger between the two devices.

Choose Source and Destination Devices

The device that provides the control and timing signals is called the source device, and the device that receives these signals is called destination device. In this example, `Dev3` is the source device and `Dev4` is the destination device.

Add Start Trigger

The RTSI cable creates a physical connection between the `RTSI0` terminal on `Dev3` and `RTSI0` terminal on `Dev4`. Use this connection to share the start trigger between the source and destination devices.

Use `addtrigger` to add a digital start trigger from `'RTSI0/PFI3'` (source) to `'RTSI0/Dev4'` (destination).

```
addtrigger(dd,"Digital","StartTrigger","Dev3/RTSI0","Dev4/RTSI0");
```

Add Scan Clock

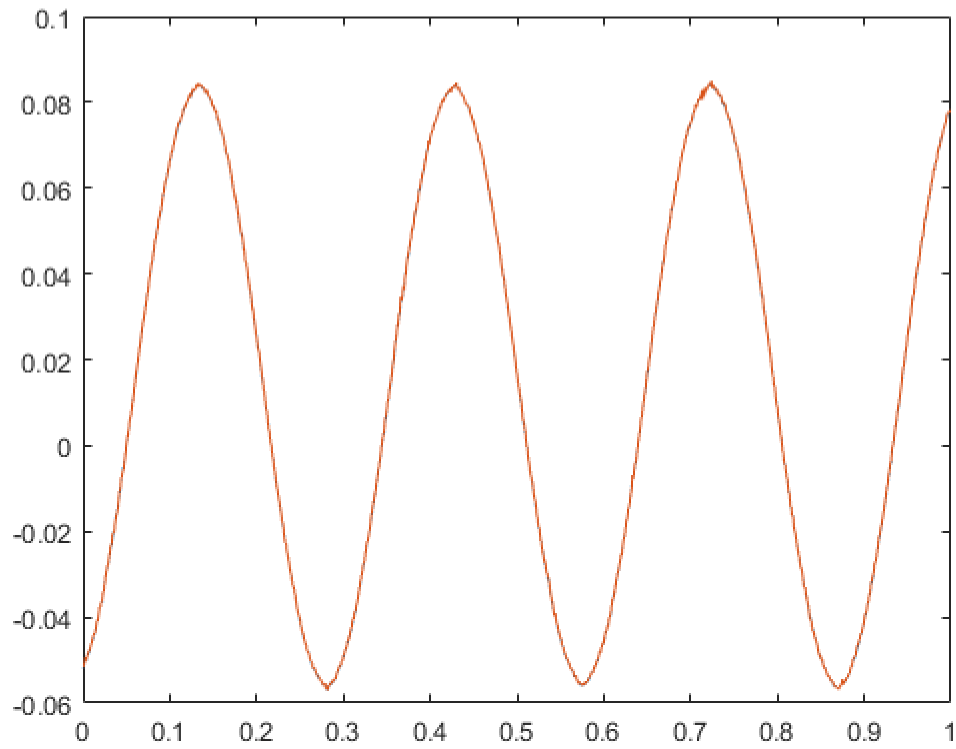
Use `addclock` to share a scan clock using the `RTSI1` terminal connection.

```
addclock(dd,"ScanClock","Dev3/RTSI1","Dev4/RTSI1");
```

Acquire Data with Synchronization

Use `read` to acquire data.

```
[data,time] = read(dd,seconds(1));
plot(time,data)
```



The two sine waves are overlapping with zero phase lag, confirming that the devices are fully synchronized.

Start a Multi-Trigger Acquisition on an External Event

This example shows how to set up and start a multi-trigger acquisition on an external event. In this instance, the device is configured to start acquiring data on a rising edge signal.

Create a DataAcquisition and Add Analog Input Channels

Create a DataAcquisition object, and add an analog input channel with the `Voltage` measurement type, using an NI PCIe 6363, with ID `Dev4`.

```
dq = daq("ni");
addinput(dq,"Dev4","ai0","Voltage");
```

Configure the DataAcquisition to Start on an External Trigger

Configure the device to acquire data on the external trigger. A trigger that starts an acquisition is called a Start Trigger. In this example, the switch is wired to terminal `PFI0` on device `Dev4`. Represent this physical connection (between the switch and terminal `PFI0`) as a start trigger.

Add Digital Start Trigger

A trigger has a trigger type (`Digital`). The allowed value for the `Digital` trigger type is `StartTrigger`.

A trigger has a source and a destination. In this example, the source is the switch (choose `'External'` as the source). The destination is the `PFI0` terminal on `Dev4` (`'PFI0/Dev4'`). Use `addtrigger` to add this trigger on the DataAcquisition.

```
addtrigger(dq,"Digital","StartTrigger","External","Dev4/PFI0");
dq.DigitalTriggers
```

```
ans = 

  DigitalTrigger with properties:

         Source: "External"
    Destination: 'Dev4/PFI0'
           Type: 'StartTrigger'
      Condition: 'RisingEdge'
```

Set Trigger Parameters

By default the DataAcquisition waits for 10 seconds for the rising edge digital trigger. Increase the timeout to 30 seconds using `DigitalTriggerTimeout` property.

```
dq.DigitalTriggerTimeout = 30;
```

You can configure a DataAcquisition to receive multiple triggers, when it should respond to multiple events. In this example, two external trigger signals are expected, enabling the device `Dev4` to start acquiring scans on receipt of the second trigger.

```
dq.NumDigitalTriggersPerRun = 2;
```

Start the Acquisition

Use `read` to acquire scans on receipt of each configured digital start trigger. The specific sequence of events is:

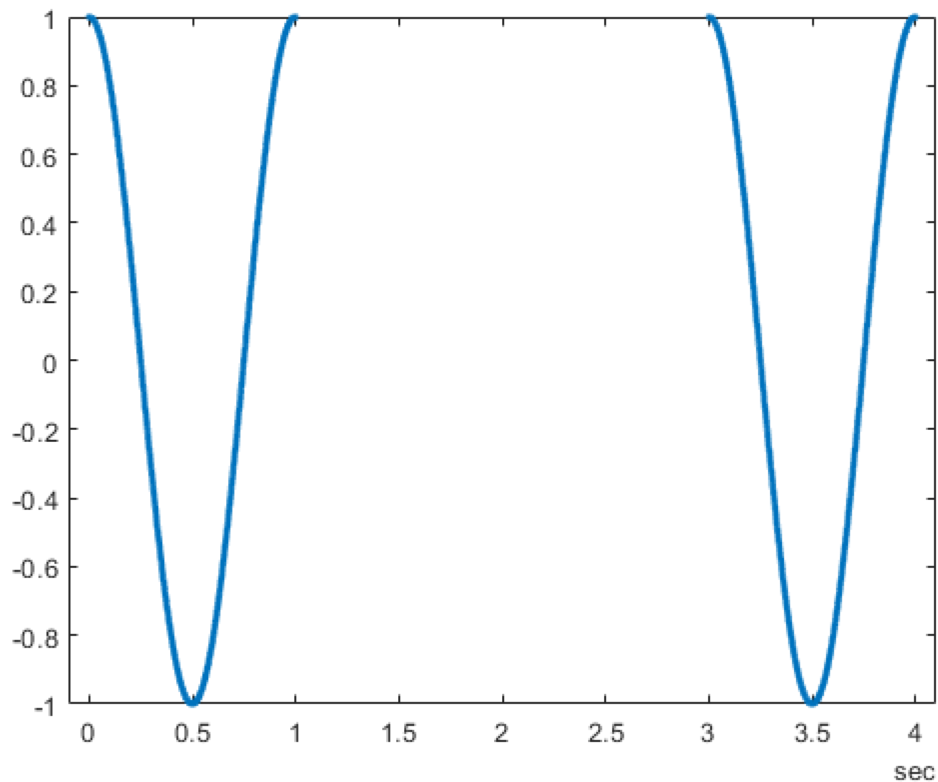
1. The DataAcquisition starts
2. One second of actual acquisition begins on receipt of the first trigger unless the timeout period expires
3. One second of actual acquisition begins on receipt of the second trigger unless the timeout period expires
4. Data is returned

```
[data, startTime] = read(dq, seconds(1));
```

Plot the Data

Observe the discontinuity based on the time between the two trigger starts.

```
plot(data.Time, data.Variables, '.')
```

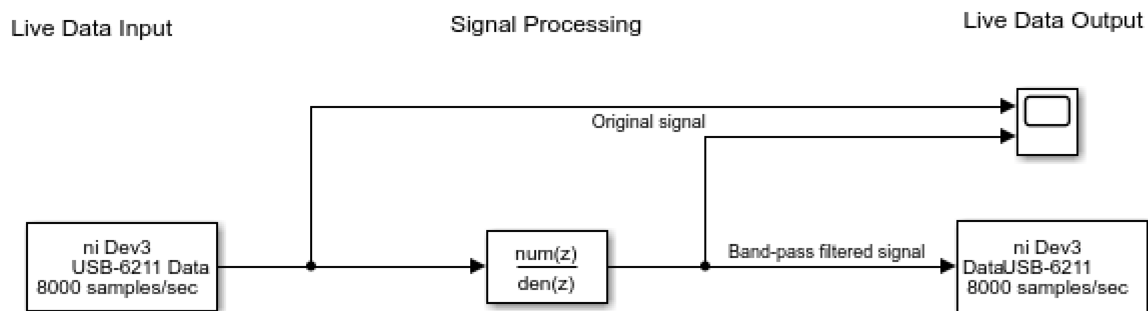


Perform Live Acquisition, Signal Processing, and Generation

This example shows how to use the Analog Input block to acquire live analog data from a data acquisition device into Simulink. The acquired data is processed in Simulink and uses the Analog Output block to output data to a data acquisition device. It shows how a Simulink model can communicate with different subsystems in the same model. In this case, the data acquisition device used is from National Instruments®.

Note: This example requires MATLAB®, Data Acquisition Toolbox, and Simulink to open and run the model.

Data Acquisition, Processing and Output



Copyright 2016 The MathWorks, Inc.

Digital filters
This block is part of the built-in "Discrete" Simulink library. Here it uses *pre-computed* coefficients to implement a bandpass filter with a bandwith of 50Hz around 100Hz and an attenuation of 60dB elsewhere, for input signals sampled at 8kHz.
To *design* digital filters, to select specific implementation structures, or to simulate fixed-point arithmetic use blocks from the "Filtering" library in DSP System Toolbox

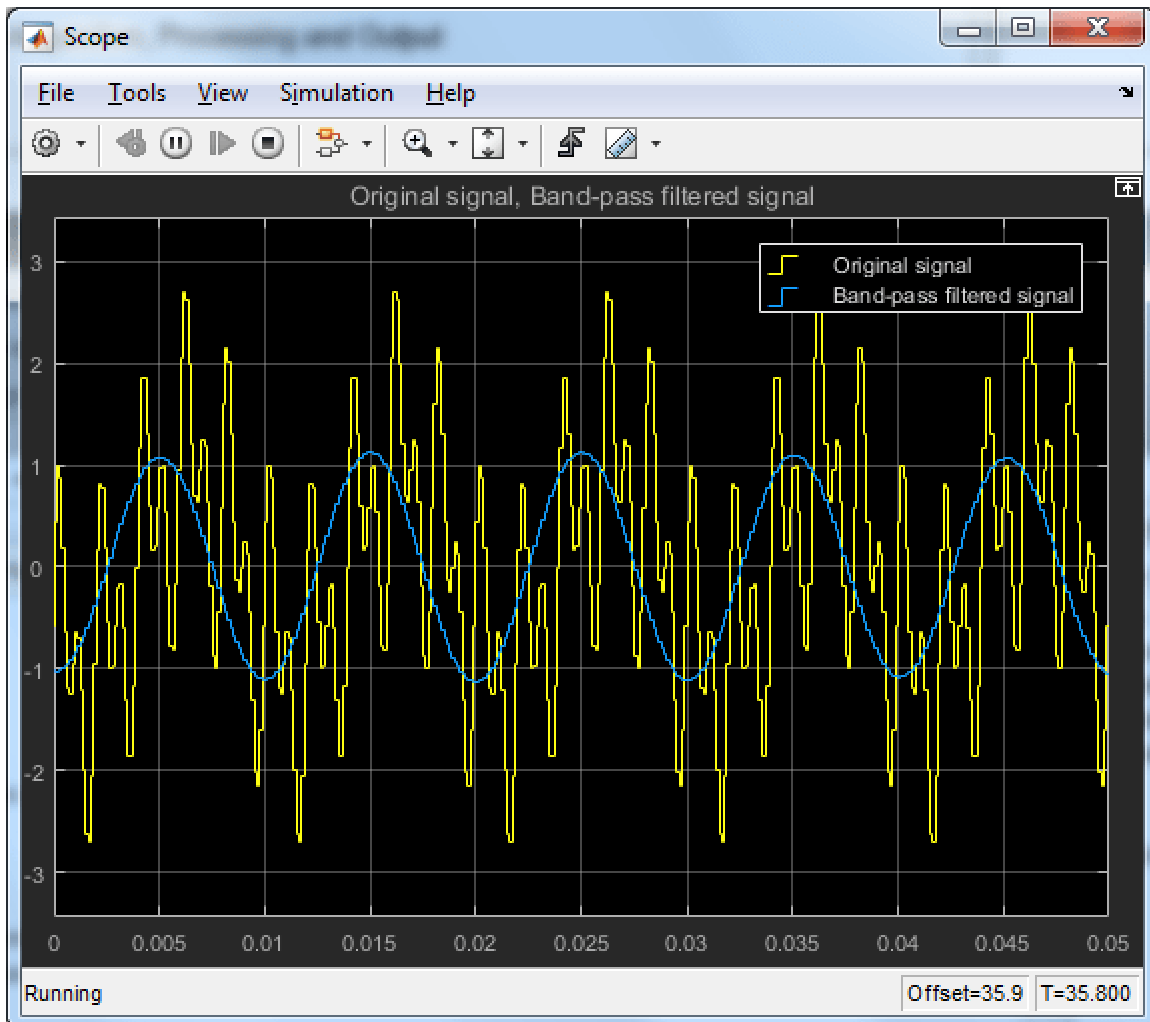
Live Data Input

The input signal is acquired from a National Instruments data acquisition device (USB-6211). Signal is acquired from channel ai0 at the rate of 8000 samples/second. The Analog Input block is configured to do synchronous acquisition, which does blocking read from the device and at each timestep it acquires a chunk of 1600 samples from the hardware.

Note: Each column in the output of Analog Input block corresponds to data from an analog input channel. To correctly interpret the data, in the downstream processing/visualization blocks, you need to use 'Columns as Channels (frame-based)' as Input Processing method.

Signal Processing

The acquired data is processed using a discrete filter. The discrete filter uses pre-computed coefficients to implement a bandpass filter with a bandwidth of 50Hz around 100Hz and an attenuation of 60dB elsewhere, for input signals sampled at 8kHz. To design digital filters, to select specific implementation structures, use blocks from the "Filtering" library in DSP System Toolbox. The capture data has three major frequency components: sine waves at 100Hz, 500Hz and 1000Hz. After the discrete filter, you would see a clear 100Hz sine wave in the output. A plot of the input and the filtered signals are shown below.



Live Data Output

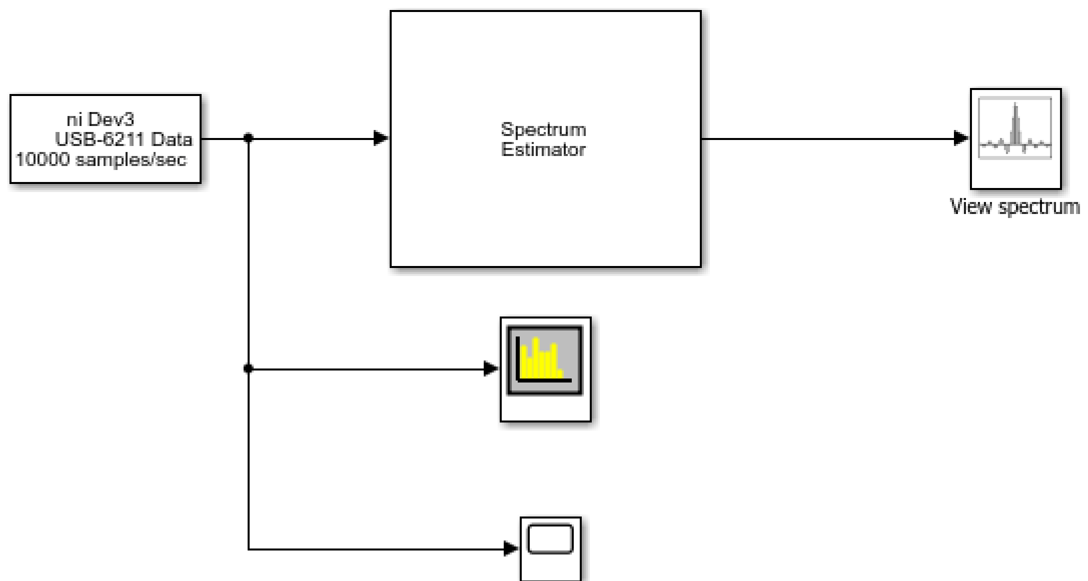
The processed data is output to a single channel of a National Instruments device (PCI-6211) at a rate of 8000 samples/second.

Even though a National Instruments device was used for this example, this model can be easily updated to connect to other supported data acquisition devices. This provides you the flexibility to reuse the same Simulink model with different data acquisition hardware.

Perform Spectral Analysis on Live Data

This example shows how to use the Analog Input block to acquire live signals from a data acquisition device into Simulink. The block uses a National Instruments(R) USB-6211 as the input device. The Simulink model uses a spectrum estimator to output a power spectrum estimate of a time-domain input using Welch's method of averaged modified periodograms.

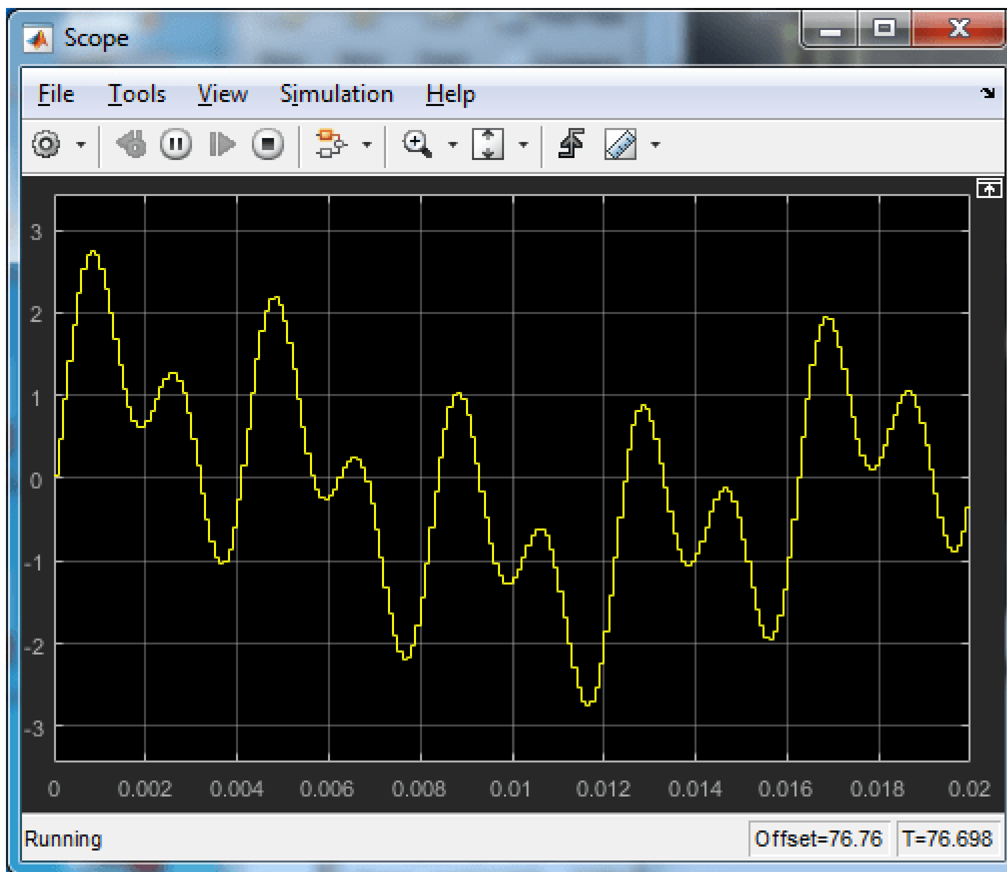
Note: This example requires MATLAB®, Simulink, Data Acquisition Toolbox and DSP System Toolbox™ to open and run the model.

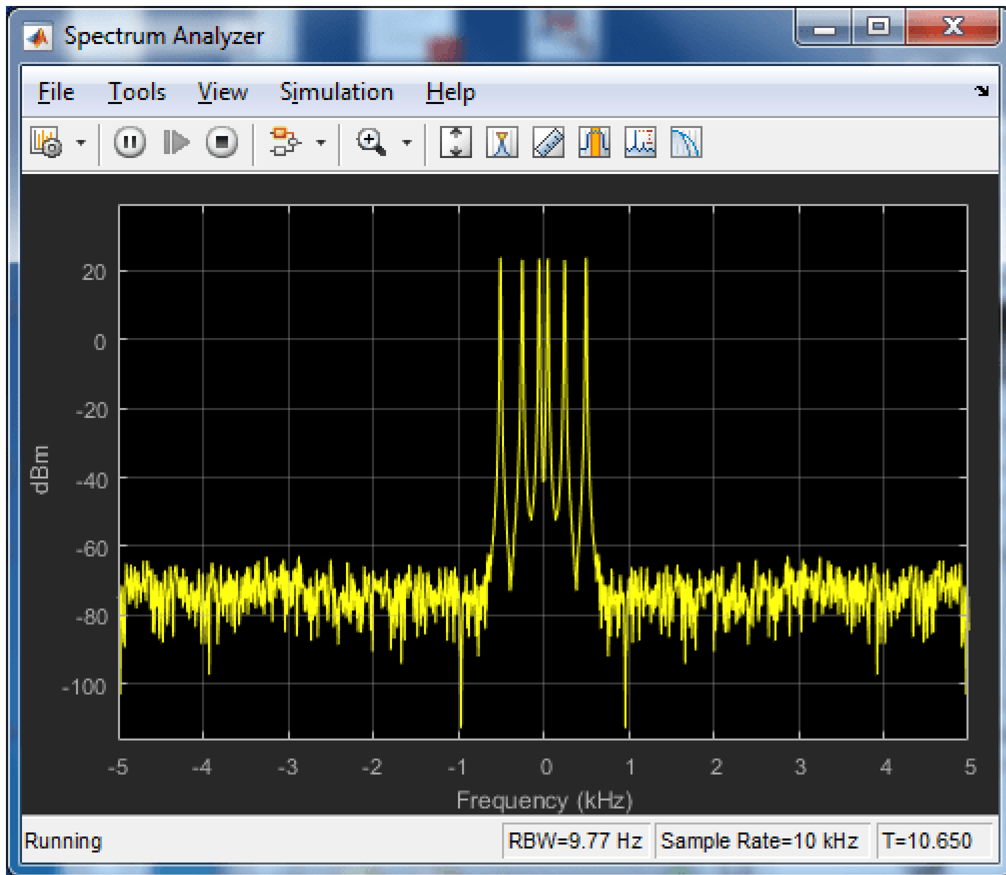


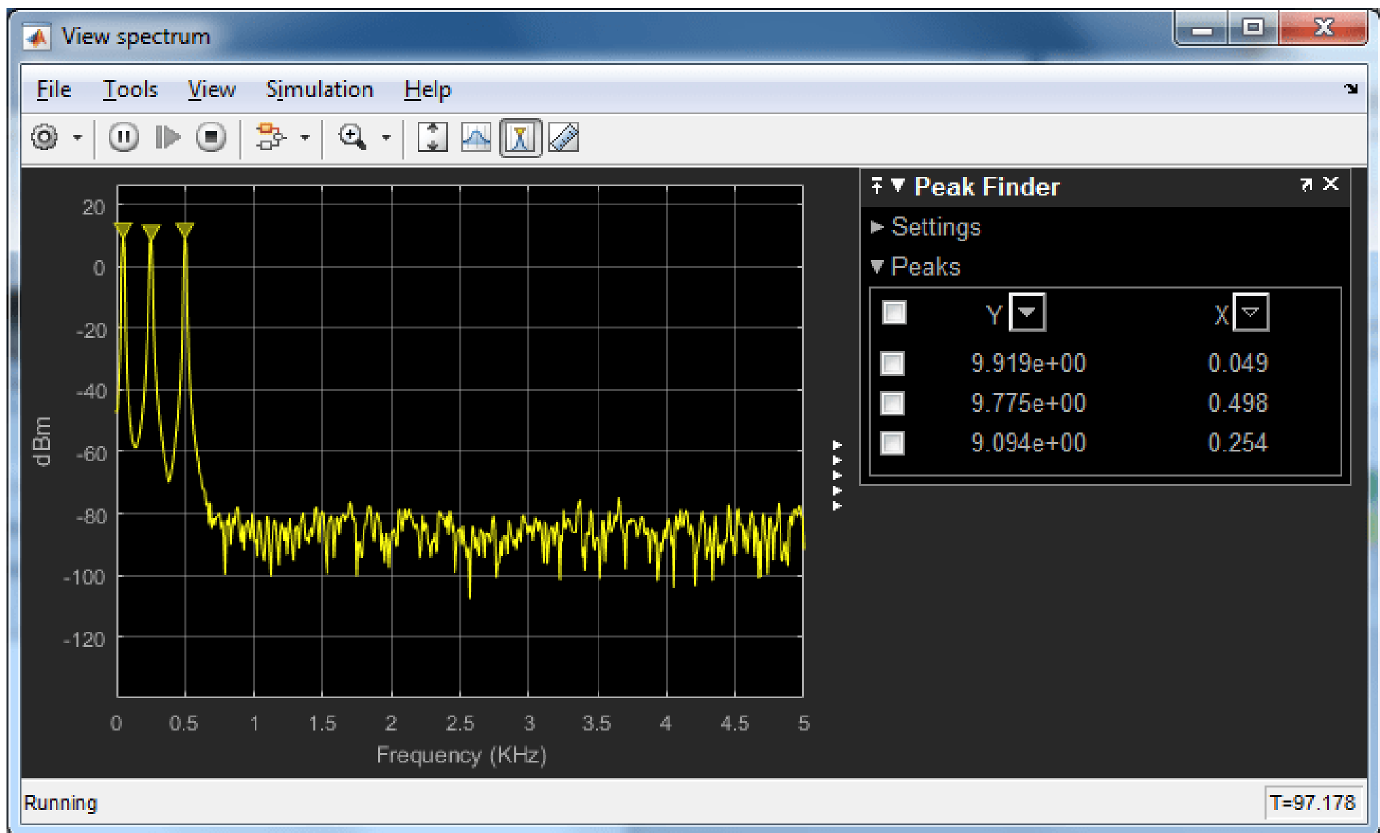
Data Acquisition and Processing

The input signal is a real-time analog signal sampled at 10000 samples per second. The Analog Input block is configured to do asynchronous acquisition, which buffers the data from the analog channels and streams the buffered data to Simulink. Each timestep, the Analog Input block outputs a chunk of 1024 samples. Each chunk of data is processed by a Spectrum Estimator to calculate the power spectrum. You can adjust the settings in the Spectrum Estimator such as different window functions.

