NI-DAQmx Key Concepts

January 2008, 371407F-01

NI-DAQmx Key Concepts covers important concepts in NI-DAQmx such as <u>channels</u> and <u>tasks</u>. The ways that NI-DAQmx handles timing, triggering, buffering, and signal routing are also central in the NI-DAQmx API.

© 2005–2008 National Instruments Corporation. All rights reserved.

Channels and Tasks in NI-DAQmx

Virtual channels and tasks are fundamental components of NI-DAQmx. <u>Virtual channels</u>, or sometimes referred to generically as channels, are software entities that encapsulate the physical channel along with other

channel specific information—range, terminal configuration, and custom scaling—that formats the data. <u>Tasks</u> are collections of one or more virtual channels with timing, triggering, and other properties.

Channels, Physical Versus Virtual

A physical channel is a terminal or pin at which you can measure or generate an analog or digital signal. A single physical channel can include more than one terminal, as in the case of a differential analog input channel or a digital port of eight lines. Every physical channel on a device has a unique name (for instance, SC1Mod4/ai0, Dev2/ao5, and Dev6/ctr3) that follows the NI-DAQmx physical channel naming convention.

Virtual channels are software entities that encapsulate the physical channel along with other channel specific information—range, terminal configuration, and custom scaling—that formats the data. To create virtual channels, use the DAQmx Create Virtual Channel function/VI or the DAQ Assistant.

Virtual channels created with the DAQmx Create Virtual Channel function/VI are called *local virtual channels* and can only be used within the <u>task</u>. With this function/VI, you choose the name to assign for the virtual channel, which is used in the rest of the NI-DAQmx software framework to refer to the physical channel.

If you create virtual channels with the DAQ Assistant, you can use them in other <u>tasks</u> and reference them outside the context of a task. Because these channels can apply to multiple tasks, they are called *global virtual channels*. You can select global virtual channels with the NI-DAQmx API or DAQ Assistant and add them to a task. If you add a global virtual channel to several tasks and modify that global virtual channel with the DAQ Assistant, the change applies to all tasks that use that global virtual channel.

See Also

<u>Creating Virtual Channels with the API</u> <u>Creating Virtual Channels with the DAQ Assistant</u> <u>Device-Specific Physical Channels</u> <u>Device-Specific Internal Channels</u>

Creating Virtual Channels with the API

The following example illustrates the difference between physical and virtual channels and demonstrates how to create virtual channels with the API.

Problem

Create an NI-DAQmx virtual channel to measure temperature in the range 50° C to 200° C using a J-type thermocouple wired to channel 0 on an M Series device configured as Device 1. Use LabVIEW or LabWindows™/CVI™ to write your application.

Solution

- 1. Call the AI Temp TC instance of the DAQmx Create Virtual Channel VI in LabVIEW (DAQmxCreateAIThrmcplChan function in LabWindows/CVI).
- 2. Use Dev1/ai0 as the physical channel on the device to which the thermocouple signal is connected.
- 3. Specify myThermocoupleChannel as the name to assign to your virtual channel.
- 4. Select the appropriate values for the thermocouple type and range inputs. NI-DAQmx applies these attributes to the virtual channel.

You have now created a virtual channel.

See Also

Choosing Whether to Use the API or the DAQ Assistant Device-Specific Physical Channels Device-Specific Internal Channels

Types of Virtual Channels

You can create a number of different types of virtual channels, depending on the signal type—analog, digital, or counter—and direction (input or output). These channels can be either <u>global virtual channels or local</u> <u>virtual channels</u>. For information on specific functions/VIs, refer to the NI reference help for your ADE.

Analog Input Channels

Analog input channels measure different physical phenomena using a variety of sensors. The type of channel to create depends on the type of sensor and/or phenomenon you want to read. For instance, you can create channels for measuring temperature with a thermocouple, measuring current, measuring voltage, and measuring voltage with excitation.

Analog Output Channels

NI-DAQmx supports two types of phenomena, voltage and current. You can use custom scales if the output from the device relates to another unit of measure.

Digital Input/Output Channels

For digital channels, you can create both <u>line-based</u> and <u>port-based</u> digital channels. A line-based channel can contain one or more digital lines from one or more ports on a device. Reading or writing to a line-based channel does not affect other lines on the hardware. You can split lines in a particular port between multiple channels and use those channels simultaneously within one or multiple tasks, but the lines in a given channel must all be input lines or all be output lines. Additionally, all channels in a <u>task</u> must be either input channels or output channels. Some devices also require that the lines of a given port all be input lines or output lines. Check your device documentation for the capabilities of your device.