In this example the captured signal contains three major frequency components: sine waves at 50Hz, 250Hz, and 500Hz. The time domain signal, frequency domain signal from Spectrum Analyzer, and the one-side power spectrum estimate by spectrum estimator are shown below.


Scope
File
Tools
View
Simulation
Help
3
2
1
0
-1
-2
-3
0
0.002
0.004
0.006
0.008
0.01
0.012
0.014
0.016
0.018
0.02
Running
Offset=76.76
T=76.698


Spectrum Analyzer
File Tools View Simulation Help
20
0
-20
-40
-60
-80
-100
dBm
-5 -4 -3 -2 -1 0 1 2 3 4 5
Frequency (kHz)
Running
RBW=9.77 Hz
Sample Rate=10 kHz
T=10.650



Even though a National Instruments device was used for this example, this model can be easily updated to connect to other supported data acquisition devices. This provides you the flexibility to reuse the same Simulink model with different data acquisition hardware.

Acquire Data from Two Devices at Different Rates

This example shows how to acquire data from two different DAQ devices running at different sampling rates. The example uses two National Instruments CompactDAQ analog input modules (9201 and 9211) that have different acquisition rate limits. The 9211 module is used for temperature measurements and acquires at a slower rate (10 Hz) than the 9201 module, which is used to measure voltage (100 Hz). Since all channels in a data acquisition object must acquire at the same rate, to acquire from two modules at multiple rates you need to use two data acquisition objects. To make both DAQ devices start simultaneously, you can use a hardware digital triggering configuration.

Hardware Setup

- CompactDAQ chassis NI cDAQ 9178 ('cDAQ1')
- NI cDAQ 9211 module with thermocouple measurement type ('cDAQ1Mod1')
- NI cDAQ 9201 module with voltage measurement type ('cDAQ1Mod2')
- Thermocouple probe (type K)
- Analog voltage signal generated by a function generator instrument

Configure Data Acquisition Objects and Channels

Create two data acquisition objects, each with one analog input channel from a 9211 module or 9201 module. The data acquisition objects acquire data at rates of 10 Hz and 100 Hz, respectively.

```
% Specify a common acquisition duration for both devices, in seconds
daqDuration = seconds(3);

% Create and configure DataAcquisition object and channels for cDAQ 9211 module
d1 = daq('ni');
addinput(d1, 'cDAQ1Mod1', 'ai0', 'Thermocouple');
d1.Channels(1).ThermocoupleType = 'K';
d1.Rate = 10;
```

```
Warning: The Rate property was reduced to 14.2857 due to changes in the channel
configuration.
```

```
% Create and configure DataAcquisition object and channels for cDAQ 9201 module
d2 = daq('ni');
addinput(d2, 'cDAQ1Mod2', 'ai0', 'Voltage');
d2.Rate = 100;
```

Configure Trigger Connections

To synchronize the acquisition start you can use hardware triggering and a source/destination approach. One of the data acquisition objects (source) is started manually and triggers the acquisition start of the other data acquisition object (destination).

Note: If you have a CompactDAQ chassis model (such as NI 9174) which does not have PFI triggering terminals, you can use an additional digital I/O module (such as NI 9402) to provide the PFI terminals for the trigger connections.

```
% Configure the source data acquisition object to export a triggering
% signal on the PFI0 terminal of cDAQ1 chassis
addtrigger(d1, 'Digital', 'StartTrigger', 'cDAQ1/PFI0', 'External');
```

```
% Configure the destination data acquisition object to start acquisition when an
% external triggering signal is received at PFI0 terminal of cDAQ1 chassis
addtrigger(d2, 'Digital', 'StartTrigger', 'External', 'cDAQ1/PFI0');
```

Start Acquisition and Wait Until Complete

The destination data acquisition object must start first and be ready for an external trigger before the source data acquisition object starts.

```
start(d2, 'Duration', daqDuration)
while ~d2.WaitingForDigitalTrigger
    pause(0.1)
end
start(d1, 'Duration', daqDuration)

% Wait until data acquisition is complete
while d1.Running || d2.Running
    pause(1)
end
```

```
Background operation has started.
Background operation will stop after 3 s.
To read acquired scans, use read.
Background operation has started.
Background operation will stop after 3 s.
To read acquired scans, use read.
```

Save Data as Timetable

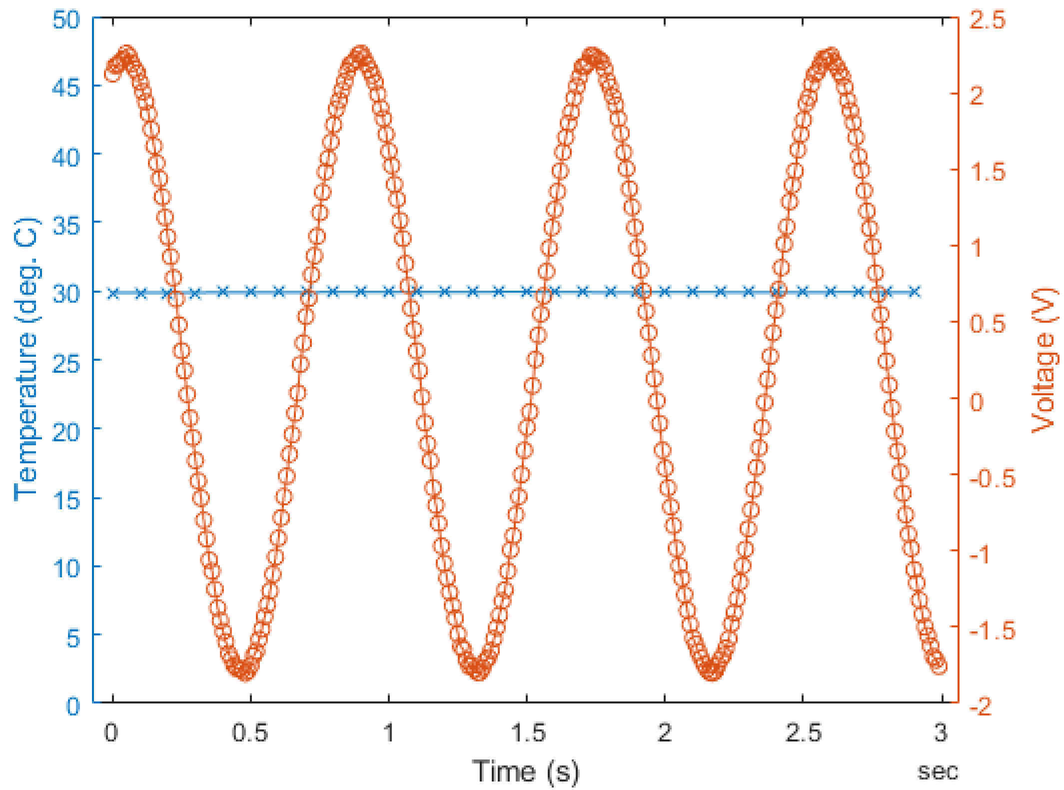
For each data acquisition object, the acquired measurement data and timestamps were stored in memory. Read all acquired data from memory in the default timetable format.

```
data1 = read(d1, 'all');
data2 = read(d2, 'all');
```

Plot Acquired Data

Because the acquired data from the two devices have different scales and units, create a chart with two y-axes.

```
figure
yyaxis left
plot(data1.Time, data1.Variables, '-x')
ylabel('Temperature (deg. C)')
ylim([0 50])
yyaxis right
plot(data2.Time, data2.Variables, '-o')
ylabel('Voltage (V)')
xlabel('Time (s)')
```



Clean Up

Clear the data acquisition objects to disconnect from hardware.

```
clear d1 d2
```

Characterize an LED with ADALM1000

This example shows how to use MATLAB to connect to an Analog Devices ADALM1000 source-measurement unit, configure it, and make current and voltage measurements to characterize an LED.

Discover Supported Data Acquisition Devices Connected to Your System

```
daqlist
```

```
ans=1×5 table
    VendorID    DeviceID              Description                  Model             DeviceInfo
    ________    ________    _________________________________    ___________    ____________________

     "adi"       "SMU1"     "Analog Devices Inc. ADALM1000"     "ADALM1000"    [1×1 daq.adi.Device
```

Create a DataAcquisition Interface for the ADALM1000 Device

```
ADIDaq = daq("adi");
```

Add Channels for Sourcing Voltage and Measuring Current

The ADALM1000 device is capable of sourcing voltage and measuring current simultaneously on the same channel. Set up the device in this mode.

Add an analog output channel with device ID SMU1 and channel ID A, and set its measurement type to voltage.

```
addoutput(ADIDaq,'smu1','a','Voltage');
```

Add an analog input channel with device ID SMU1 and channel ID A, and set its measurement type to current.

```
addinput(ADIDaq,'smu1','a','Current');
```

Confirm the configuration of the channels.

```
ADIDaq.Channels
```

```
ans=1×2 object
    Index    Type    Device    Channel      Measurement Type             Range              Name
    _____    ____    ______    _______    _____________________    __________________    ________

      1      "ao"    "SMU1"      "A"      "Voltage (SingleEnd)"    "0 to +5.0 Volts"     "SMU1_A'
      2      "ai"    "SMU1"      "A"      "Current"                "-0.20 to +0.20 A"    "SMU1_A_
```

Blink Attached LED Five Times

Connect an LED in series with a 330-Ω resistor between the ADALM1000 channel A and ground. Alternately apply 5 V and 0 V.

```
for iLoop = 1:5
    % Turn on LED by generating an output of 5 volts.
    write(ADIDaq,5);
    pause(0.2);
    % Turn off LED by generating an output of 0 volts.
```

```
    write(ADIDaq,0);
    pause(0.2);
end
```

Characterize the LED

To understand the LED's I-V characteristics, sweep a range of voltage values from 0 V to 5 V, and measure the current for each value. The aggregate of all measurements provides data to graph the current across the LED over a range of voltages.

```
v = linspace(0,5,250)';
i = readwrite(ADIDaq,v,"OutputFormat","Matrix");
```

Plot the Characteristic Curve of the LED and Estimate a Mathematical Model

When you have the measured data, you can visualize it. You can also calculate a mathematical model that approximates the behavior of the LED within the range of the measurements.

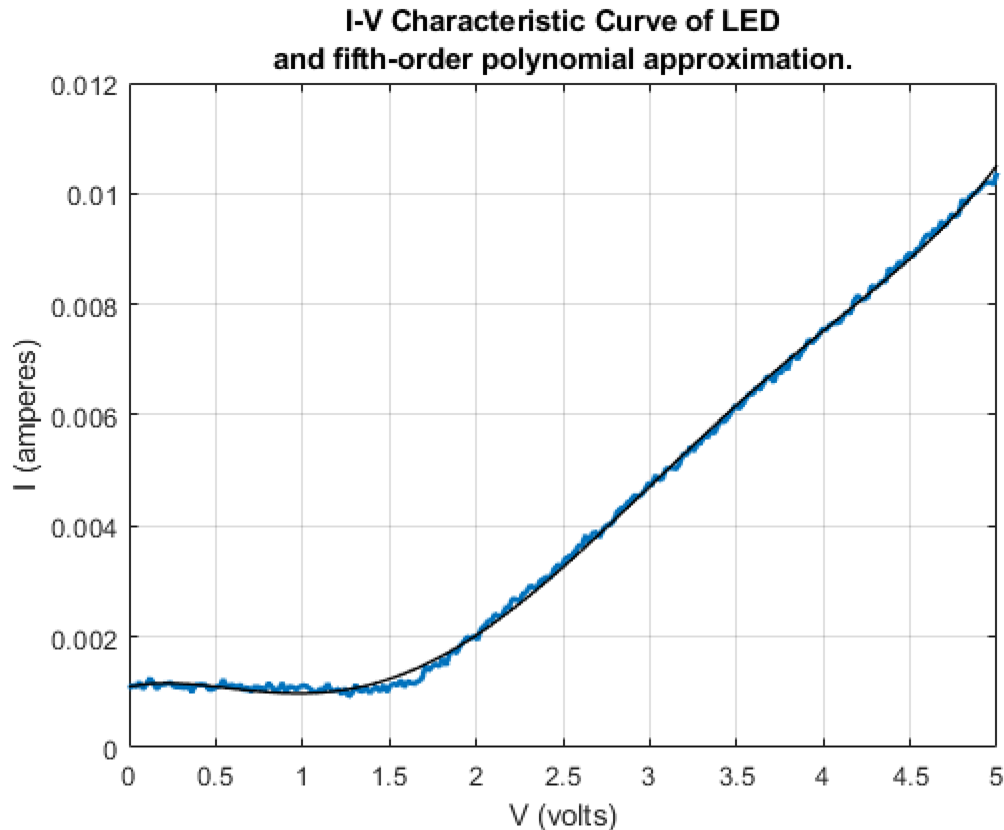
```
% Plot the measured data.
plot(v,i,'LineWidth',2);
hold on;
grid on;
ylabel('I (amperes)');
xlabel('V (volts)');
title({'I-V Characteristic Curve of LED';'and fifth-order polynomial approximation.'});
```

Fit the data using a fifth-order polynomial and overlay the acquired data with the model of the LED approximated by a fifth-order polynomial.

```
approxPoly = polyfit(v,i,5);
```

Plot the approximated data.

```
plot(v,polyval(approxPoly,v),'-k','Linewidth',1);
```



Calculate the Voltage at Which the LED Turns On

Based on the fifth-order polynomial approximation, you can find a first-order approximation that represents the linearly increasing portion of the curve. The voltage at which the LED turns on is approximately where this line intersects the voltage axis.

Find the line that passes through the linear portion of the signal.

```
normErr = -1;
errThreshold = 0.001;
numPointsForLine = numel(v) - 10;
while (numPointsForLine > 0) && (normErr < errThreshold)
    approximation = polyval(approxPoly,v(numPointsForLine:end));
    [linearPoly, errorStruct] = polyfit(v(numPointsForLine:end),approximation, 1);
    numPointsForLine = numPointsForLine - 5;
    normErr = errorStruct.normr;
end
```

Evaluate the linear polynomial over the range of measurements. The value where this intersects the horizontal line representing any leakage current is the voltage at which the LED turns on.

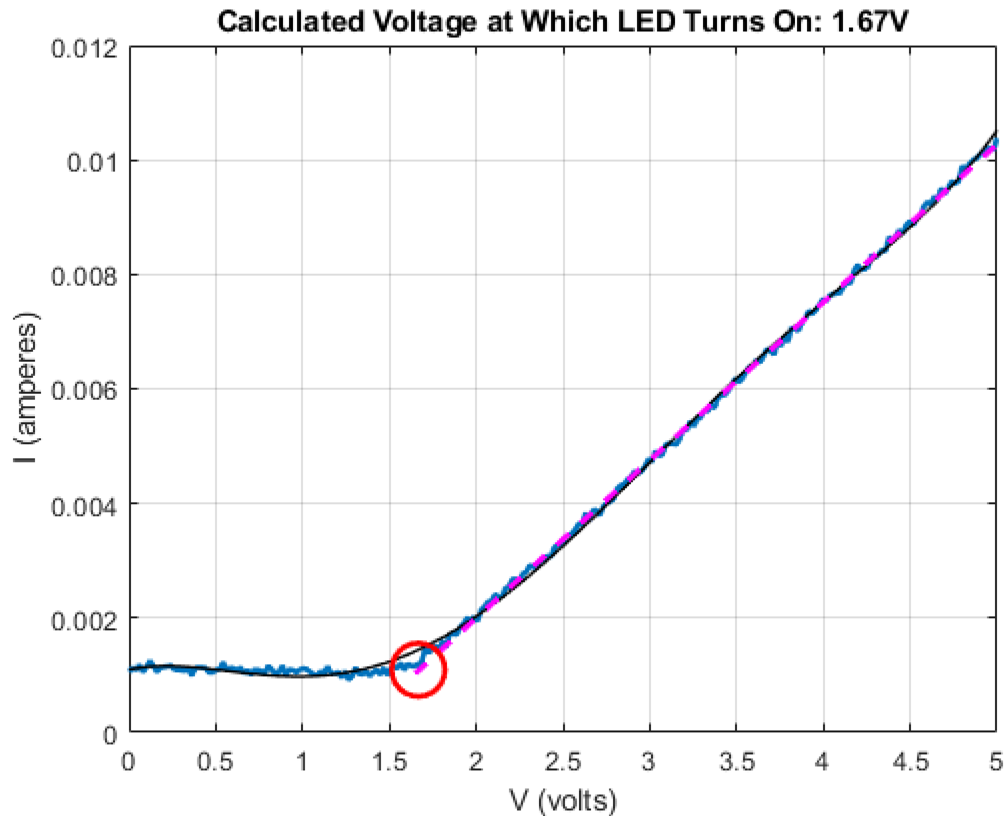
```
LEDThreshold = 1.2;
leakageCurrent = mean(i(v<LEDThreshold));
linearIV = polyval(linearPoly,v);
minIndex = sum(linearIV<leakageCurrent);
```

Plot the linear portion of the curve.

```
plot(v(minIndex-1:end),polyval(linearPoly,v(minIndex-1:end)),'Magenta','Linewidth',2,'LineStyle'
```

Circle the approximate voltage at which the LED turns on.

```
plot(v(minIndex),leakageCurrent,'o','Linewidth',2,'MarkerSize',20,'MarkerEdgeColor','Red')
title(sprintf('Calculated Voltage at Which LED Turns On: %1.2fV',v(minIndex)));
```



Turn Off the LED and Clear the DataAcquisition

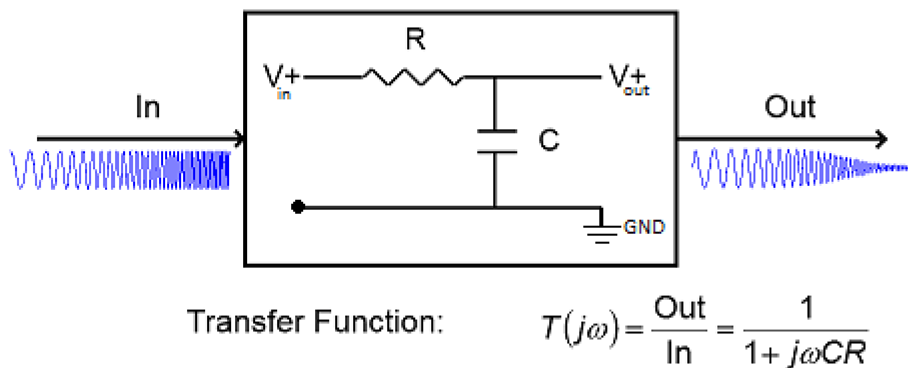
```
write(ADIDaq,0);
close
clear ADIDaq
```

Estimate the Transfer Function of a Circuit with ADALM1000

This example shows how to use MATLAB to connect to an ADALM1000 source-measurement unit, configure it to generate an arbitrary signal, make live measurements, and use the measurements to calculate the transfer function of the connected circuit.

Introduction

In this example you have an R-C circuit consisting of a 1 kΩ resistor in series with a 0.1 μF capacitor. The R-C circuit is attached to the ADALM1000 device with Channel A of the device providing the voltage stimulus consisting of a chirp signal. The output of Channel A is connected to the resistor, and the ground is connected to the capacitor. Channel B of the device is used to measure the voltage across the capacitor. The following circuit diagram shows the measurement setup.



$$T(j\omega) = \frac{\text{Out}}{\text{In}} = \frac{1}{1 + j\omega CR}$$

You can use the stimulus and the measured waveforms to estimate the transfer function of the R-C circuit, and compare the measured response to the theoretical response of the R-C circuit.

Discover Devices Connected to Your System

```
daqlist("adi")
```

```
ans=1×4 table
    DeviceID               Description                  Model              DeviceInfo
    ________    ________________________________    ___________    _________________________

     "SMU1"     "Analog Devices Inc. ADALM1000"     "ADALM1000"    [1×1 daq.adi.DeviceInfo]
```

Create a DataAcquisition for the ADALM1000 Device

```
ADIDaq = daq("adi")
```

```
ADIDaq = 
DataAcquisition using Analog Devices Inc. hardware:

                     Running: 0
                        Rate: 100000
           NumScansAvailable: 0
            NumScansAcquired: 0
```

```
              NumScansQueued: 0
    NumScansOutputByHardware: 0
                   RateLimit: []

Show channels
Show properties and methods
```

Add Voltage Source and Measurement Channels

The ADALM1000 device is capable of sourcing and measuring voltage simultaneously on separate channels. Set up the device in this mode.

To source voltage, add an analog output channel with device ID SMU1 and channel ID A, and set its measurement type to Voltage.

```
addoutput(ADIDaq,'smu1','a','Voltage');
```

To measure voltage, add an analog input channel with device ID SMU1 and channel ID B, and set its measurement type to Voltage.

```
addinput(ADIDaq,'smu1','b','Voltage');
```

Confirm the configuration of the channels.

```
ADIDaq.Channels
```

```
ans=1×2 object
    Index    Type    Device    Channel      Measurement Type             Range              Name
    _____    ____    ______    _______    _____________________    ________________    _______

      1      "ao"    "SMU1"      "A"      "Voltage (SingleEnd)"    "0 to +5.0 Volts"    "SMU1_A"
      2      "ai"    "SMU1"      "B"      "Voltage (SingleEnd)"    "0 to +5.0 Volts"    "SMU1_B"
```

Define and Visualize a Chirp Waveform for Stimulating the Circuit

Use a chirp waveform of 1 volt amplitude, ranging in frequency from 20 Hz to 20 kHz for stimulating the circuit. The chirp occurs in a period of 1 second.

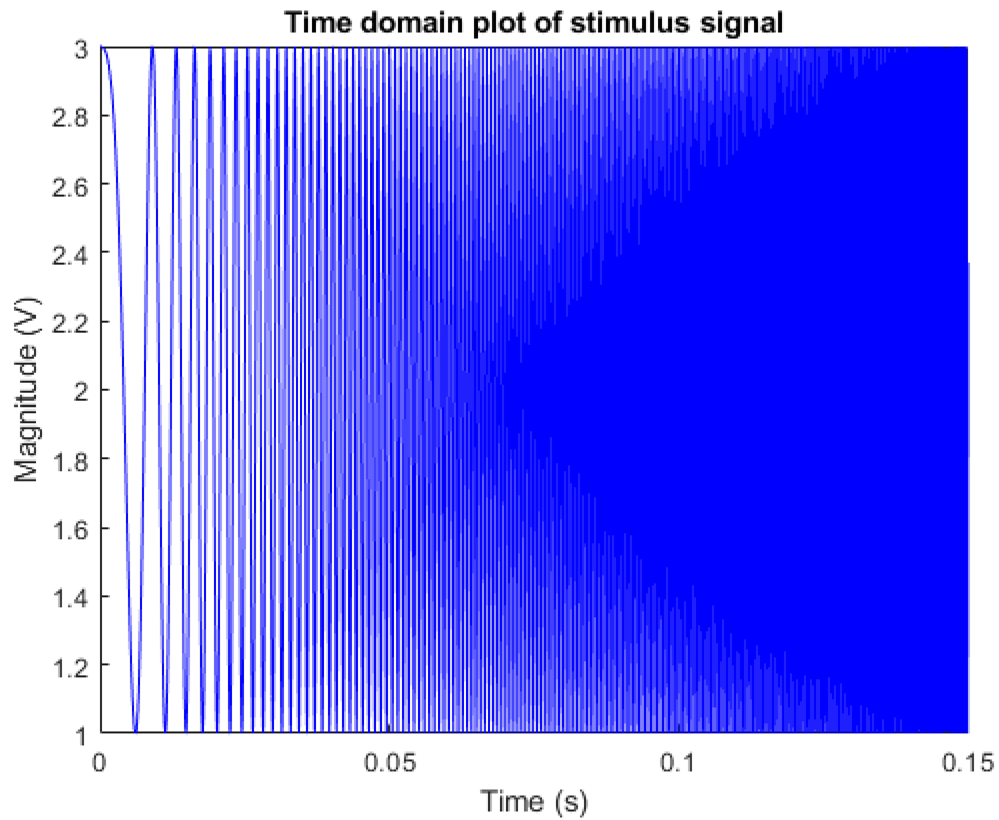
```
Fs = ADIDaq.Rate;
T = 0:1/Fs:1;
ExcitationSignal = chirp(T,20,1,20e3,'linear');
```

Add a DC offset of 2 V to ensure that the output voltage of the device is always above 0 V.

```
Offset = 2;
ExcitationSignal = ExcitationSignal + Offset;
```

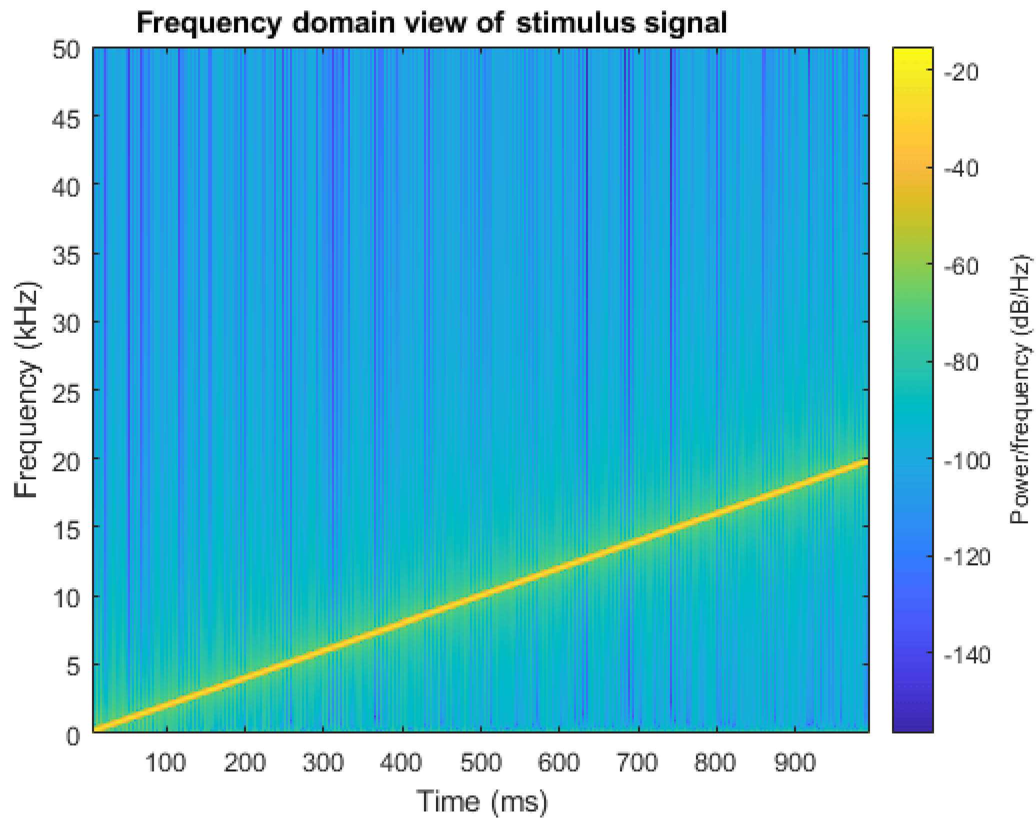
Visualize the stimulus signal in the time-domain.

```
figure(1)
plot(T, ExcitationSignal,'Blue')
xlim([0 0.15])
xlabel('Time (s)')
ylabel('Magnitude (V)')
title('Time domain plot of stimulus signal')
```



Visualize the stimulus signal in the frequency domain.

```
figure(2)
spectrogram(ExcitationSignal,1024,1000,1024,Fs,'yaxis')
title('Frequency domain view of stimulus signal')
```



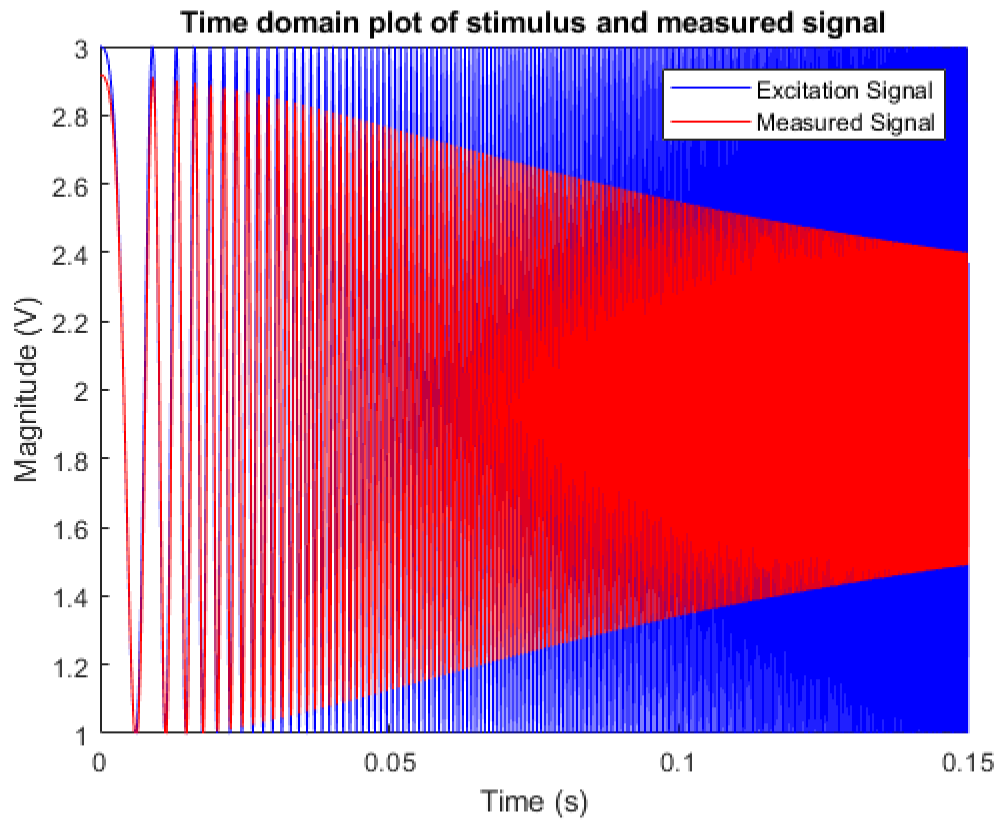
Stimulate the Circuit and Simultaneously Measure the Frequency Response

Generate device output and measure measure the voltage across the capacitor at the same time on the other channel.

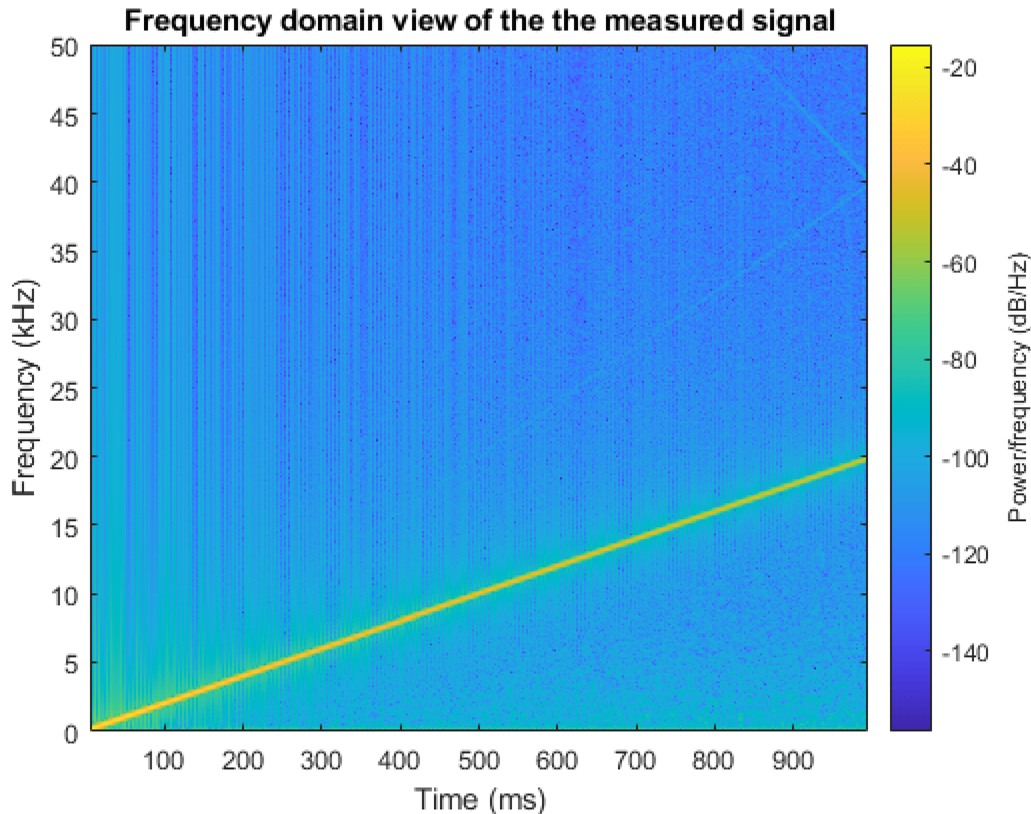
```
MeasuredTable = readwrite(ADIDaq,ExcitationSignal');
MeasuredSignal = MeasuredTable{:,1};
```

Plot the Stimulus and the Measured Signals

```
figure(1)
hold on;
plot(T,MeasuredSignal,'Red');
xlim([0 0.15])
ylim([1 3])
title('Time domain plot of stimulus and measured signal')
legend('Excitation Signal','Measured Signal')
```




```
figure(3)
spectrogram(MeasuredSignal,1024,1000,1024,Fs,'yaxis')
title('Frequency domain view of the the measured signal')
```



Calculate the Transfer Function of the Circuit

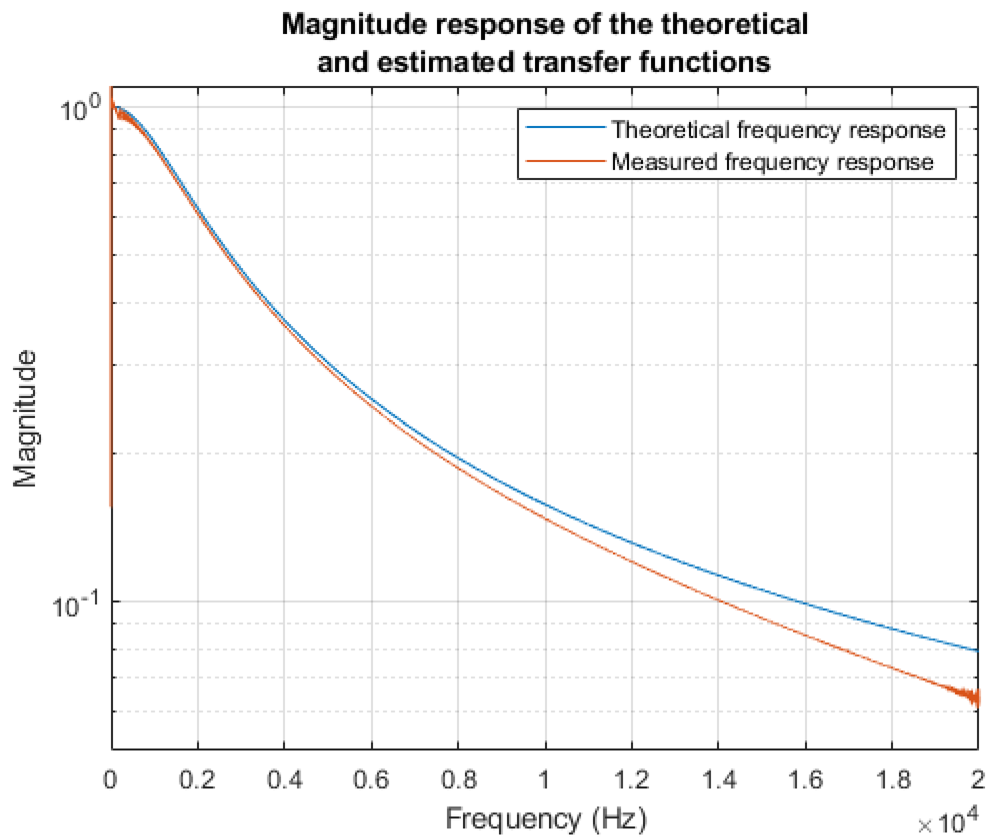
Compare the measured signal and the stimulus signal to calculate the transfer function of the R-C circuit, and plot the magnitude response.

Remove DC offset before processing.

```
MeasuredSignal = MeasuredSignal - mean(MeasuredSignal);
ExcitationSignal = ExcitationSignal - Offset;
[TFxy,Freq] = tfestimate(ExcitationSignal,MeasuredSignal,[],[],[],Fs);
Mag = abs(TFxy);
```

Compare the estimated transfer function to the theoretical magnitude response.

```
R = 1000;    % Resistance (ohms)
C = 0.1e-6;  % Capacitance (farads)
TFMagTheory = abs(1./(1 + (1i*2*pi*Freq*C*R)));

figure(4);
semilogy(Freq,TFMagTheory,Freq,Mag);
xlim([0 20e3])
xlabel('Frequency (Hz)')
ylim([0.05 1.1])
ylabel('Magnitude')
grid on
legend('Theoretical frequency response','Measured frequency response')
title({'Magnitude response of the theoretical'; 'and estimated transfer functions'});
```



Clear the DataAcquisition and Figures

```
clear ADIDaq
close all
```

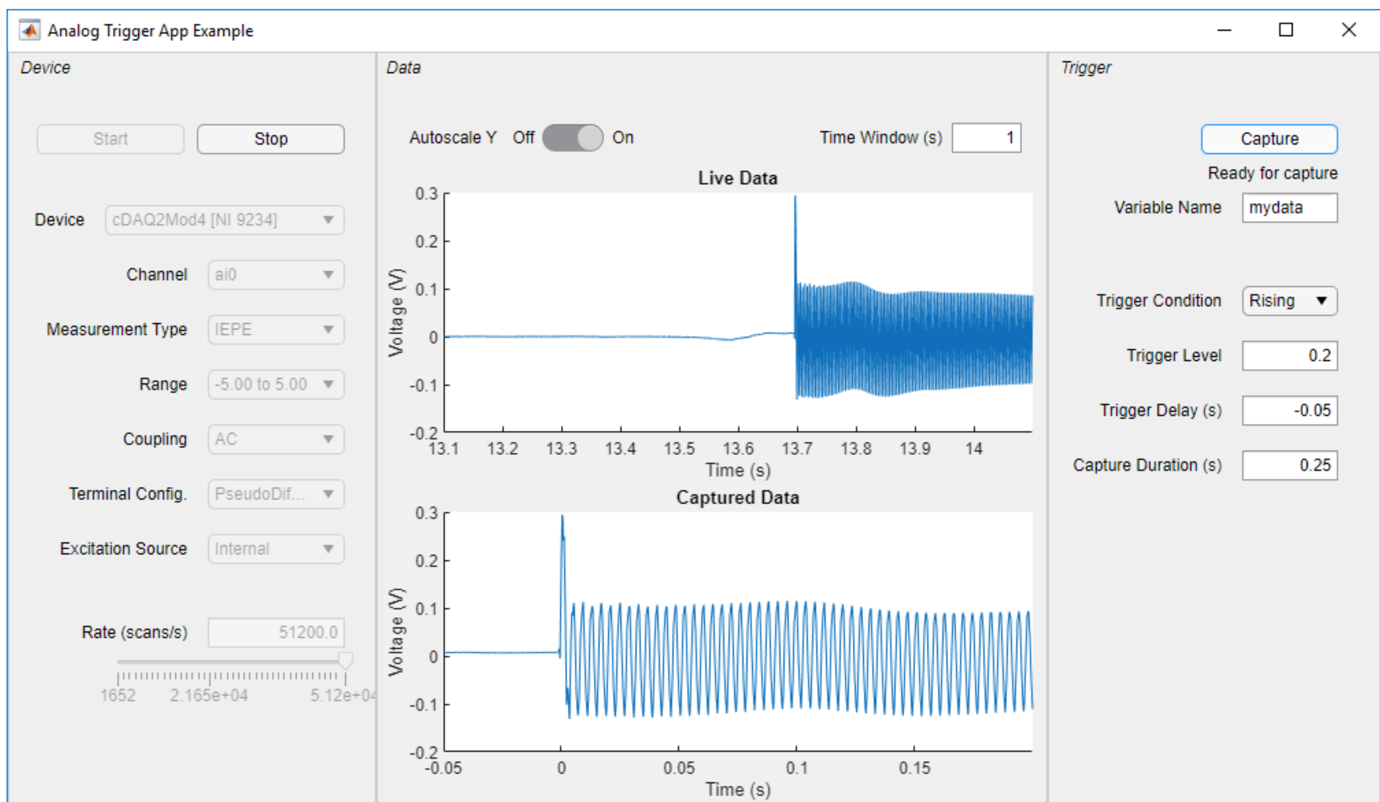
Create an App for Analog Triggered Data Acquisition

This example shows how to create an analog-triggered data acquisition app by using Data Acquisition Toolbox™ and App Designer.

Data Acquisition Toolbox provides functionality for acquiring measurement data from a DAQ device or audio sound card. For certain applications, an analog-triggered acquisition that starts capturing or logging data based on a condition in the analog signal being measured is recommended. Software-analog triggered acquisition enables you to capture only a segment of interest out of a continuous stream of measurement data. For example, you can capture an audio recording when the signal level passes a certain threshold.

This example app shows how to implement these operations:

- Discover available DAQ devices and select which device to use.
- Configure device acquisition parameters.
- Display a live plot in the app UI during acquisition.
- Perform a triggered data capture based on a programmable trigger condition.
- Save captured data to a MATLAB® base workspace variable.
- Control the operating modes of the app by defining app states in code.



By default, the app will open in design mode in App Designer. To run the app click the Run button or execute the app from the command line:

```
AnalogTriggerApp
```

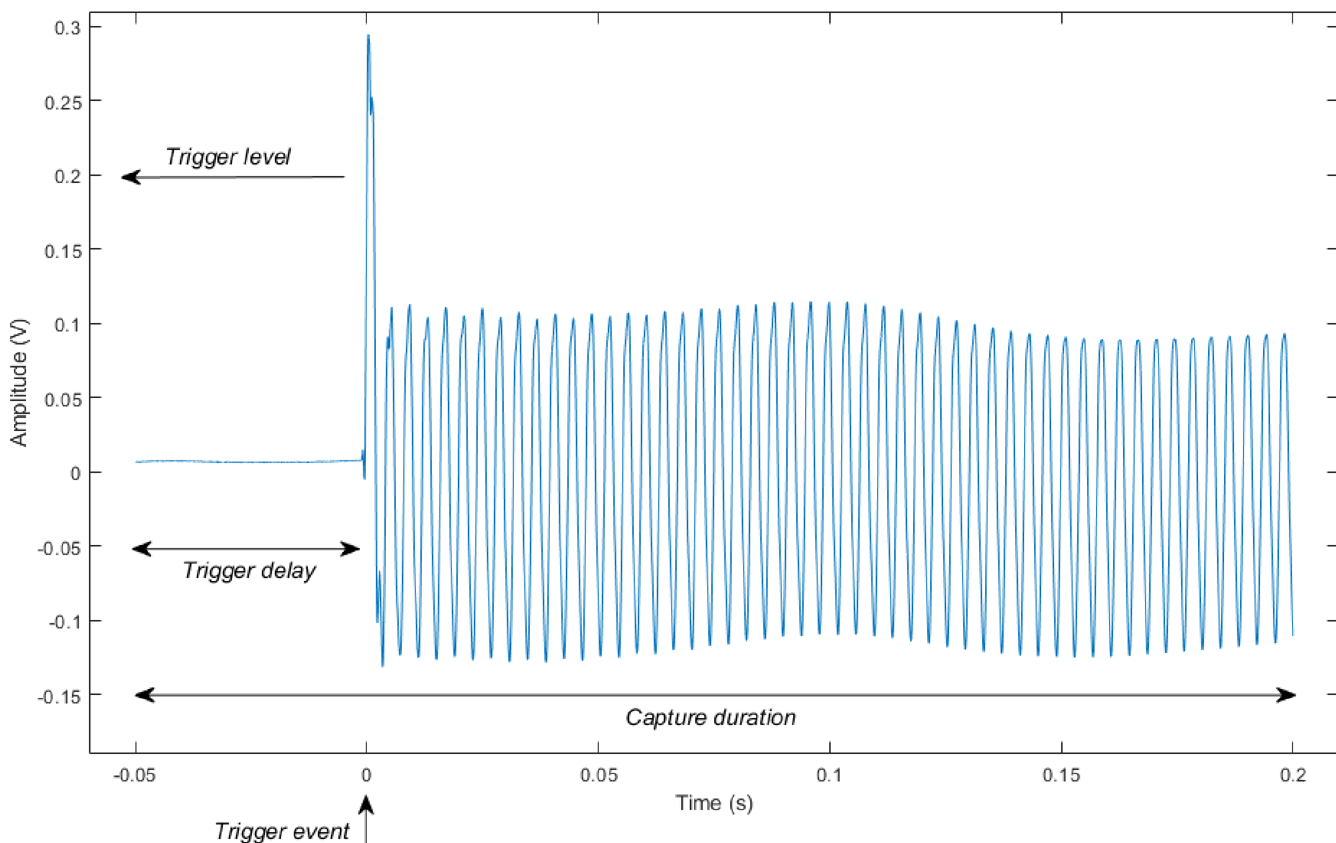
Requirements

This example app requires:

- MATLAB R2020a or later.
- Data Acquisition Toolbox.
- A supported DAQ device or sound card. For example, any National Instruments or Measurement Computing device that supports analog input `Voltage` or `IEPE` measurements and background acquisition.
- Corresponding hardware support package and device drivers.

Analog Trigger Condition

The analog trigger capture is specified by the trigger level, trigger condition, trigger delay, and capture duration which are defined as in the figure below. A negative trigger delay means pre-trigger data will be captured.



Controlling the App Operation

When creating an app that has complex logic, consider the various states that correspond to the operating modes of the app. For this app, the app logic is implemented in MATLAB code and the following app states are used:

- DeviceSelection
- Configuration
- Acquisition (Buffering, ReadyForCapture, Capture, LookingForTrigger, CapturingData, CaptureComplete)

You can use a Stateflow chart to visualize, organize, and control the app states as illustrated in the "Analog Trigger App by Using Stateflow Charts" example.

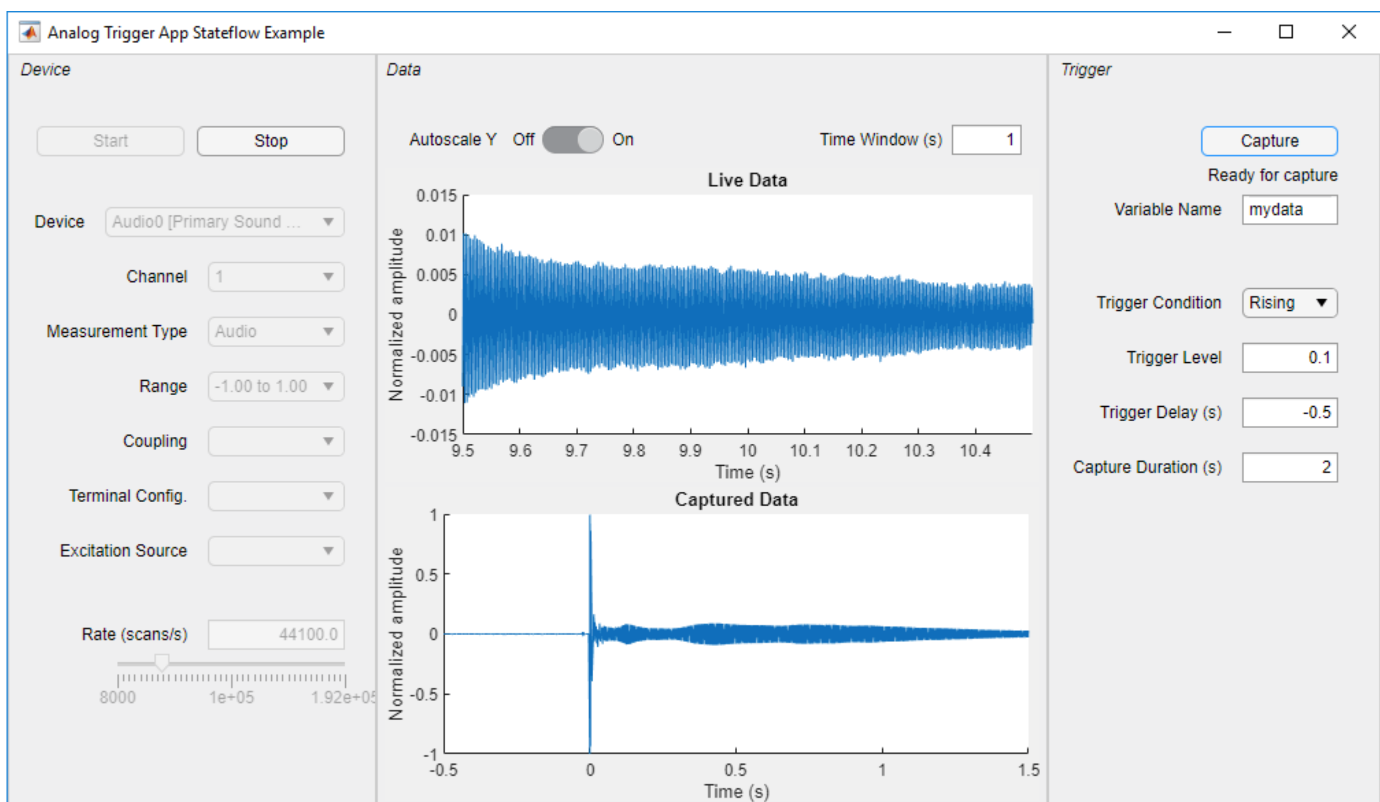
Analog Triggered Data Acquisition Using Stateflow Charts

This example shows how to create an analog-triggered data acquisition app by using Stateflow®, Data Acquisition Toolbox™, and App Designer.

Data Acquisition Toolbox provides functionality for acquiring measurement data from a DAQ device or audio soundcard. For certain applications, an analog-triggered acquisition that starts capturing or logging data based on a condition in the analog signal being measured is recommended. Software-analog triggered acquisition enables you to capture only a segment of interest out of a continuous stream of measurement data. For example, you can capture an audio recording when the signal level passes a certain threshold.

This example app, created by using App Designer and Stateflow, shows how to implement these operations:

- Control the app state logic by using a Stateflow chart.
- Discover available DAQ devices and select which device to use.
- Configure device acquisition parameters.
- Display a live plot in the app UI during acquisition.
- Perform a triggered data capture based on a programmable trigger condition.
- Save captured data to a MATLAB® base workspace variable.



By default, the app opens in design mode in App Designer. To run the app click the **Run** button or execute the app from the command line:

`AnalogTriggerAppStateflow`

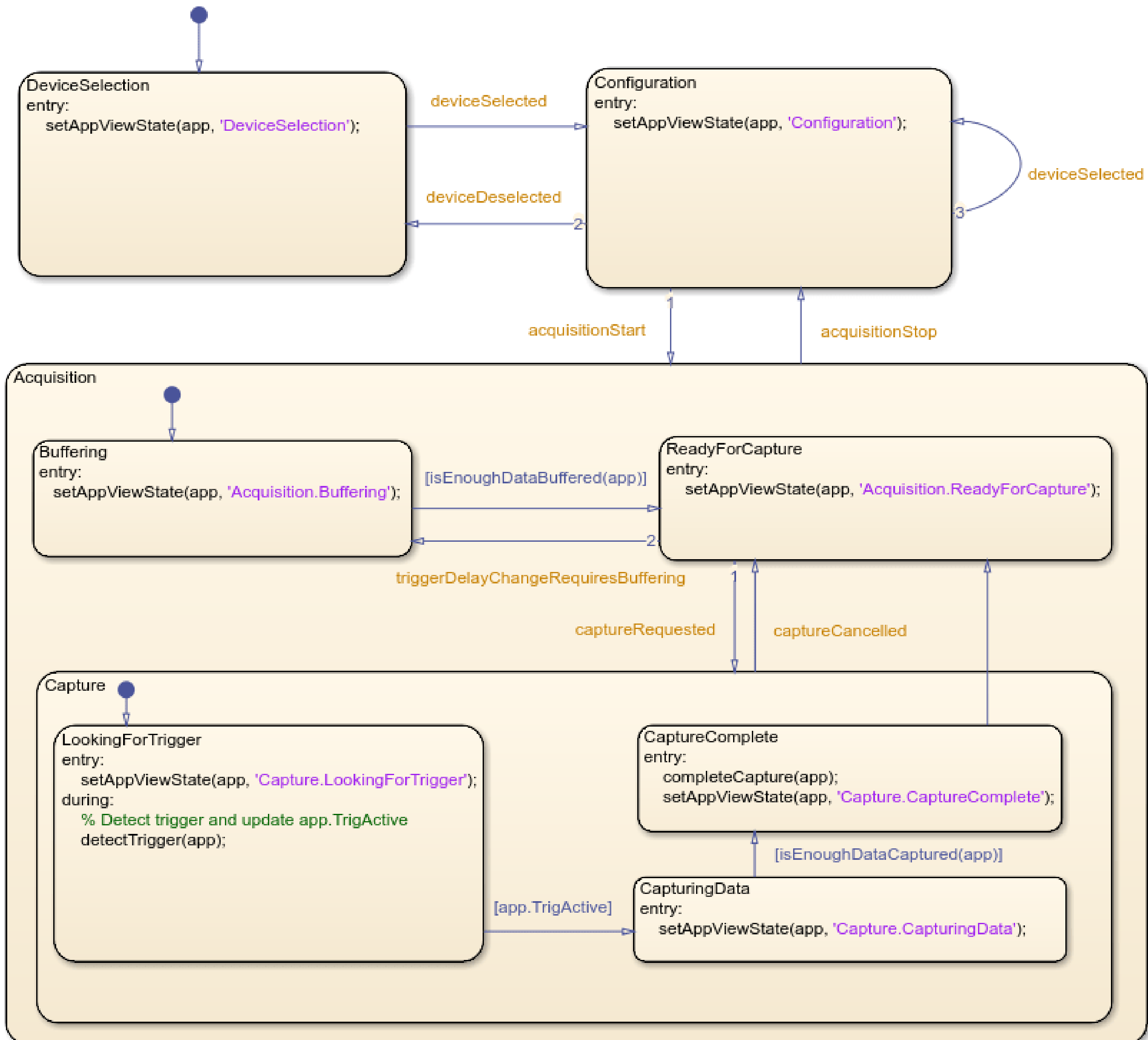
Requirements

This example app requires:

- MATLAB R2020a or later.
- Data Acquisition Toolbox (supported on Windows® only).
- Stateflow (for creating and editing charts only).
- A supported DAQ device or sound card. For example, any National Instruments or Measurement Computing device that supports analog input `Voltage` or `IEPE` measurements and background acquisition.
- Corresponding hardware support package and device drivers.

App States and the Stateflow Chart

When creating an app that has complex logic, consider the various states that correspond to the operating modes of the app. You can use a Stateflow chart to visualize and organize these app states. Use transitions between states to implement the control logic of your app. For example, the file `AnalogTriggerAppLogic.sfx` defines the Stateflow chart that controls the logic for this app. The chart can transition between states based on an action in the app UI or on a data-driven condition. For example, if you click the **Start** button, the chart transitions from the `Configuration` state to the `Acquisition` state. If the value of the signal crosses the specified trigger level, the chart transitions from the `LookingForTrigger` state to the `CapturingData` state.



Integrating the App with the Stateflow Chart

To establish a bidirectional connection between the MATLAB app and the Stateflow chart, in the `startupFcn` function of your app, create a chart object and store its handle in an app property.

```
app.Chart = AnalogTriggerAppLogic(app=app);
```

The app uses this handle to trigger state transitions in the chart. For example, when you click **Start**, the `StartButtonPushed` app callback function calls the `acquisitionStart` input event for the chart. This event triggers the transition from the `Configuration` state to the `Acquisition` state.

To evaluate transition conditions that are not events in the chart, the app calls the `step` function for the chart object. For example, while acquiring data from the device, the `dataAvailable_Callback`

app function periodically calls the `step` function. When the trigger condition is detected, the chart transitions from the `LookingForTrigger` State to the `CapturingData` state.

In the Stateflow chart, store the app object handle as chart local data. To share public properties and call public functions of the app, the Stateflow chart can use this handle in state actions, transition conditions, or transition actions.

Create an App for Live Data Acquisition

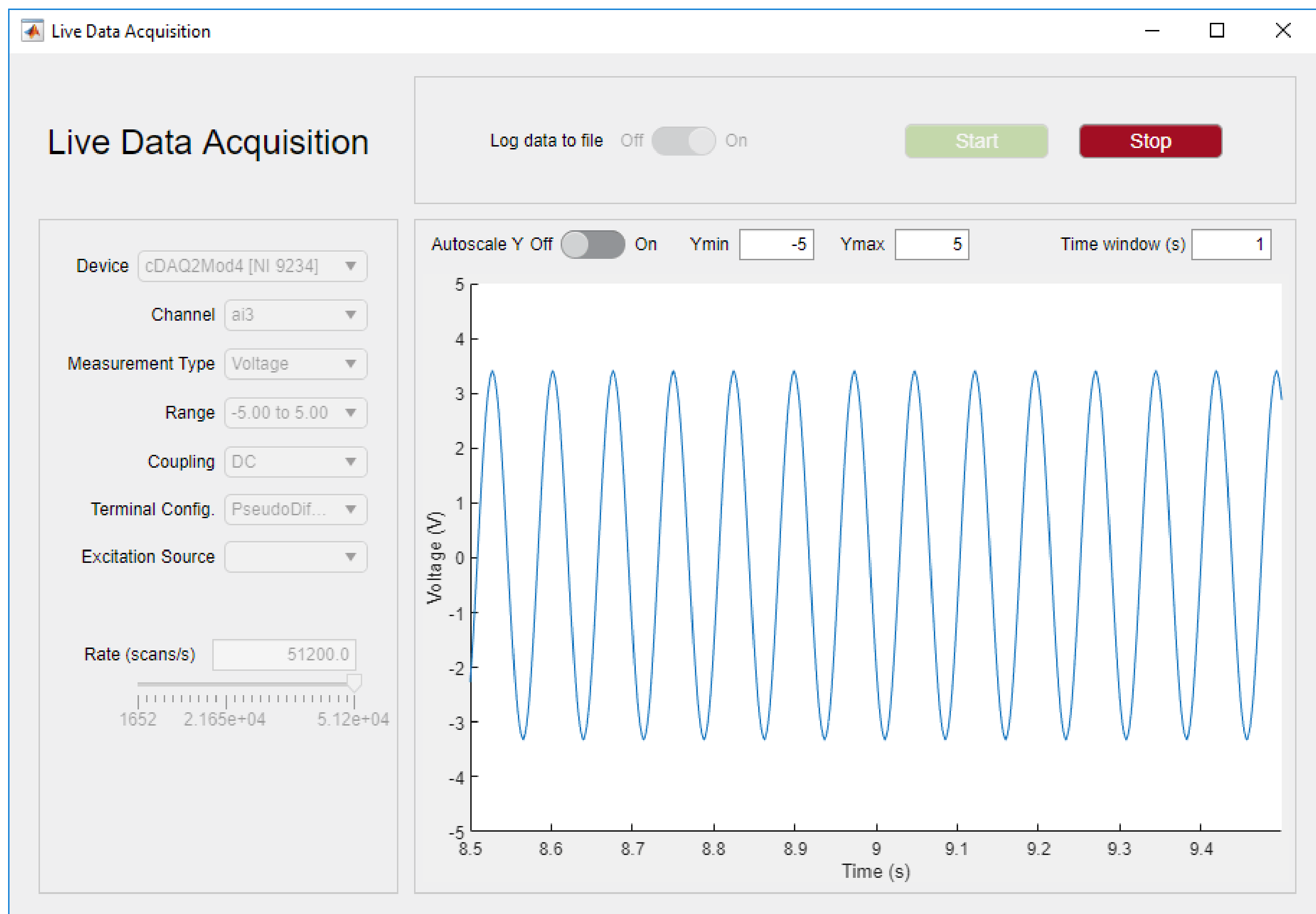
This example shows how to create an app which acquires data from a DAQ device or sound card, displays a live data view, and logs data to a MAT-file.

This example app shows how to implement these operations:

- Discover available DAQ devices and select which device to use.
- Configure device acquisition parameters.
- Display a live plot in the app UI during acquisition.
- Save acquired data to a MAT-file by writing to an intermediate binary file during acquisition.

By default, the app will open in design mode in App Designer. To run the app click **Run** or execute the app from the command line:

```
LiveDataAcquisition
```



Requirements

This example app requires:

- MATLAB® R2020a or later.
- Data Acquisition Toolbox™.
- Corresponding hardware support package for your device vendor.
- A supported DAQ device or sound card. For example, any National Instruments or Measurement Computing device that supports analog input `Voltage` or `IEPE` measurements and background acquisition.

Acquire Data Using NI FieldDAQ Device

This example shows how to acquire data from an NI FieldDAQ device.

Discover Analog Input Devices

To discover a device that supports input measurements, access the device in the table returned by the `daqlist` command. This example uses a NI FD-11603 device. This device has two banks, each with 4 channels. Channel 0 of Bank 1 is connected to a frequency generator that produces a 1 kHz sine wave (1 Vpp centered around 0.5V).

```
d = daqlist("ni")
```

```
d=10×4 table
           DeviceID                          Description                     Model           DeviceI
    __________________________    ___________________________________    ___________    _____________

    "Dev1"                        "National Instruments(TM) USB-6351"    "USB-6351"     [1×1 daq.ni.D
    "FD11603-1D3BB09-Bank1"       "National Instruments(TM) FD-11603"    "FD-11603"     [1×1 daq.ni.D
    "FD11603-1D3BB09-Bank2"       "National Instruments(TM) FD-11603"    "FD-11603"     [1×1 daq.ni.D
    "FieldDAQ1-Bank1"             "National Instruments(TM) FD-11603"    "FD-11603"     [1×1 daq.ni.D
    "FieldDAQ1-Bank2"             "National Instruments(TM) FD-11603"    "FD-11603"     [1×1 daq.ni.D
    "FieldDAQ2-Bank1"             "National Instruments(TM) FD-11613"    "FD-11613"     [1×1 daq.ni.D
    "FieldDAQ3-Bank1"             "National Instruments(TM) FD-11634"    "FD-11634"     [1×1 daq.ni.D
    "FieldDAQ3-Bank2"             "National Instruments(TM) FD-11634"    "FD-11634"     [1×1 daq.ni.D
    "FieldDAQ4-Bank1"             "National Instruments(TM) FD-11637"    "FD-11637"     [1×1 daq.ni.D
    "FieldDAQ4-Bank2"             "National Instruments(TM) FD-11637"    "FD-11637"     [1×1 daq.ni.D
```

Create a DataAcquisition and Add Analog Input Channels

Create a DataAcquisition, set the `Rate` property (the default is 1000 scans per second), and add analog input channels using `addinput`.

```
dq = daq("ni");
dq.Rate = 20000;
addinput(dq,"FD11603-1D3BB09-Bank1","ai0","Voltage");
```

```
Warning: Added channel does not support on-demand operations: only clocked operations are allowed
```

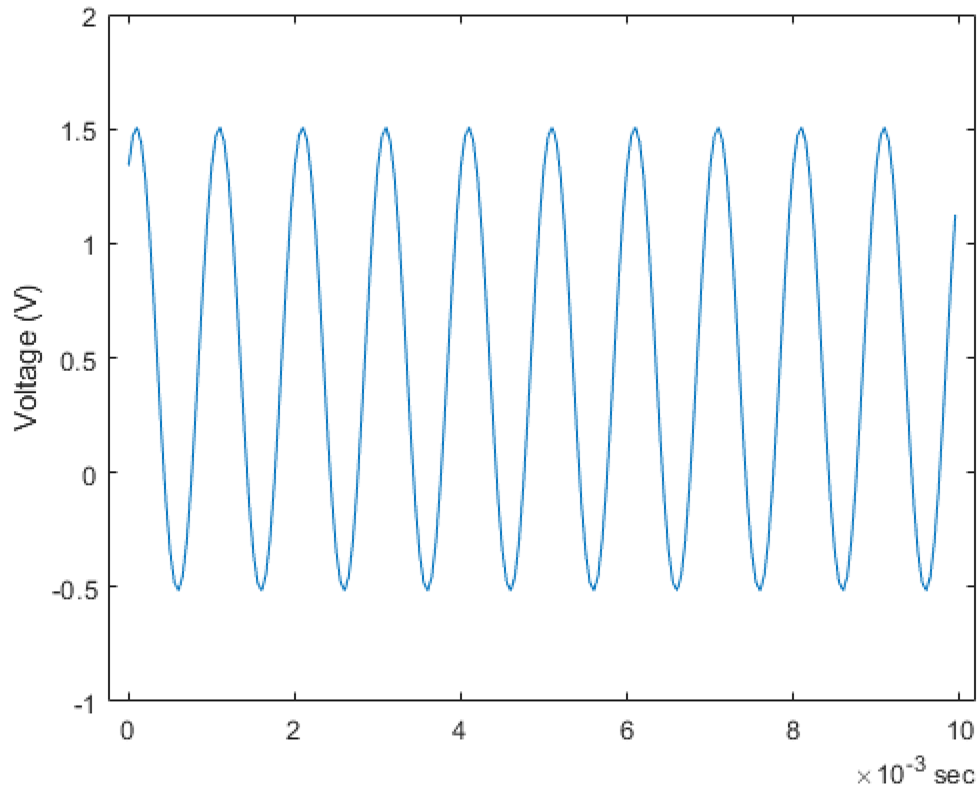
Acquire Data For a Specified Duration

Use `read` to acquire multiple scans, blocking MATLAB execution until all the data requested is acquired. The acquired data is returned as a timetable with width equal to the number of channels and height equal to the number of scans.

```
% Acquire data for one second at 20000 scans per second.
data = read(dq, seconds(1));
```

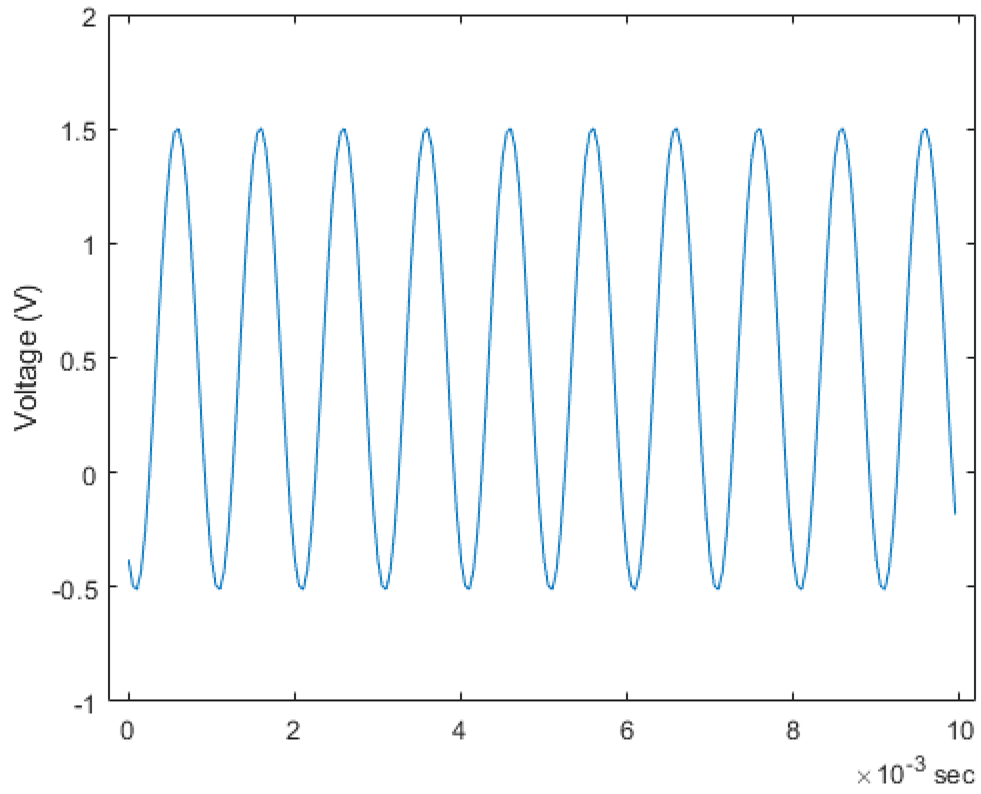
Plot the Acquired Data

```
t = data.Time;
v = data.Variables;
n = 200;
plot(t(1:n), v(1:n));
ylabel("Voltage (V)")
```



Acquire Specified Number of Scans

```
data = read(dq, 200);
plot(data.Time, data.Variables);
ylabel("Voltage (V)")
```


Voltage (V)
2
1.5
1
0.5
0
-0.5
-1
0
2
4
6
8
10
×10^-3 sec

Create an Echometer Using Audio Measurements

This example shows how to create an echometer sonar using audio data acquisition and signal processing, which can measure distance by determining the time of flight of a sound pulse reflected off of a surface.

This example employs an approach for post synchronization of audio output and input data timestamps, which is required for applications where the input signal is in response to the output and/or when output/input timing correlation is relevant. Example applications include an acoustic characterization setup or stimulus-response experiments. The relative output/input lag is determined and corrected using correlation functions in Signal Processing Toolbox.

Requirements

- MATLAB R2020a or later
- Data Acquisition Toolbox
- Data Acquisition Toolbox Support Package for Windows Sound Cards
- Signal Processing Toolbox

Hardware Setup

Running this example requires:

- Focusrite Scarlett 2i2 audio interface device, or another device / sound card with two output and two input audio channels
- Audio interface device DirectSound drivers provided by the vendor, or using default Windows device drivers if available
- One powered speaker and one microphone compatible with the audio interface device
- Audio patch cables and connector adaptors

Typical DirectSound audio interface devices supported by Data Acquisition Toolbox do not support hardware synchronization between the output and input channels. Pairs of audio input or output channels are synchronized by the audio device, however the output and input channels can have a non-negligible relative start lag.

To synchronize the output and input data timestamps in post-processing, the following setup can be used:

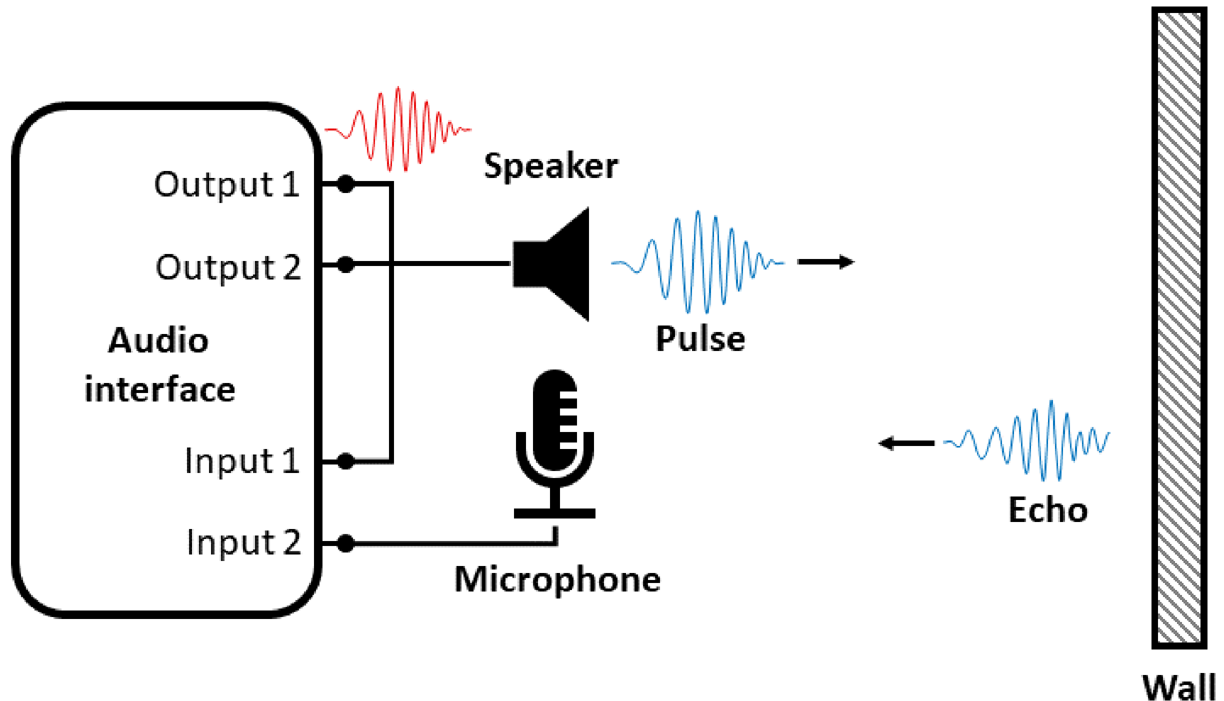
- Connect one of the output channels (output 1 or left channel of stereo plug) to one of the input channels (input 1 or left channel) to generate and acquire a synchronization signal in a loopback configuration.
- The other output (output 2 or right channel) and input channel (input 2 or right channel) are used to output an excitation/stimulus signal, and respectively acquire a measurement/response signal.
- Read data from both audio input channels, with one of the channels being used for reading the synchronization signal, and the other channel for reading the actual response signal.
- With this setup, you read data from both audio input channels, and simultaneously write data to both audio output channels.

Echometer Sonar

As a demonstration of this approach, you can use an audio device, a powered speaker, and a microphone to put together an echometer sonar setup. A pulse echometer measures the distance to

an object by emitting a short sound pulse, measuring the reflected pulse echo, and determining the time of flight by comparing the original output pulse signal with the measured input response signal.

The speaker and microphone are placed next to each other and oriented toward a wall off of which the sound pulse reflects, as in the diagram below.



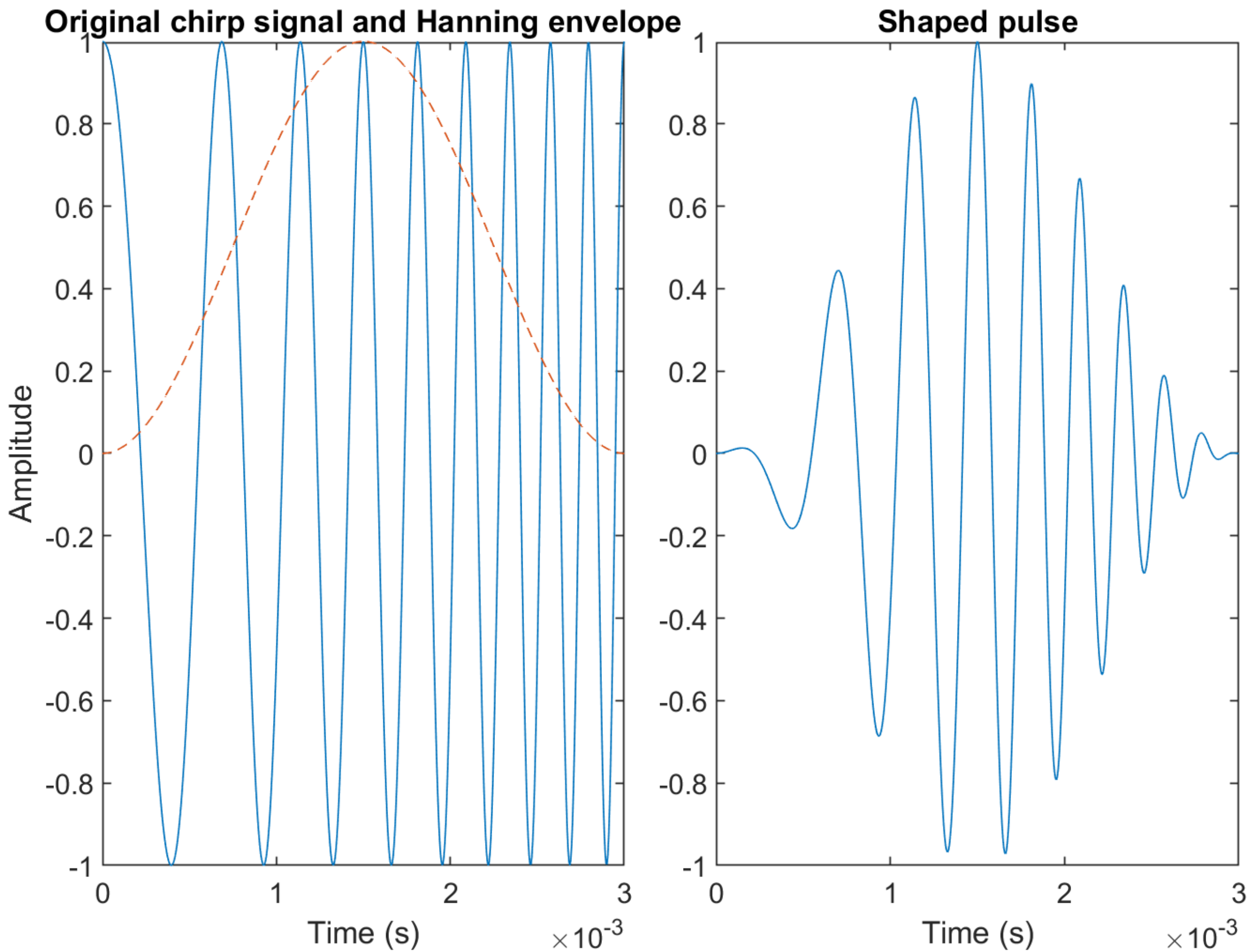
Synthesize a Pulse Signal

A pulse signal typically used for sonar applications is a short duration frequency sweep, or a chirp. Because the sharp amplitude edges at the beginning and end of a flat chirp signal can cause measurement artifacts, the pulse is attenuated/shaped by an envelope function. Options include Hanning, Gaussian, Kaiser, etc. Frequency range, pulse width or duration, pulse envelope/shape depend on the intended application. In this example the measurements are taken with a 3 ms duration pulse with a 1-5 kHz linear frequency chirp, and a Hanning window. The synthesized signal is shown in the figure below. The original chirp signal is shown on the left and the shaped pulse is shown on the right. The Hanning window is shown as a dotted line.

Make sure to use a signal frequency range that can be properly generated by the speaker, picked up by the microphone, and sampled by the audio interface device. The sampling rate used for the audio device measurements is 192 kHz.

```
% Pulse width (s)
T = 3E-3;

% Sampling rate (Hz)
Fs = 192E+3;

% Initial and final pulse frequency (Hz)
f0 = 1E+3;
```

```
f1 = 5E+3;

% Time vector
t = (0:1/Fs:T)';

% Pulse signal, chirp attenuated by an windowing function
yc = chirp(t,f0,t(end),f1);
w = hanning(numel(t));
y = yc.*w;

% Plot the signal
figure
tileplot = tiledlayout(1,2);
tileplot.TileSpacing = 'compact';
tileplot.Padding = 'compact';

nexttile
plot(t,yc)
hold on
plot(t,w,'--')
xlabel("Time (s)")
ylabel("Amplitude")
title("Original chirp signal and Hanning envelope")

nexttile
plot(t,y)
xlabel("Time (s)")
title("Shaped pulse")
```



Data Acquisition

Use two separate `DataAcquisition` objects, one for the audio output channels and one for the audio input channels. Since there is no automatic synchronization possible between the audio input and output channel pairs even if a common `DataAcquisition` object is used for all channels, this approach allows for more control over the data acquisition operations.

```
do = daq("directsound");
addoutput(do,"Audio4","1","Audio");
addoutput(do,"Audio4","2","Audio");
do.Rate = Fs;

di = daq("directsound");
addinput(di,"Audio1","1","Audio");
addinput(di,"Audio1","2","Audio");
di.Rate = Fs;
```

Since the pulse duration is relatively short, pad the ending of the pulse signal with zero values (200 ms duration) to ensure that the pulse is generated correctly.

```
yout = [y; zeros(Fs*200E-3,1)];
```

Start the input first as a continuous background acquisition, then generate the same signal on both audio output channels. One of the channels is used as a synchronization signal.

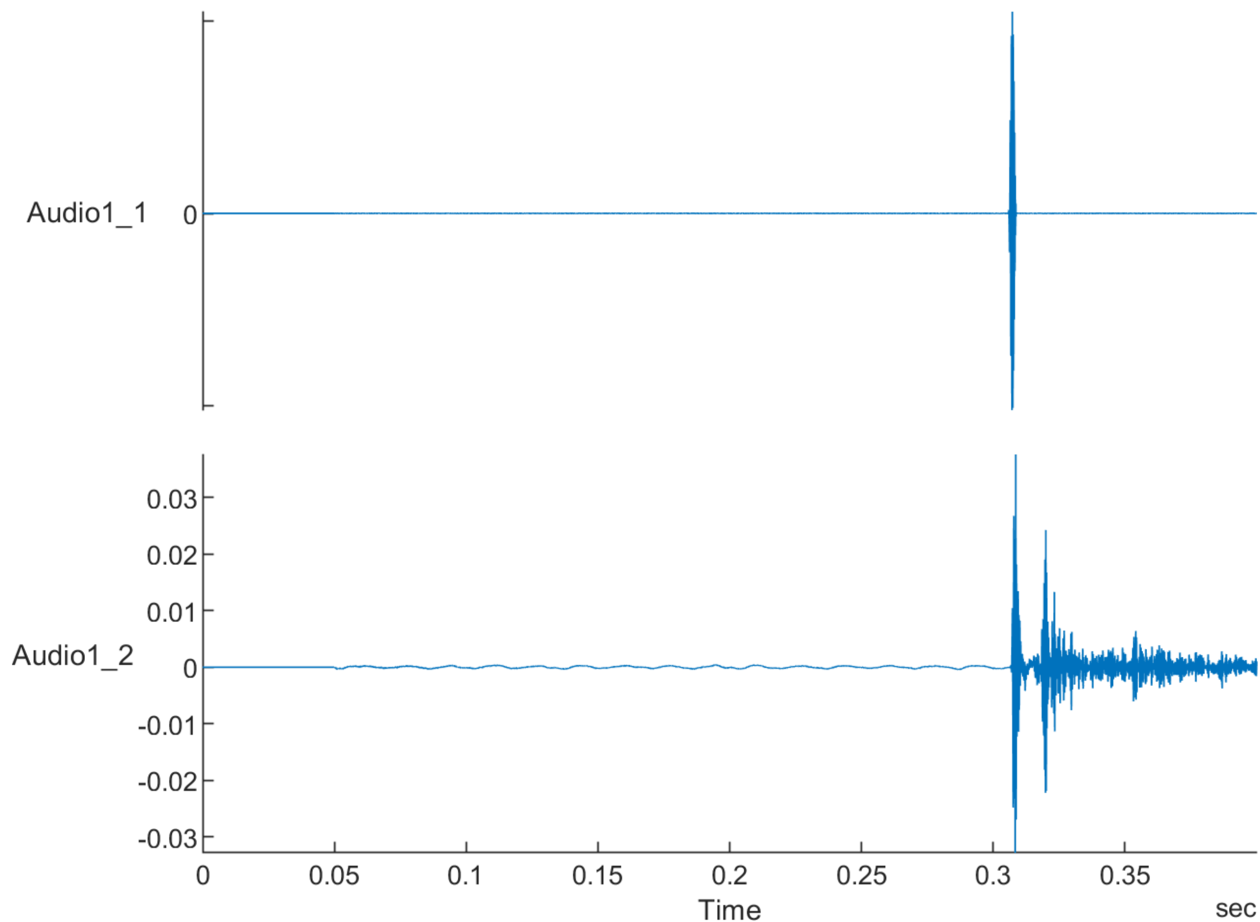
```
start(di,"continuous")
write(do,[yout yout])
stop(di)
```

Read the acquired data into the workspace. By default, the `read` function returns a `timetable`.

```
data = read(di,"all");
```

Plot the acquired data. Signals Audio1_1 and Audio1_2 correspond to the audio input channels 1 and 2. Input channel 1 was used to record the synchronization signal generated by output channel 1 in a loopback configuration. Input channel 2 was used to record the actual response signal picked by the microphone. Two pulses are observed in the response signal, followed by other secondary echoes which depend on the room acoustics. Also notice the large relative start lag time between the input and output channels.

```
figure
stackedplot(data)
```



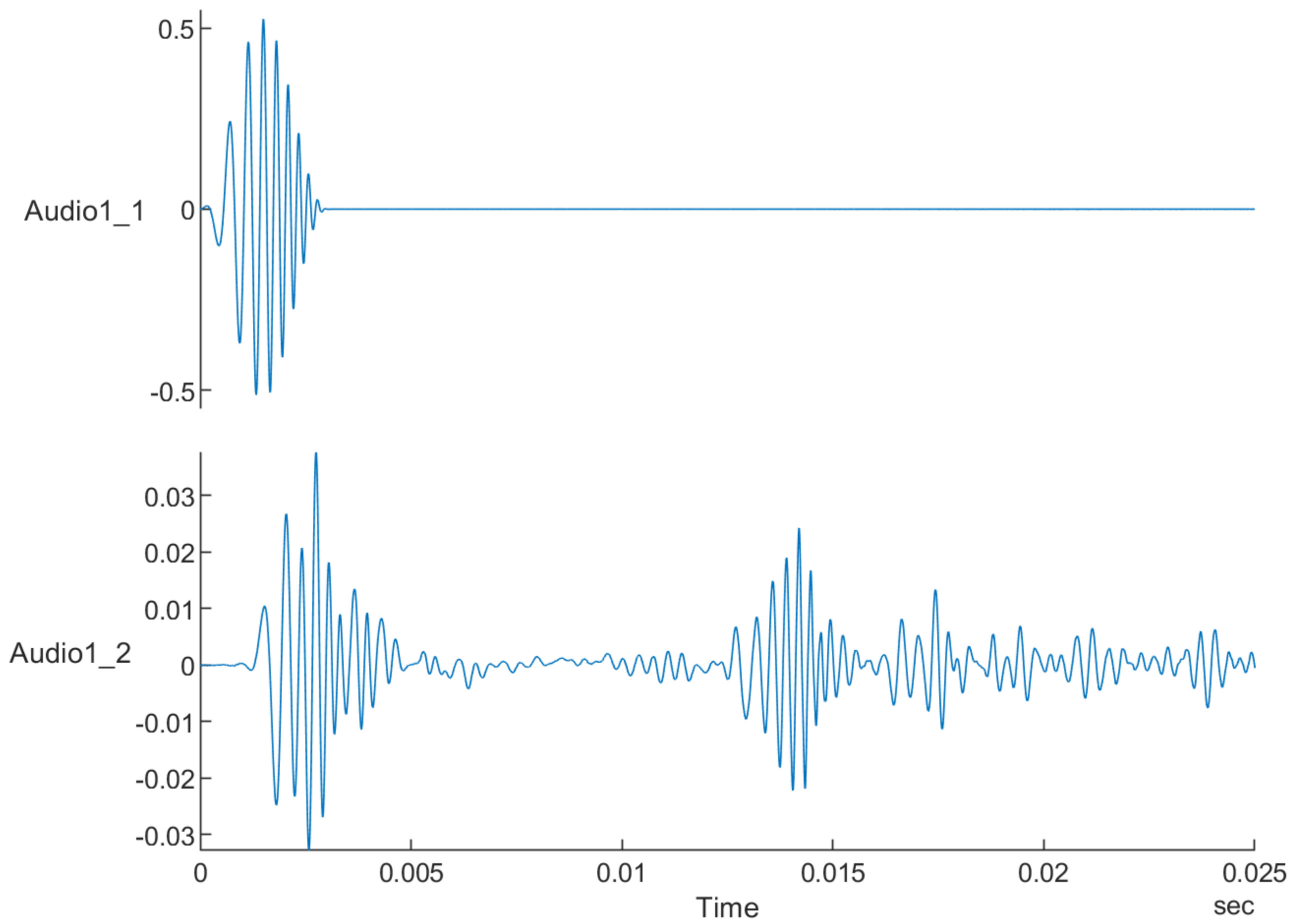
Synchronize the Output and Input Data Timestamps

Find the lag and align the output and input signal timestamps by discarding the points before the detected synchronization signal.

```
lag = finddelay(y, data.Audio1_1);
t0 = lag/Fs
```

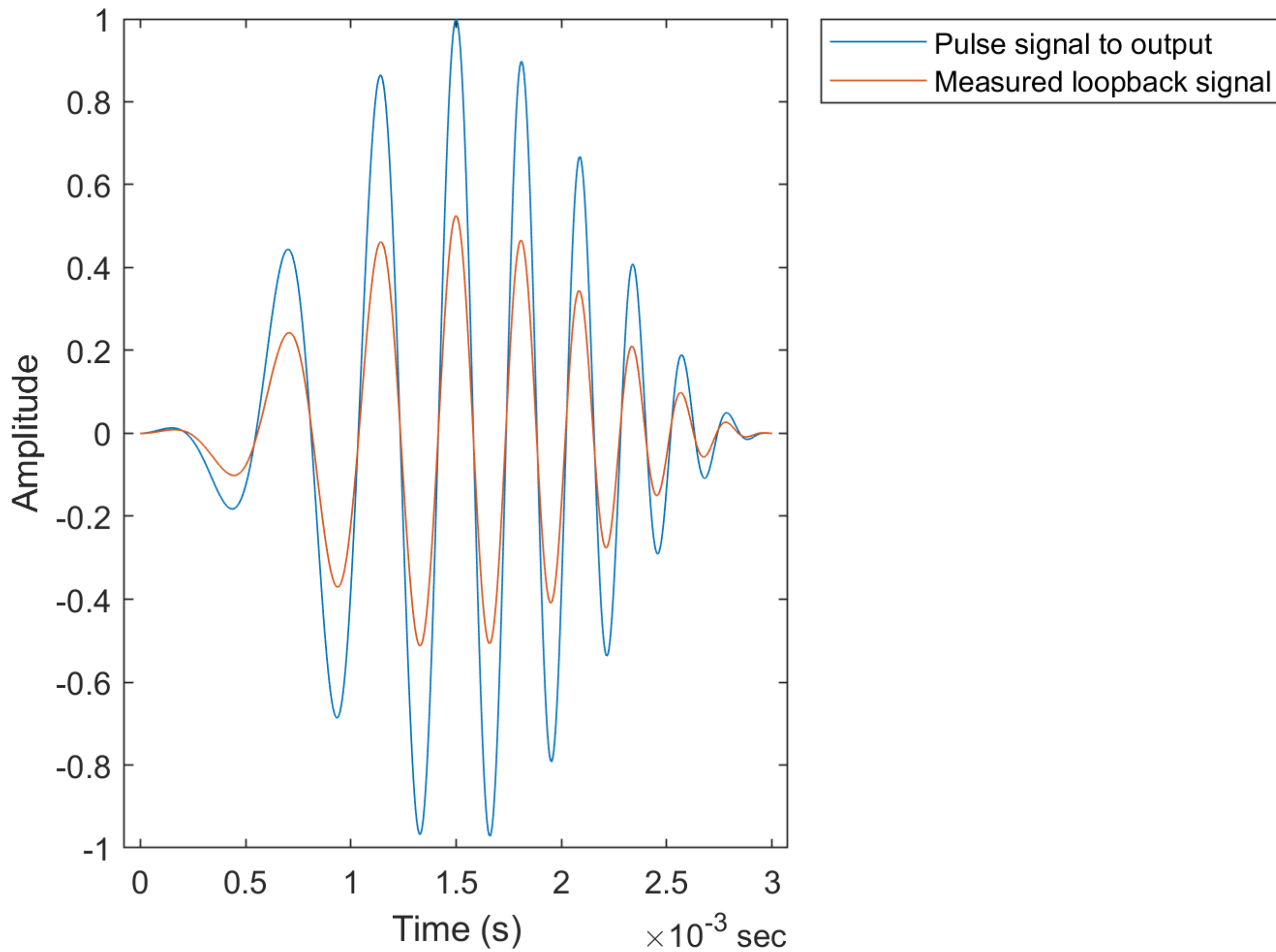
```
t0 = 0.3058
```

```
alignedData = data(lag+1:end,:);
alignedData.Time = alignedData.Time-alignedData.Time(1);

figure
s = stackedplot(alignedData);
xlim(seconds([0 0.025]))
s.AxesProperties(1).YLimits = [-0.55 0.55];
```



Visually validate the quality of the time alignment, by comparing the synthetic pulse data with the measured loopback signal.

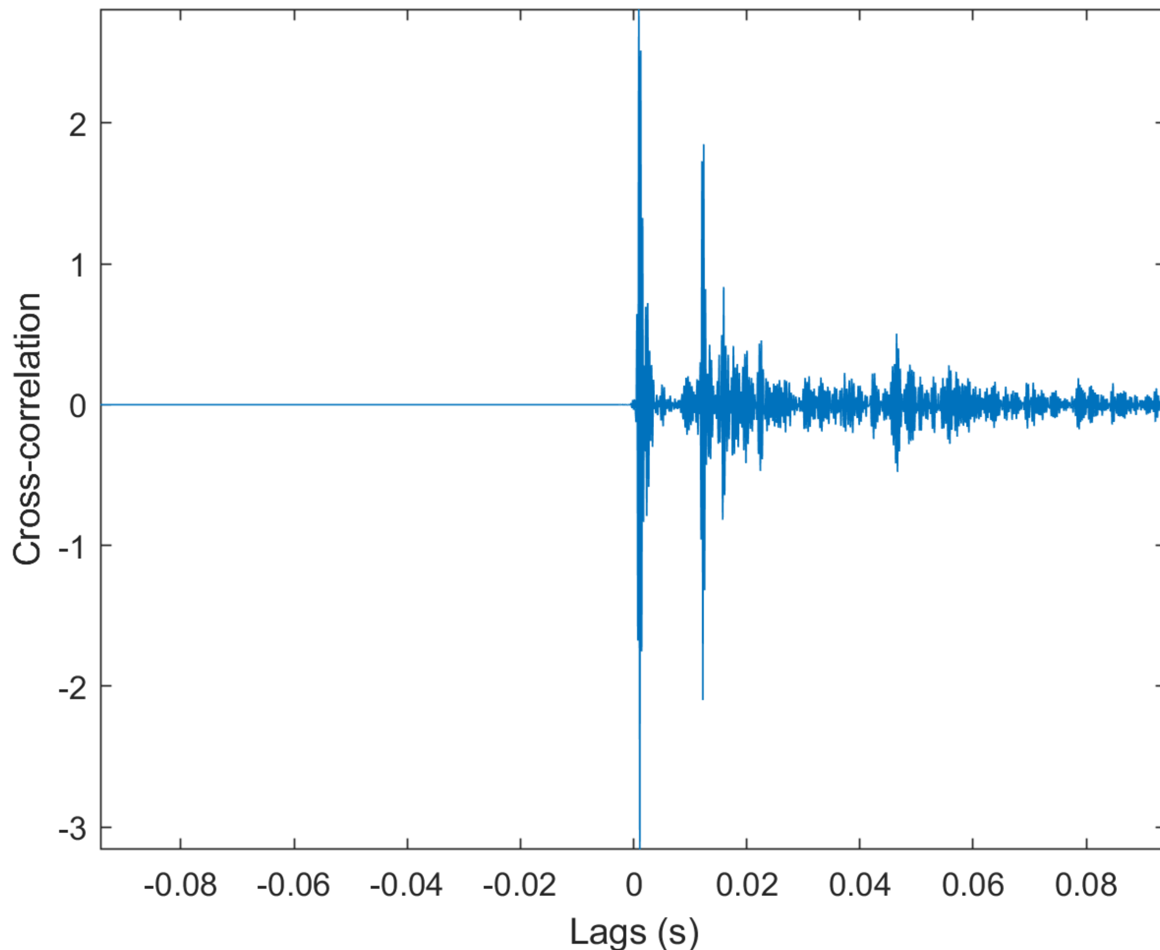
```
figure
plot(seconds(t),y,alignedData.Time(1:numel(t)),alignedData.Audio1_1(1:numel(t)))
ylabel("Amplitude")
xlabel("Time (s)")
legend(["Pulse signal to output","Measured loopback signal"],"Location","bestoutside")
```



Determine Pulse Propagation Time and Distance

You can use the `xcorr` cross-correlation function to determine and visualize similarities between the original pulse signal and the measured response signal.

```
[xCorr,lags] = xcorr(alignedData.Audio1_2,y);

figure
plot(lags/Fs,xCorr)
xlabel('Lags (s)')
ylabel('Cross-correlation')
axis tight
```



The cross-correlation plot indicates several similarities, with two larger peaks and other smaller peaks from reverberations.

Find timestamp and total propagation distance corresponding to the first two observed correlated pulses in the measured signal. The first observed pulse corresponds to the direct propagation path from the speaker to the microphone. The second observed pulse is the echo pulse reflected by the wall. The function `finddelay` returns the lag for which the normalized cross-correlation has the highest value, and in this case it corresponds to the first pulse in the response signal.

```
t1 = finddelay(y,alignedData.Audio1_2)/Fs
```

```
t1 = 0.0011
```

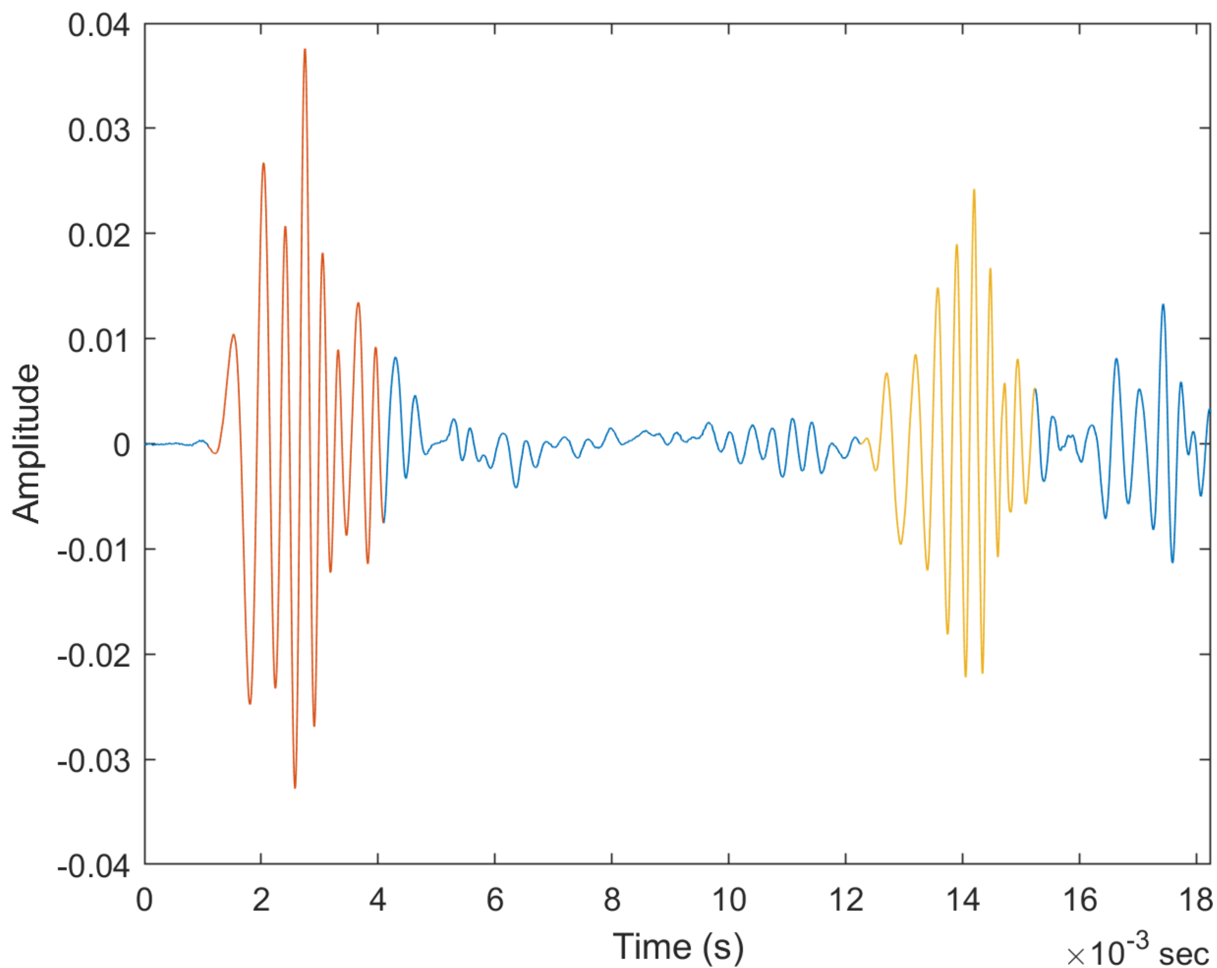
You can calculate the propagation time of the echo pulse as the starting timestamp of the second pulse in the response signal by using `finddelay` in the signal region after the first pulse.

```
t2 = t1 + T + finddelay(y,alignedData(timerange(seconds(t1+T),"inf"),:).Audio1_2)/Fs
```

```
t2 = 0.0122
```

```
% Plot the response signal and highlight the first two detected pulses
figure
```

```
plot(alignedData.Time,alignedData.Audio1_2)
xlim(seconds([0 t2+2*T]))

hold on
firstPulse = alignedData(timerange(seconds(t1),seconds(t1+T)),:);
plot(firstPulse.Time,firstPulse.Audio1_2)

echoPulse = alignedData(timerange(seconds(t2),seconds(t2+T)),:);
plot(echoPulse.Time,echoPulse.Audio1_2)
ylabel("Amplitude")
xlabel("Time (s)")
```




```
% Calculate distance corresponding to echo pulse (m)
% Speed of sound in air at 20 deg. C (m/s)
v = 343.1;
d2 = t2*v/2
```

```
d2 = 2.1006
```

The distance (2.10 m) measured by the echometer sonar setup (half of the total path) is very close to the actual distance (2.06 m) between the speaker/microphone setup and the wall reflecting the echo pulse.

Get Started Reading a TDMS-File

This example shows how to read data from a TDMS-file into MATLAB® for analysis.

The example TDMS-file contains measurement data of a sine wave amplitude and phase. The measurements are in two channels, in the same channel group.

Inspect the TDMS-File Contents

Use the `tdmsinfo` function to obtain channel group and channel names in the TDMS-file.

```
fileName = "SineWave.tdms";
info = tdmsinfo(fileName);
info.ChannelList
```

```
ans=2×8 table
    ChannelGroupNumber    ChannelGroupName    ChannelGroupDescription       ChannelName       Cha
    __________________    ________________    _______________________    _________________    ___

            1             "Measured Data"               ""               "Amplitude sweep"
            1             "Measured Data"               ""               "Phase sweep"
```

Read Data Properties from the TDMS-File

Use the `tdmsreadprop` function to view data properties from the file.

```
tdmsreadprop(fileName)
```

```
ans=1×7 table
       name       description              datetime               author              title
    __________    ___________    _____________________________    ______    ____________________

    "SineWave"        ""         2022-01-12 23:33:31.000000000    "Admin"    "Amplitude And Phase
```

Specify a `ChannelGroupName` and `ChannelName` arguments to view properties of a specific channel.

```
group = "Measured Data";
channel = "Amplitude sweep";
tdmsreadprop(fileName, ChannelGroupName=group, ChannelName=channel)
```

```
ans=1×19 table
           name           description    unit_string     datatype      displaytype       monoton
    _________________    ___________    ___________    ___________    ___________    ___________

    "Amplitude sweep"        ""             ""         "DT_DOUBLE"     "Numeric"     "not calcul
```

Read Timetable Data from the TDMS-File into MATLAB

To read data into a timetable, derive the start time and time step, typically contained in the channel properties.

```
timeStep = tdmsreadprop(fileName, ChannelGroupName=group, ChannelName=channel, PropertyNames="wf_
```

```
timeStep=table
    wf_increment
```

```
    ____________

       0.001
```

```
startTime = tdmsreadprop(fileName, ChannelGroupName=group, ChannelName=channel, PropertyNames="wf
```

```
startTime=table
            wf_start_time
    _____________________________

    1903-12-31 19:00:00.000000000
```

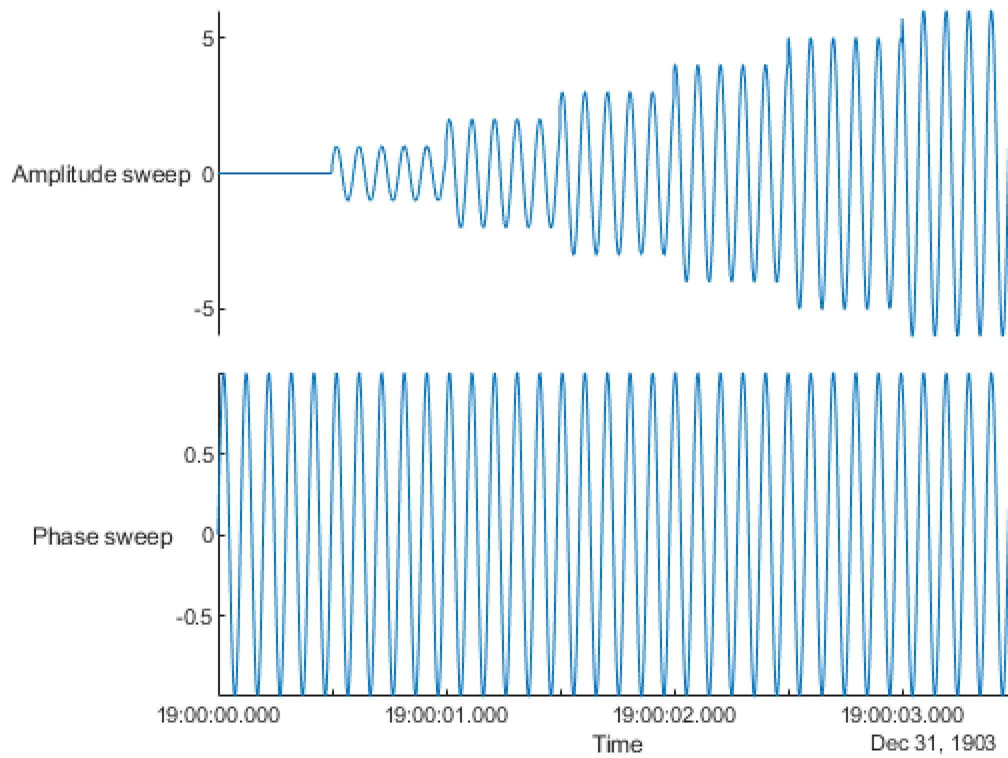
Using the start time and time step as arguments to the `tdmsread` function, read the data into MATLAB as a cell array of timetables. View some of the data from first channel group.

```
data = tdmsread(fileName, StartTime=startTime.wf_start_time, TimeStep=seconds(timeStep.wf_increme
ttData = data{1};
head(ttData)
```

```
ans=8×2 timetable
                Time                Amplitude sweep    Phase sweep
    _____________________________    _______________    ___________

    1903-12-31 19:00:00.000000000          0                    0
    1903-12-31 19:00:00.001000000          0             0.063418
    1903-12-31 19:00:00.002000000          0              0.12658
    1903-12-31 19:00:00.003000000          0              0.18923
    1903-12-31 19:00:00.004000000          0              0.25112
    1903-12-31 19:00:00.005000000          0                0.312
    1903-12-31 19:00:00.006000000          0              0.37163
    1903-12-31 19:00:00.007000000          0              0.42975
```

Use a stacked plot to visualize the relationship between the data of different channels.

```
stackedplot(ttData);
```



Amplitude sweep
5
0
-5
Phase sweep
0.5
0
-0.5
19:00:00.000
19:00:01.000
19:00:02.000
19:00:03.000
Time
Dec 31, 1903

Read Multiple TDMS-Files into MATLAB

This example shows how to use a TDMS datastore to read data from multiple TDMS-files into MATLAB® for analysis.

The multiple TDMS-files of this example contain measurement data of a sine wave amplitude and phase, based on a trigger.

Read Multiple TDMS-Files at Once

The `tdmsDatastore` function creates a datastore object, which allows you to treat all the TDMS-files in a folder as one dataset.

```
folderName = fullfile(pwd,"Trigger");
ds = tdmsDatastore(folderName);
```

Review all the channels in the dataset.

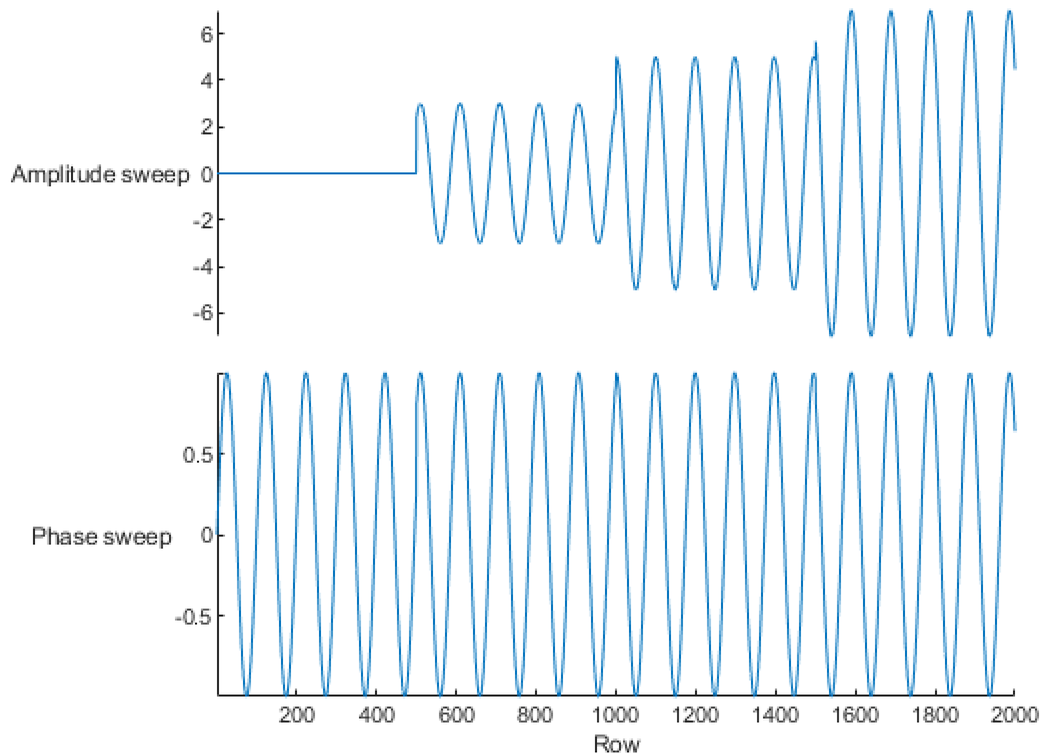
```
ds.ChannelList
```

```
ans=2×8 table
    ChannelGroupNumber    ChannelGroupName    ChannelGroupDescription        ChannelName        Cha
    __________________    ________________    _______________________    ___________________    ___

            1                "Trigger"                  ""               "Amplitude sweep"
            1                "Trigger"                  ""               "Phase sweep"
```

All the TDMS-files in the datastore must have the same channel groups and channels. The NumSamples in the channel list is that of the first file in the datastore.

The `readall` function retrieves all the data from the datastore into MATLAB. Read and plot the data from the first channel group.

```
data = readall(ds);
stackedplot(data{1})
```



Read Datastore TDMS-Files Individually

You can also analyze data in the datastore one file at a time. Redefine the TDMS datastore with a read size of `"file"`.

```
ds = tdmsDatastore(folderName, readSize="file");
```

Iterate through the datastore, reading data from each TDMS-file. The `read` function uses the set read size. For each file, find its maximum and minimum values in the `Amplitude sweep` channel, and compare it to the cumulative maximum and minimum.

```
maxAmplitude = 0;
minAmplitude = 0;
while(hasdata(ds))
    data = read(ds);
    maxAmplitude = max(maxAmplitude, max(data{1}.("Amplitude sweep")));
    minAmplitude = min(minAmplitude, min(data{1}.("Amplitude sweep")));
end
```

After the entire dataset is read, view the cumulative maximum and minimum amplitudes.

```
maxAmplitude
```

```
maxAmplitude = 6.9998
```

```
minAmplitude
```

```
minAmplitude = -6.9982
```

Read a Large TDMS-File into MATLAB

This example shows how to use a TDMS datastore to read data from a large TDMS-file into MATLAB® for analysis.

The example TDMS-file contains measurement data of a sine wave amplitude and phase. It also contains a channel group with sporadic events triggered at random times. The example file itself is not large, but is used to show the techniques for handling a large file.

You can use a TDMS datastore object to read a large TDMS-file into MATLAB by iteratively reading available chunks of data. The size of the data in each iteration might depend on a time interval of interest, an absolute number of samples, or other group definition.

Define the Datastore and View Its Channel Information

Given the name of the TDMS-file, use the `tdmsDatastore` function to create a TDMS datastore object for a specified read size and channel group.

```
fileName = "SineWaveWithEvents.tdms";
readSize = 1000;
ds = tdmsDatastore(fileName, ReadSize=readSize, SelectedChannelGroup="Measured Data");
```

Examine the data store channel list to identify the channel group and channel names of interest.

```
ds.ChannelList
```

```
ans=4×8 table
    ChannelGroupNumber    ChannelGroupName    ChannelGroupDescription      ChannelName       Cha
    __________________    ________________    _______________________    _________________   ___

            1             "Measured Data"               ""               "Amplitude sweep"
            1             "Measured Data"               ""               "Phase sweep"
            2             "Events"                      ""               "Time"
            2             "Events"                      ""               "Description"
```

Read Data into MATLAB

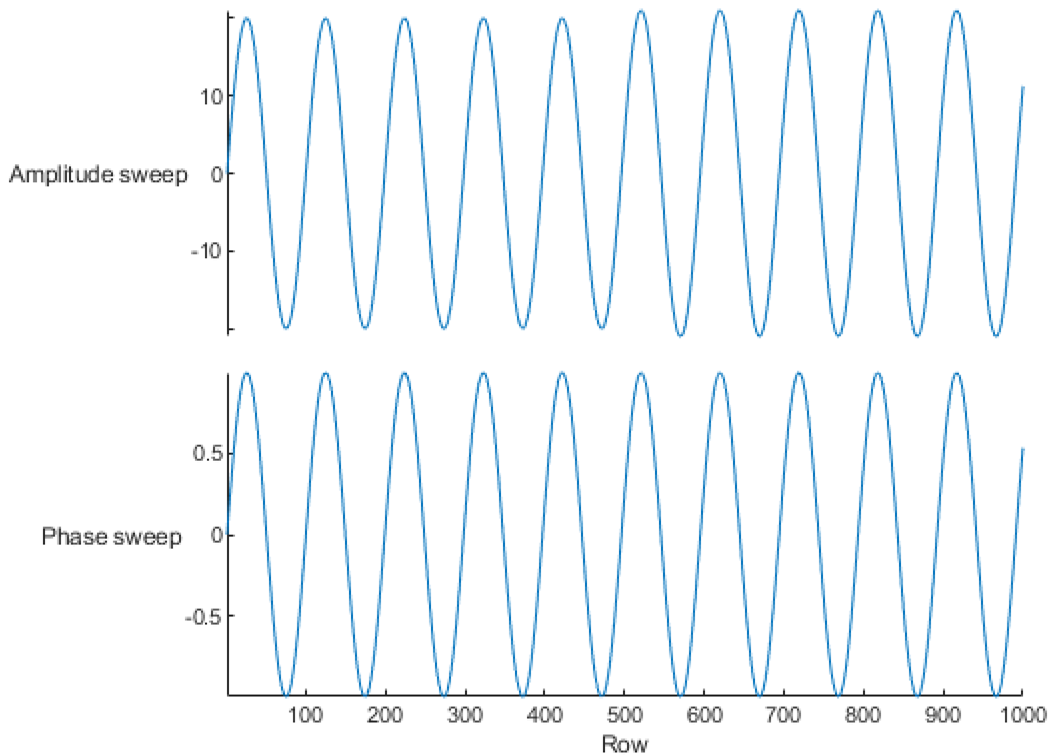
Loop through the file to read 1000 samples at a time, until the end of the file. The datastore read size and selected channel group were defined when the datastore `ds` was created above with the `tdmsDatastore` function. The `read` function is constrained by these parameters. Use the `hasdata` function if you do not know the number of iterations required.

```
while(hasdata(ds))
    data = read(ds);
        % This loop overwrites the last iteration data,
        % so any analysis of individual data or cumulative
        % reduction operations must happen inside the loop.
end
```

Visualize the Most Recent Chunk of Data

You can visualize the last 1000 samples of data using a stacked plot on the first channel group.

```
stackedplot(data{1});
```



Examine Event Data

Read and display the channel group containing the event data. This requires redefining the datastore object with `tdmsDatastore` function for that channel group. Assuming it is not a large set of data, you can use the `readall` function to retrieve it all at once.

```
ds = tdmsDatastore(fileName, SelectedChannelGroup="Events");
data = readall(ds);
data{1}
```

```
ans=5×2 table
                Time                   Description
    _____________________________    _____________

    2022-01-12 22:21:36.058876037    "Failure 123"
    2022-01-12 22:21:38.558219909    "Failure 123"
    2022-01-12 22:21:40.058685302    "Failure 123"
    2022-01-12 22:21:41.558692932    "Event 4711"
    2022-01-12 22:21:44.559547424    "Failure 123"
```

Get Started Writing a TDMS-File

This example shows how to write data from MATLAB® to a TDMS-file.

For this example, the file `weather.mat` contains a regional weather report from 3 December 1998 to 30 Nov 2000.

Set Up the Workspace

Load the data to be written to a TDMS-file, and define the TDMS-file, channel group, and channel names. Later you can add some custom attributes, such as title and units, to the TDMS-file.

```
load("weather.mat");
fileName = "weather.tdms";
group = "ChannelGroup1";
channel = "T_min";
whos
```

```
  Name             Size              Bytes  Class     Attributes

  channel          1x1                 150  string              
  fileName         1x1                 166  string              
  group            1x1                 166  string              
  weather        729x19             116009  table               
```

The variable `weather` is a table that holds the data.

Write the Table of Data to a New TDMS-File

Use the `tdmswrite` function to write the table of weather data to a TDMS-file from MATLAB.

```
tdmswrite(fileName, weather)
```

Inspect the contents of the file using `tdmsinfo`.

```
info = tdmsinfo(fileName)
```

```
info = 
  TdmsInfo with properties:

           Path: "C:\Users\rkoshy\OneDrive - MathWorks\Documents\MATLAB\ExampleManager\rkoshy.Exa
           Name: "weather.tdms"
    Description: ""
          Title: "Weather Report"
         Author: ""
        Version: "2.0"
    ChannelList: [38×8 table]
```

View channel groups and channels in the TDMS-file.

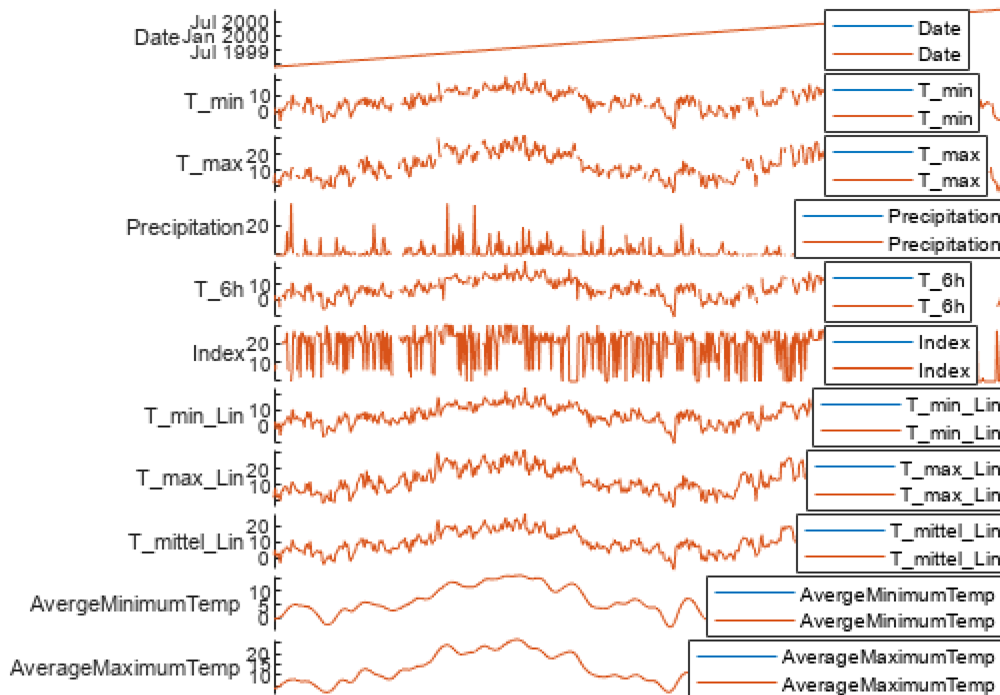
```
info.ChannelList
```

```
ans=38×8 table
    ChannelGroupNumber    ChannelGroupName    ChannelGroupDescription          ChannelName
    __________________    ________________    _______________________    _______________________
```

```
1       "ChannelGroup1"      ""      "Date"
1       "ChannelGroup1"      ""      "T_min"
1       "ChannelGroup1"      ""      "T_max"
1       "ChannelGroup1"      ""      "Precipitation"
1       "ChannelGroup1"      ""      "T_6h"
1       "ChannelGroup1"      ""      "Index"
1       "ChannelGroup1"      ""      "T_min_Lin"
1       "ChannelGroup1"      ""      "T_max_Lin"
1       "ChannelGroup1"      ""      "T_mittel_Lin"
1       "ChannelGroup1"      ""      "AvergeMinimumTemp"
1       "ChannelGroup1"      ""      "AverageMaximumTemp"
1       "ChannelGroup1"      ""      "AverageTemp"
1       "ChannelGroup1"      ""      "RealTemperatureDiff"
1       "ChannelGroup1"      ""      "AverageTemperatureDiff"
1       "ChannelGroup1"      ""      "Month"
1       "ChannelGroup1"      ""      "Tm_min"
    :
```

Use `tdmsread` to examine the data written into the new file.

```
rData = tdmsread(fileName);
stackedplot(rData)
```



Inspect file default properties using `tdmsreadprop`.

```
tdmsreadprop(fileName)
```

ans=*1×5 table*

name	description	title	author	timestamp
"weather.tdms"	""	"Weather Report"	""	2022-04-21 19:25:30.357063999

Inspect channel group default properties using `tdmsreadprop`.

```
tdmsreadprop(fileName, ChannelGroupName=group)
```

ans=*1×2 table*

name	description
"ChannelGroup1"	""

Inspect channel default properties using `tdmsreadprop`.

```
tdmsreadprop(fileName, ChannelGroupName=group, ChannelName=channel)
```

ans=*1×3 table*

name	description	unit_string
"T_min"	""	"°C"

Modify the TDMS-File Metadata

To update the file properties, channel group properties, or channel properties of an existing TDMS file, use `tdmswriteprop`.

You can inspect the updated properties using `tdmsreadprop`.

Update the file property `Title`.

```
tdmswriteprop(fileName, "title", "Weather Report")
tdmsreadprop(fileName)
```

ans=*1×5 table*

name	description	title	author	timestamp
"weather.tdms"	""	"Weather Report"	""	2022-04-21 19:25:30.357063999

Also add a custom file property called `timestamp`, and set its value to the current date and time.

```
tdmswriteprop(fileName, "timestamp", datetime("now"))
tdmsreadprop(fileName)
```

ans=*1×5 table*

name	description	title	author	timestamp
"weather.tdms"	""	"Weather Report"	""	2022-04-21 19:34:37.310101999

Finally, update the units of a channel, which is specified by the default property `unit_string`.

```
tdmswriteprop(fileName, "unit_string", "°C", ChannelGroupName=group, ChannelName=channel)
tdmsreadprop(fileName, ChannelGroupName=group, ChannelName=channel)
```

```
ans=1×3 table
      name      description    unit_string
    _______    ___________    ___________

    "T_min"        ""             "°C"
```

Write Timetable Data to TDMS-file

This example shows how to write timetable data in various time channel layouts from MATLAB® to a TDMS-file.

For this example, you have measurents of revolution and electrical current of a circular saw in a scenario where it stops on detecting contact with skin.

Load the data to workspace.

```
load("sawstopper.mat")
whos
```

```
  Name                        Size             Bytes  Class        Attributes

  circular_saw_data      23572x2            378432  timetable
```

```
fileNameTCNone = "sawstopper_none.tdms";
fileNameTCSingle = "sawstopper_single.tdms";
channelGroup = "Circular Saw Data";
```

Write Timetable Without a Time Channel

On specifying `TimeChannel` as `"none"`, the start time and time step are added as properties of the channel. This time channel layout can be used only with a timetable that is regular in time, that is, with uniform time steps.

```
tdmswrite(fileNameTCNone, circular_saw_data, ChannelGroupName=channelGroup, TimeChannel="none");
info = tdmsinfo(fileNameTCNone);
info.ChannelList
```

```
ans=2×8 table
    ChannelGroupNumber     ChannelGroupName      ChannelGroupDescription         ChannelName
    __________________    ___________________    _______________________    _____________________

            1             "Circular Saw Data"              ""               "Revolutions (1/min)"
            1             "Circular Saw Data"              ""               "Current (A)"
```

The channel names in the TDMS-file map to the original timetable variable names.

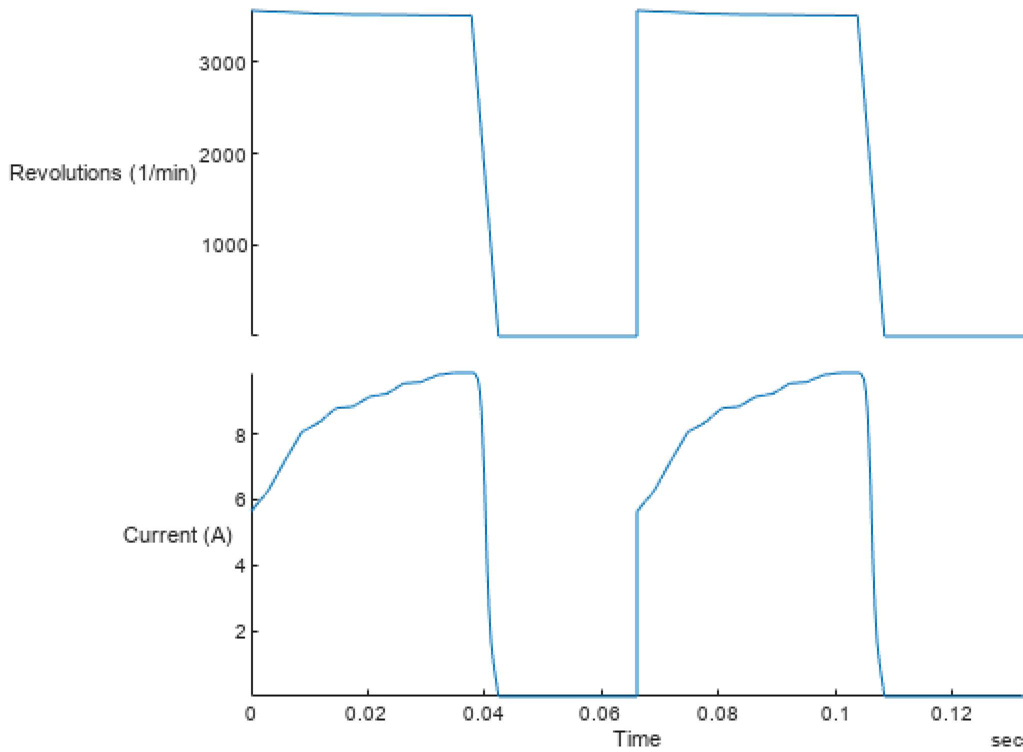
Use the `tdmsreadprop` function to inspect the start time (`wf_start_time`) and time step (`wf_increment`) of the data.

```
channel = info.ChannelList.ChannelName{1};
prop = tdmsreadprop(fileNameTCNone, ChannelGroupName=channelGroup, ChannelName=channel)
```

```
prop=1×7 table
            name            description    unit_string            wf_start_time            wf_s
    _____________________   ___________    ___________    _____________________________    ____

    "Revolutions (1/min)"       ""             ""         2022-04-19 14:18:32.304446999
```

Read the data from the TDMS-file and visually analyze the data using a stacked plot.

```
stackedplot(tdmsread(fileNameTCNone, TimeStep=seconds(prop.wf_increment)));
```



Write Timetable with a Time Channel

By default, the time channel layout is `TimeChannel="single"`, which means a time channel is created that contains a timestamp for every sample. Typically this time channel layout is useful when writing measurments that are irregular in time.

```
tdmswrite(fileNameTCSingle, circular_saw_data, ChannelGroupName=channelGroup, TimeChannel="single
```

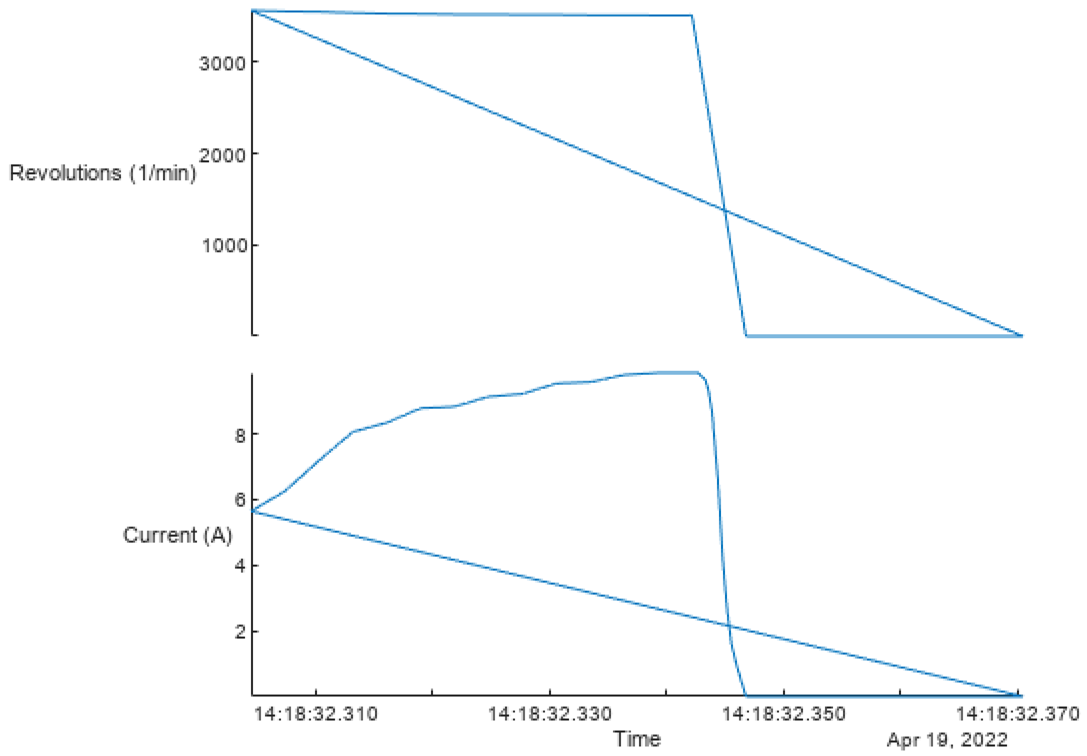
Inspect the contents of the file. See that a time channel called `"Time"` is created, which is derived from the time column of the original timetable.

```
info = tdmsinfo(fileNameTCSingle);
info.ChannelList
```

```
ans=3×8 table
    ChannelGroupNumber       ChannelGroupName       ChannelGroupDescription          ChannelName
    __________________    ____________________    _______________________    ______________________

            1             "Circular Saw Data"               ""               "Time"
            1             "Circular Saw Data"               ""               "Revolutions (1/min)"
            1             "Circular Saw Data"               ""               "Current (A)"
```

Read the data from the TDMS-file, and visually analyze the data using a stacked plot with the time channel as the x-axis.

```
stackedplot(tdmsread(fileNameTCSingle, ChannelGroupName=channelGroup, RowTimes="Time"));
```


Revolutions (1/min)
3000
2000
1000
Current (A)
8
6
4
2
14:18:32.310
14:18:32.330
14:18:32.350
14:18:32.370
Time
Apr 19, 2022

Write Metadata to TDMS-File

This example shows how to write file properties, channel group properties, and channel properties to a TDMS-file.

For this example, you have measurents of revolution and electrical current of a circular saw in a scenario where it stops on detecting contact with skin. You can write these measurement to a new TDMS-file, update the default properties, and add custom properties to the file, channel group, and channels.

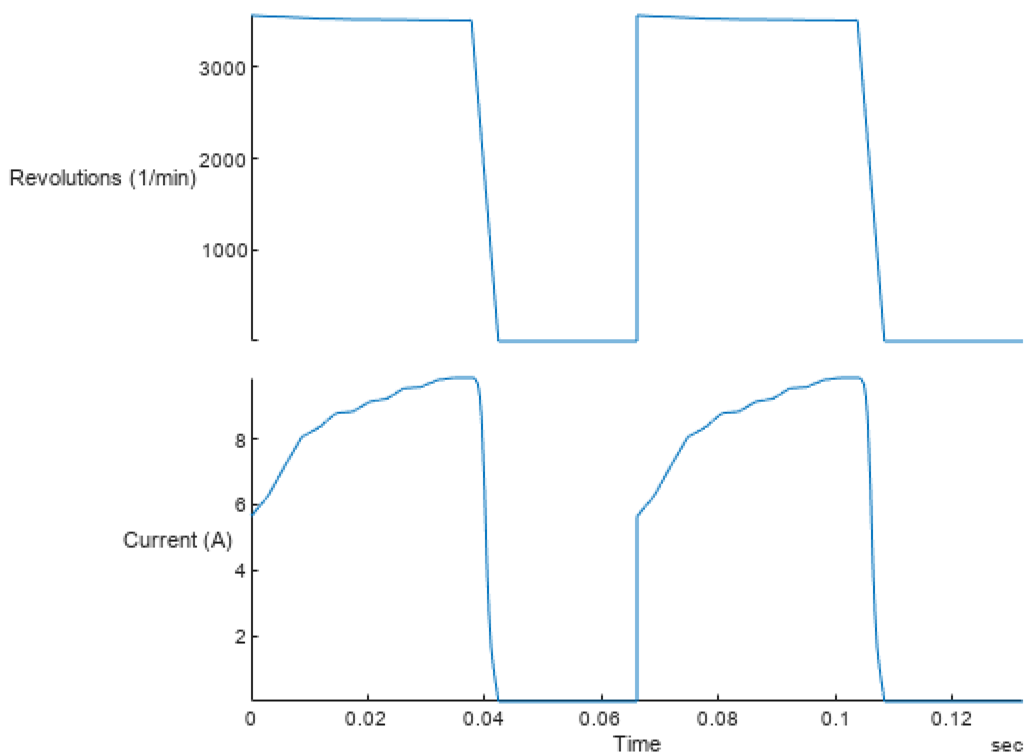
Set Up Workspace

Read the data into the workspace and define the TDMS-file channel group names.

```
load("sawstopper.mat")
fileName = "sawstopper.tdms";
channelGroup = "SawData";
channelRev = "Revolutions (1/min)";
channelCurrent = "Current (A)";
```

Create the TDMS-file, and then read and view its data.

```
tdmswrite(fileName, circular_saw_data, ChannelGroupName=channelGroup, TimeChannel="none");
stackedplot(tdmsread(fileName, TimeStep=milliseconds(0.0028)))
```



Write File Properties

Observe the default file properties.

```
tdmsreadprop(fileName)
```

ans=*1×5 table*

name	description	title	author	timestamp
"sawstopper.tdms"	""	"Saw Stopper Data"	""	2022-04-26 13:14:31.6755

Update the `title` property using `tdmswriteprop`, and add a new custom `timestamp` property set with the current datetime.

```
tdmswriteprop(fileName, "title", "Saw Stopper Data");
tdmswriteprop(fileName, "timestamp", datetime("now"));
tdmsreadprop(fileName)
```

ans=*1×5 table*

name	description	title	author	timestamp
"sawstopper.tdms"	""	"Saw Stopper Data"	""	2022-04-26 13:15:23.3508

Write Channel Group Properties

Observe the default channel group properties.

```
tdmsreadprop(fileName, ChannelGroupName=channelGroup)
```

ans=*1×3 table*

name	description	scenario
"SawData"	"Circular Saw Measurements"	"Stop on detecting contact with skin"

Update the channel group property `description`. Then add a custom property `scenario` to the channel group.

```
tdmswriteprop(fileName, "description", "Circular Saw Measurements", ChannelGroupName=channelGrou
tdmswriteprop(fileName, "scenario", "Stop on detecting contact with skin",ChannelGroupName=chann
tdmsreadprop(fileName, ChannelGroupName=channelGroup)
```

ans=*1×3 table*

name	description	scenario
"SawData"	"Circular Saw Measurements"	"Stop on detecting contact with skin"

Write Channel Properties

Similarly, you can update the channel properties such `unit_string` by specifying the channel group name and channel name.

```
tdmsreadprop(fileName, ChannelGroupName=channelGroup, ChannelName=channelRev)

ans=1×8 table
            name                          description                    unit_string          wf_
    _____________________    ______________________________________    ___________    __________

    "Revolutions (1/min)"    "Rotational speed of the circular saw"       "rpm"        2022-04-19

tdmswriteprop(fileName, "unit_string", "rpm", ChannelGroupName=channelGroup, ChannelName=channelR
tdmswriteprop(fileName, "formula", "rpm = frequency(Hz)/60", ChannelGroupName=channelGroup, Chann
tdmswriteprop(fileName, "description", "Rotational speed of the circular saw", ChannelGroupName=c
tdmsreadprop(fileName, ChannelGroupName=channelGroup, ChannelName=channelRev)

ans=1×8 table
            name                          description                    unit_string          wf_
    _____________________    ______________________________________    ___________    __________

    "Revolutions (1/min)"    "Rotational speed of the circular saw"       "rpm"        2022-04-19
```

You can also specify arrays of property names and values. If the property values have different data types, use a cell array.

```
tdmsreadprop(fileName, ChannelGroupName=channelGroup, ChannelName=channelCurrent)

ans=1×7 table
        name                 description              unit_string            wf_start_time
    _____________    _____________________________    ___________    ____________________________

    "Current (A)"    "Current in the circular saw"        "A"        2022-04-19 14:18:32.30444699

tdmswriteprop(fileName, ["description" "unit_string"], ["Current in the circular saw" "A"], ...
    ChannelGroupName=channelGroup, ChannelName=channelCurrent);
tdmsreadprop(fileName, ChannelGroupName=channelGroup, ChannelName=channelCurrent)

ans=1×7 table
        name                 description              unit_string            wf_start_time
    _____________    _____________________________    ___________    ____________________________

    "Current (A)"    "Current in the circular saw"        "A"        2022-04-19 14:18:32.30444699
```

Examine the property settings in the `ChannelList` display.

```
info = tdmsinfo(fileName);
info.ChannelList

ans=2×8 table
    ChannelGroupNumber    ChannelGroupName       ChannelGroupDescription            ChannelName
    __________________    ________________    ___________________________    ___________________

            1                "SawData"        "Circular Saw Measurements"    "Revolutions (1/min)
            1                "SawData"        "Circular Saw Measurements"    "Current (A)"
```

Merge Multiple TDMS-Files

This example shows how to merge multiple TDMS-files by appending their data into a single TDMS-file.

In this example you have vibration data of a rotating shaft recorded into a collection of TDMS-files. The goal is to read each file and merge its data into an aggregate TDMS-file for analysis. You can merge the data either by creating separate channel groups or by appending the data to a single channel group. This example shows both approaches.

```
fileMultipleChannelGroup = "RotatingShaftAnalysis_MultipleChannelGroup.tdms";
fileSingleChannelGroup = "RotatingShaftAnalysis_SingleChannelGroup.tdms";
```

Append Data by Creating New Channel Groups

Create a datastore with which you can iteratively read the collection of TDMS files in the folder `RotatingShaftAnalysis`, one file at a time.

```
ds = tdmsDatastore("RotatingShaftAnalysis", ReadSize="file");
```

You can write data to an existing TDMS-file by adding new channel groups. By default, the `tdmswrite` function creates auto incrementing channel group names.

```
while(hasdata(ds))
    data = read(ds); %read one file.
    tdmswrite(fileMultipleChannelGroup, data);
end
```

On inspecting the contents of the written file you see there are 3 channel groups that map to 3 files in the folder.

```
info = tdmsinfo(fileMultipleChannelGroup);
info.ChannelList
```

```
ans=9×8 table
    ChannelGroupNumber    ChannelGroupName    ChannelGroupDescription    ChannelName    ChannelDe
    __________________    ________________    _______________________    ___________    _________

            1             "ChannelGroup1"               ""               "Pulse"
            1             "ChannelGroup1"               ""               "Sensor_X"
            1             "ChannelGroup1"               ""               "Sensor_Y"
            2             "ChannelGroup2"               ""               "Pulse"
            2             "ChannelGroup2"               ""               "Sensor_X"
            2             "ChannelGroup2"               ""               "Sensor_Y"
            3             "ChannelGroup3"               ""               "Pulse"
            3             "ChannelGroup3"               ""               "Sensor_X"
            3             "ChannelGroup3"               ""               "Sensor_Y"
```

Append Data to a Single Channel Group

You can append the data from all the TDMS-files to a single channel group. By specifying a particular channel group name, the data is append to that channel group.

The channel names map to the table variable names, therefore the data is appended to all the channels in the channel group. If there is a table variable that does not already exist as channel, a new channel gets added to the channel group.

```
reset(ds);
while(hasdata(ds))
    tdmswrite(fileSingleChannelGroup, read(ds), ChannelGroupName="Dataset")
end
```

On inspecting the contents of the file, the `NumSamples` property of `ChannelList` increased to 3 × 20000, indicating that the data is appended to the file.

```
info = tdmsinfo(fileSingleChannelGroup);
info.ChannelList
```

```
ans=3×8 table
    ChannelGroupNumber    ChannelGroupName    ChannelGroupDescription    ChannelName    ChannelDe
    __________________    ________________    _______________________    ___________    _________

            1                "Dataset"                  ""               "Pulse"
            1                "Dataset"                  ""               "Sensor_X"
            1                "Dataset"                  ""               "Sensor_Y"
```

Analyze TDMS-Files Using Tall Tables

This example shows how to access a TDMS datastore using tall tables in MATLAB®.

For this example, you have measurent vibration measurements of a rotating shaft recorded across multiple TDMS-files. Your goals is to extract portions of the files using tall tables and visually analyze them.

Set up the Datastore and Create a Tall Table

Identify the location of the TDMS datastore files. Create a tall table from the TDMS datastore with an applied transform. The `tdmsDatastore` function can be applied to either a set of multiple files, or a single large file. Here a transform is used to convert the returned datastore type from a cell array to a table.

```
folder = "RotatingShaftAnalysis";
mapreducer(0); % Sets the global execution environment to be the local MATLAB session.
tData = tall( transform(tdmsDatastore(folder), ...
    @(cellData)table2timetable(cellData{1}, TimeStep=milliseconds(0.1))) );
```

Identify Data of Interest

Extract Top and Bottom Rows of the TDMS Tall Table

You can extract the first or last N rows using `head` or `tail`, respectively.

```
tHead = head(tData, 1000);
tTail = tail(tData, 1000);
```

Extract Specific Data

You can also apply relational operators or colon expressions to filter the data. The `?` characters indicate that evaluation is deferred until you call `gather`.

```
filteredPulse = tail(tData(tData.Pulse > 2,:), 1000)
```

```
filteredPulse =

  M×3 tall timetable

    Time    Pulse    Sensor_X    Sensor_Y
    ____    _____    ________    ________

     ?        ?          ?           ?
     ?        ?          ?           ?
     ?        ?          ?           ?
     :        :          :           :
     :        :          :           :

Preview deferred. Learn more.
```

```
tDataInRange =  tData(5000:10000,:)
```

```
tDataInRange =

  5,001×3 tall timetable
```

```
    Time    Pulse    Sensor_X    Sensor_Y
    ____    _____    ________    ________

     ?        ?         ?           ?
     ?        ?         ?           ?
     ?        ?         ?           ?
     :        :         :           :
     :        :         :           :

Preview deferred. Learn more.
```

Find the Number of Rows in the Table

All table operations can be applied to tall table. One such operation is `height`, to get the total number of rows in the collection.

```
tRows = height(tData);
```

Gather Data for Evaluatation

Finally, evaluate the computations using `gather`, which triggers execution of the operations that were delayed earlier. You can use `gather` to pass all the variables of interest in a single function call.

```
[tHead, tTail, filteredPulse, tDataInRange, tRows] = gather(tHead, tTail, filteredPulse, tDataInR
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.2 sec
- Pass 2 of 2: Completed in 0.051 sec
Evaluation completed in 0.38 sec

tHead=1000×3 timetable
       Time          Pulse        Sensor_X    Sensor_Y
    ___________    __________    ________    ________

    0 sec          0.00087333    0.12865      0.5479
    0.0001 sec      0.0007626    0.13118     0.55483
    0.0002 sec     0.00085302    0.12971     0.53442
    0.0003 sec     0.00081109    0.12775      0.5556
    0.0004 sec     0.00078684    0.12813     0.55271
    0.0005 sec     0.00084647    0.13173     0.54674
    0.0006 sec     0.00081109     0.1281     0.52979
    0.0007 sec     0.00078488    0.12769     0.52421
    0.0008 sec     0.00074556    0.13112      0.5452
    0.0009 sec     0.00084647    0.12977     0.55233
    0.001 sec      0.00080191    0.13064     0.53133
    0.0011 sec     0.00088578     0.1315     0.52998
    0.0012 sec      0.0008314    0.13487     0.51265
    0.0013 sec     0.00080388    0.13436      0.5269
    0.0014 sec     0.00083074    0.13597     0.52844
    0.0015 sec     0.00083336    0.13029     0.53499
         ⋮

tTail=1000×3 timetable
       Time          Pulse        Sensor_X    Sensor_Y
    ___________    __________    ________    ________

    1.9 sec        -0.0015896     0.1368      0.5503
```

```
    1.9001 sec    -0.0016119    0.13764      0.5842
    1.9002 sec    -0.0016119    0.14088     0.55338
    1.9003 sec    -0.0016571    0.13696     0.55896
    1.9004 sec    -0.0016938    0.13838     0.56551
    1.9005 sec    -0.0016394    0.13716      0.5894
    1.9006 sec    -0.0016401    0.13857     0.55954
    1.9007 sec    -0.0017646    0.13809     0.56879
    1.9008 sec    -0.0016237    0.13918      0.5711
    1.9009 sec    -0.0016938    0.13934     0.53142
    1.901 sec     -0.0016027    0.13822     0.55607
    1.9011 sec    -0.0016217    0.13854     0.56821
    1.9012 sec    -0.0016951    0.14184     0.57283
    1.9013 sec    -0.0016283    0.13783     0.55858
    1.9014 sec     -0.001743     0.1361     0.55627
    1.9015 sec    -0.0017508    0.13786     0.54201
      ⋮

filteredPulse=27×3 timetable
       Time       Pulse     Sensor_X    Sensor_Y
    __________    ______    ________    ________

    0.1985 sec    2.7111     0.17406    0.57833
    0.6452 sec    3.3274     0.12907    0.53788
    1.0919 sec    3.3197     0.11504    0.53865
    1.5386 sec    2.7469     0.14389    0.51669
    1.9853 sec    2.8597    0.075178    0.55406
    0.2596 sec    3.3864     0.20214    0.52591
    0.5694 sec     2.182     0.17233     0.5367
    0.5695 sec    2.5165     0.17727    0.60874
    0.8793 sec    3.4009     0.13664     0.5315
    1.1892 sec    2.2825     0.12525    0.55307
    1.1893 sec    2.4268    0.097772    0.60046
    1.4991 sec    3.4114     0.14396    0.54325
    1.8089 sec    2.3788     0.11822    0.55134
    1.809 sec     2.3314    0.081532    0.58023
    0.0948 sec    3.2981    0.090041    0.57322
    0.3035 sec    2.3475     0.14608    0.56763
      ⋮

tDataInRange=5001×3 timetable
       Time          Pulse       Sensor_X    Sensor_Y
    __________    ___________    ________    ________

    0.4999 sec    -0.00076734    0.13439     0.53519
    0.5 sec       -0.00077651    0.13228     0.53557
    0.5001 sec    -0.00081844    0.13125     0.54597
    0.5002 sec    -0.00086627    0.12756     0.53557
    0.5003 sec    -0.00079617    0.12939     0.54231
    0.5004 sec    -0.00080272    0.12887     0.55175
    0.5005 sec    -0.00075685    0.12964     0.55849
    0.5006 sec    -0.00085841    0.12788     0.57198
    0.5007 sec    -0.00084007    0.12406     0.54289
    0.5008 sec    -0.00092131    0.12685     0.55792
    0.5009 sec    -0.00078765    0.12656     0.54963
    0.501 sec      -0.0007942    0.12801     0.56235
    0.5011 sec    -0.00076472    0.12984     0.54732
```

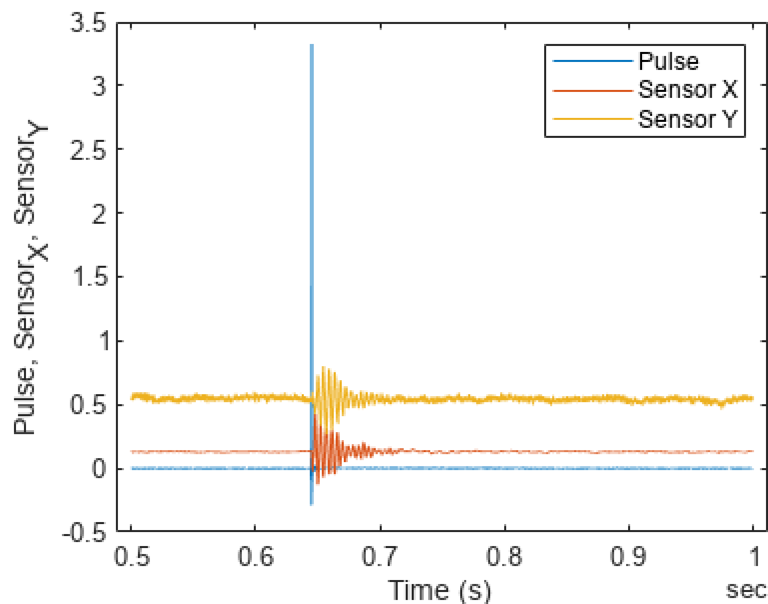
```
    0.5012 sec    -0.00074113    0.12968     0.57872
    0.5013 sec    -0.00083744    0.13102     0.58065
    0.5014 sec    -0.00078175    0.12935     0.58334
       :
```

```
tRows = 60000
```

Plot the Data

Use a plot to visually analyze the filtered data.

```
plot(tDataInRange.Time, tDataInRange.Pulse);
hold on
plot(tDataInRange.Time, tDataInRange.Sensor_X);
hold on
plot(tDataInRange.Time, tDataInRange.Sensor_Y);
legend('Pulse','Sensor X', 'Sensor Y');
xlabel('Time (s)')
ylabel('Pulse, Sensor_X, Sensor_Y');
```



Summarize Tall Table Contents

You can call the `summary` function to trigger immediate evaluatation of all the table variables.

```
summary(tData);
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.094 sec
Evaluation completed in 0.13 sec

RowTimes:

    Time: 60,000×1 duration
        Values:
```

```
            Min         0 sec
            Max    1.9999 sec

Variables:

    Pulse: 60,000×1 double
        Values:

            Min    -0.69755
            Max      3.4114

    Sensor_X: 60,000×1 double
        Values:

            Min     -0.212
            Max    0.50583

    Sensor_Y: 60,000×1 double
        Values:

            Min    0.18122
            Max    0.94865
```

To learn more about tall tables, see “Tall Arrays”. Note: `tdmsDatastore` is not enabled to work with Parallel Computing Toolbox.

Impulse Response Measurement Using a NI USB-4431 Device

This example shows how to measure an impulse response using a National Instruments (NI) USB-4431 sound and vibration device. You set up the device, create an excitation signal to play, and simultaneously record the response. Finally, you compute the response from the recording and plot the results.

For this example, you need Audio Toolbox™ and Data Acquisition Toolbox™. You also need to have installed the NI drivers (recommended) or the MATLAB NI support package.

Device Setup

Verify if the NI USB-4431 is connected and the proper drivers are installed using the `daqlist` command.

```
daqlist("ni")
```

```
ans=1×4 table
    DeviceID                Description                   Model                 DeviceInfo
    ________    ______________________________________    ___________    ______________________________

     "Dev1"     "National Instruments(TM) USB-4431"    "USB-4431"     1×1 daq.ni.DeviceSpecializa
```

Next, knowing that this device only supports clocked operations, you can disable the warning.

```
ws = warning("off","daq:Session:clockedOnlyChannelsAdded");
% restore warning state when cleared
oc = onCleanup(@() warning(ws));
```

Setup the NI device with a sampling rate of 48 kHz, one input, and one output.

To do so, create a `daq` object for NI devices and set `Rate` to 48 kHz.

```
FS = 48e3;
dq = daq("ni");
dq.Rate = FS;
dev = "Dev1";
```

Add one input and one output. In this example, the microphone has its own pre-amplifier, so set the measurement type to `Voltage`. The output is connected to a powered loudspeaker and the measurement type is set to `Voltage`.

```
addinput(dq,dev,"ai0","Voltage");
addoutput(dq,dev,"ao0","Voltage");
```

Stimulus Signal

Create an exponential swept sine going from 20 Hz to 24 kHz over a period of 3 seconds, followed by one second of silence. This limits the maximum impulse response length to one second. You can also set the output level, in this case -18 dB.

```
irDur = 1;
sweepDur = 3;
outputLevel = -18;
sweepRange = [20 24000];
```

```
exc = sweeptone( ...
    sweepDur, irDur, FS, ...
    ExcitationLevel=outputLevel, ...
    SweepFrequencyRange=sweepRange);
```

Response Measurement

Using the `readwrite` method of the `daq` object (`dq`), play the swept sine stimulus (`exc`) and simultaneously record the response.

```
data = readwrite(dq,exc);
dq = []; % release
```

Compute the Impulse Response

Use the `impzest` function to compute the impulse response (`ir`) from the stimulus (`exc`) and its response.

```
response = data.Variables;
ir = impzest(exc,response);
```

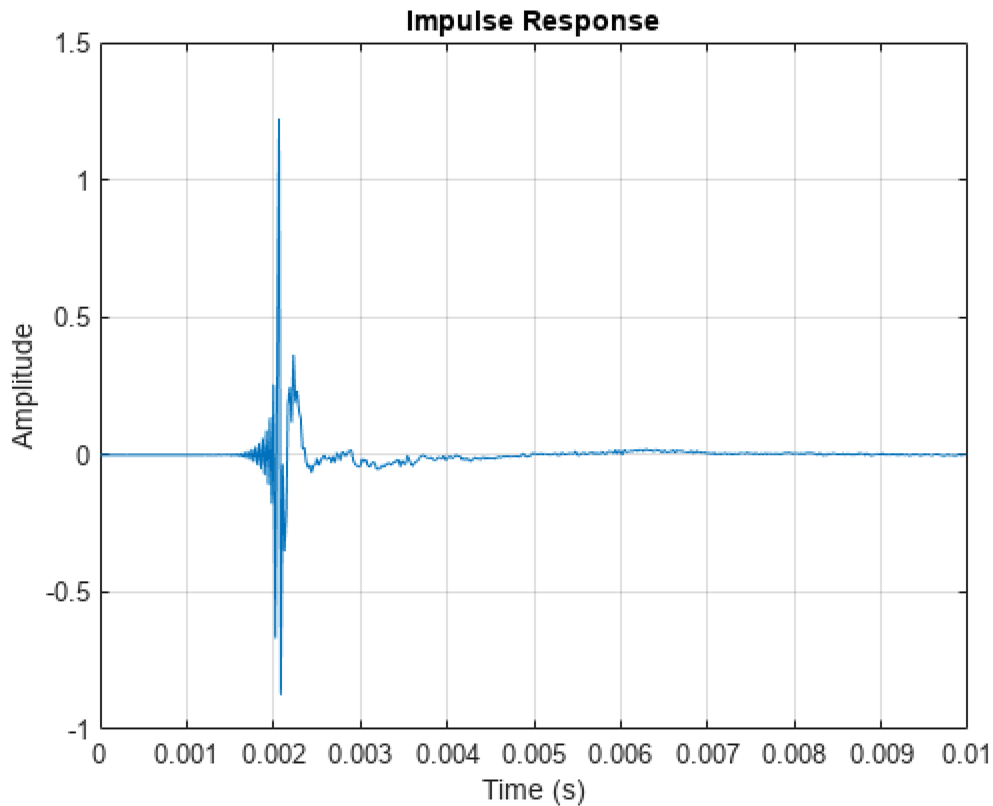
Create the corresponding time vector.

```
t_sec = (0:length(ir)-1).'/FS;
```

Display the Results

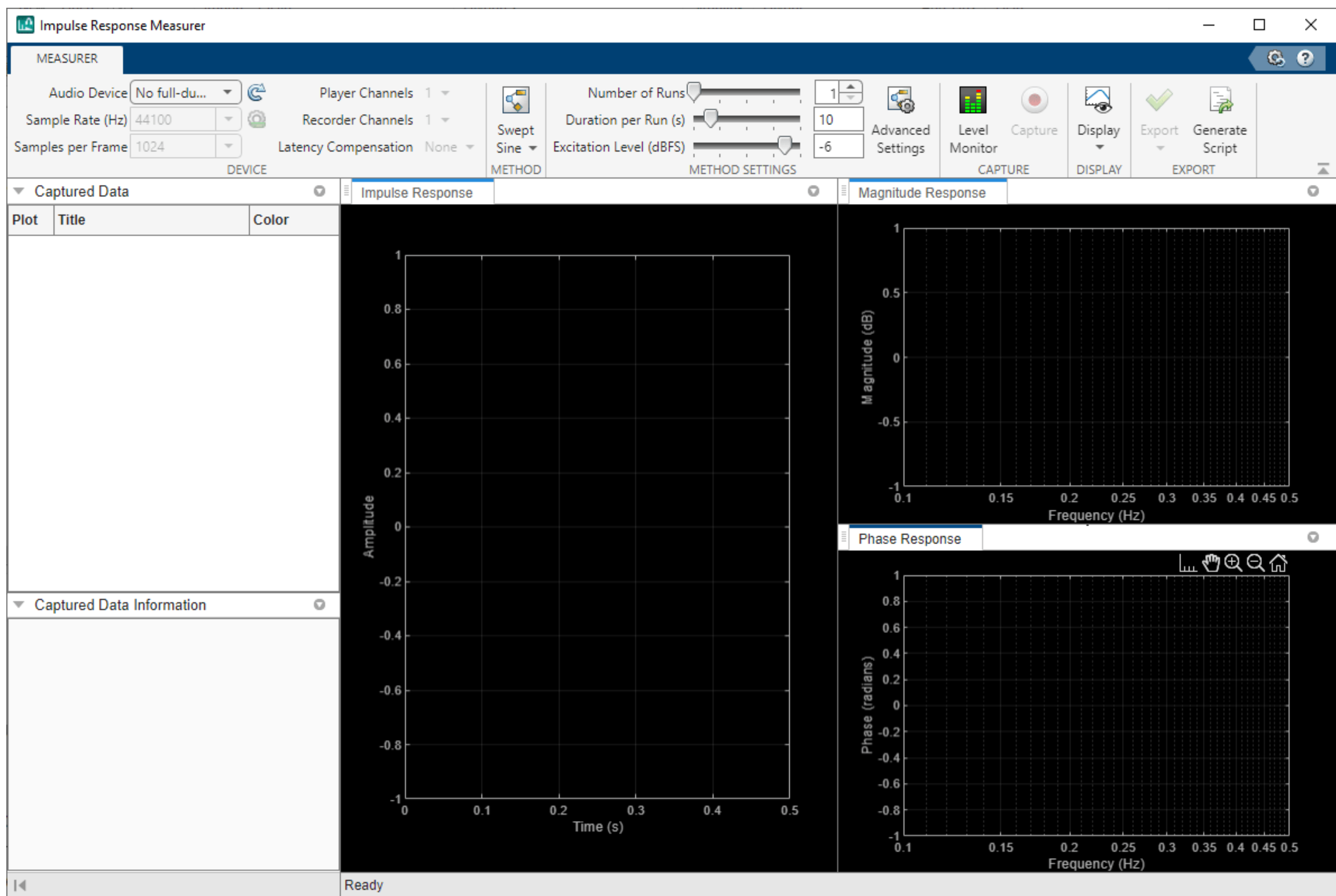
Plot the first 10 milliseconds, in this case, 480 samples.

```
totalDur = sweepDur*FS;
n = 480;
plot(t_sec(1:n), ir(1:n))
title("Impulse Response")
ylabel("Amplitude")
xlabel("Time (s)")
grid on
```



Impulse Response Measurer App

Another way of writing the code above is to start from a script generated from the **Impulse Response Measurer** app. Start the app by entering `impulseResponseMeasurer` at the command prompt. You can also click the app icon on the **Apps** tab of the MATLAB® Toolstrip. Even without a supported device connected, you can set the **Method Settings**, **Display Settings** and a linear or log scale for magnitude and phase responses (using the toolbar that appears when hovering the mouse over these plots). Then, click **Generate Script**. A new document will appear in the editor with code for an audio device, that you will modify to use the NI USB-4431.



Script Customization

First, you may want to make this a function by adding `function capture = irm_usb4431` to the top of the file, and an `end` statement before the first embedded function.

The function to interface with the NI device can take a complete signal, without proceeding frame by frame. Consequently, remove the code for "`Allocate the input/output buffers`" and "`Copy the excitation to the output buffer`".

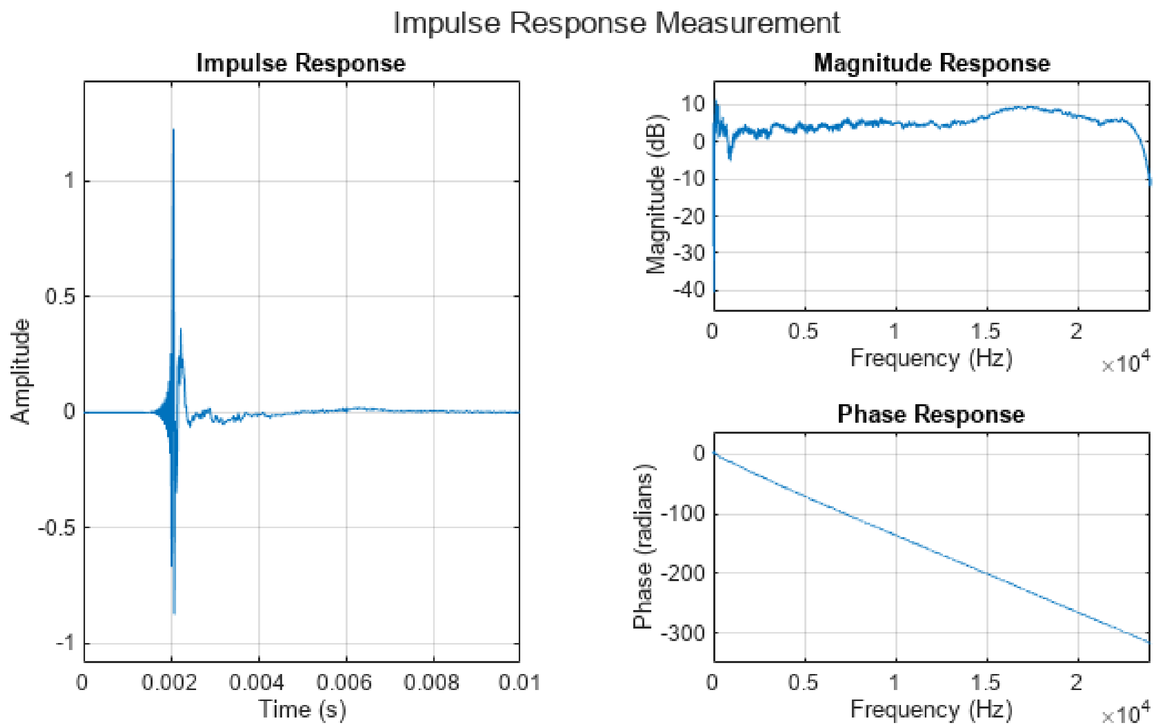
The code required to "`Setup the audio device`" can be replaced by code to setup the NI USB-4431 device. Set the sample rate of the device and add an output and at least one input.

```
dq = daq("ni");
dq.Rate = 48e3; % Sampling rate
addinput(dq,"Dev1","ai0","Voltage");
addoutput(dq,"Dev1","ao0","Voltage");
```

The playback and capture loop can be replaced by a `readwrite` statement. Get the results from `data.Variables` instead of the buffer. Also get the handle from the first figure to facilitate zooming in.

Now run the customized function `irm_usb4431` to obtain the capture data and plot the results and zoom in.

```
[capture,ax1] = irm_usb4431();
xlim(ax1,[0 10e-3]);
```



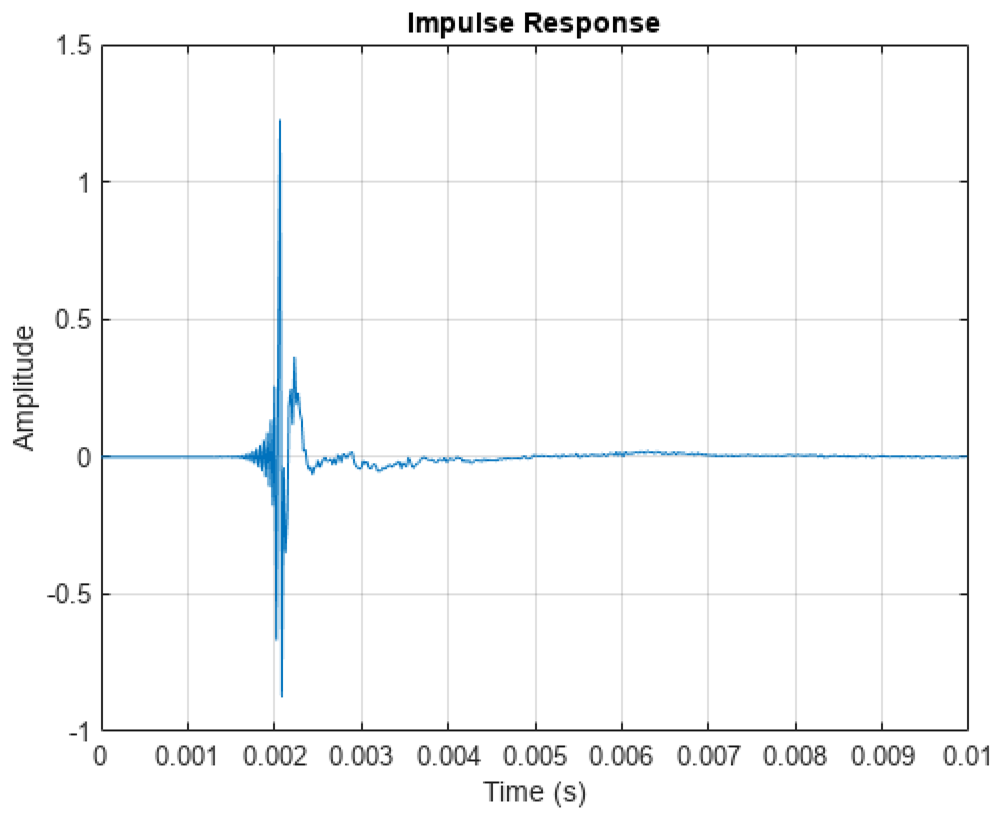
The data is also available programmatically.

```
capture
```

```
capture = struct with fields:
      ImpulseResponse: [1×1 struct]
    MagnitudeResponse: [1×1 struct]
        PhaseResponse: [1×1 struct]
```

For example, plot the first 480 samples of the impulse response.

```
figure % new figure
plot(capture.ImpulseResponse.Time(1:480), capture.ImpulseResponse.Amplitude(1:480))
title("Impulse Response")
ylabel("Amplitude")
xlabel("Time (s)")
grid on
```


Impulse Response
1.5
1
0.5
0
-0.5
-1
Amplitude
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
0.009
0.01
Time (s)