A port-based channel represents a fixed collection of lines on the device. Reading or writing to a port affects all the lines on the port. The number of lines in the port (commonly referred to as port width) is hardware dependent and typically varies from 8 lines (MIO device) to 32 lines (SCXI digital modules).

Counter Input/Output Channels

NI-DAQmx supports several types of counter input and output channels for different types of counter measurements and generations. To find out more about counter measurements and terminals used for common applications, refer to <u>Counter Parts in NI-DAQmx</u>.

Physical Channel Syntax

Use this syntax to refer to physical channels and groups of physical channels in NI-DAQmx.

Physical Channel Names

Physical channel names consist of a device identifier and a slash (/) followed by a channel identifier. For example, if the physical channel is Dev0/ai1, the device identifier is Dev0, and the channel identifier is ai1. MAX assigns device identifiers to devices in the order they are installed in the system, such as Dev0 and Dev1. You also can assign arbitrary device identifiers with MAX.

For analog I/O and counter I/O, channel identifiers combine the type of the channel, such as analog input (ai), analog output (ao), and counter (ctr), with a channel number such as the following:

ai1

ctr0

For digital I/O, channel identifiers specify a port, which includes all lines within a port:

port0

Or, the channel identifier can specify a line within a port:

port0/line1

All lines have a unique identifier. Therefore, you can use lines without specifying which port they belong to. For example, line31—is equivalent to port3/line7 on a device with four 8-bit ports.

Physical Channel Ranges

To specify a range of physical channels, use a colon between two channel numbers or two physical channel names:

Dev0/ai0:4

Dev0/ai0:Dev0/ai4

For digital I/O, you can specify a range of ports with a colon between two port numbers:

Dev0/port0:1

You also can specify a range of lines:

Dev0/port0/line0:4 Dev0/line0:31

You can specify channel ranges in reverse order:

Dev0/ai4:0

Dev0/ai4:Dev0/ai0

Dev0/port1/line3:0

Physical Channel Lists

Use commas to separate physical channel names and ranges in a list as follows:

Dev0/ai0, Dev0/ai3:6 Dev0/port0, Dev0/port1/line0:2

See Also

Device-Specific Physical Channels Device-Specific Internal Channels Multidevice Tasks

Digital Lines, Ports, and Port Width

Digital lines and ports are important parts of a digital input/output system.

Line—A line is an individual signal. It refers to a physical terminal. The data that the line carries are called bits, binary values that are either 1 or 0. The terms *line* and *bit* are fairly interchangeable. For example, an 8-bit port is the same as a port with eight lines.

Port—A port is a collection of digital lines. Usually, the lines are grouped into an 8-bit or 32-bit port.

Port Width—The port width refers to the number of lines in a port. For example, a device with one port with eight lines has a port width of eight.



Channel Name Generation

NI-DAQmx assigns names to local virtual channels that you create programmatically with the NI-DAQmx API when you do not provide a name for each local virtual channel.

Physical Channel Names	Name To Assign	Generated Local Virtual Channel Names
Dev1/ai0:1	—	Dev1/ai0, Dev1/ai1
Dev1/ai0:7	"foo"	foo0, foo1,, foo7
Dev1/ai0:7	"a0:3, b"	a0, a1, a2, a3, b0, b1, b2, b3

Naming Channels, Tasks, and Scales

Use the following guidelines to name your channels, tasks, and scales:

- Use any alphanumeric characters.
- Do not use nonalphanumeric characters with the following exceptions:
 - In NI-DAQmx 7.4 or later, dashes are allowed in channel, task, and scale names.
 - Spaces are allowed.
 - You can use underscores within the channel, task, or scale name, but you cannot use leading underscores, such as Dev1.



Note You can use other nonalphanumeric characters when creating channels, tasks, and scales, but exporting that configuration to another system might not work correctly, especially if the operating system is in a different language.

You must use no more than 256 characters.

Switch Channels

You can program your NI switch modules with NI-DAQmx. This section covers switch basics, including <u>switch channel and relay strings</u>, <u>connection/disconnection syntax</u>, and <u>switch scan list syntax</u>.

Switch Channel Strings and Switch Relay Strings

Switch channel strings and switch relay strings identify a specific channel or relay of a switch. These strings are typically used when an operation or query is performed on the switch—connect, disconnect, find path, and so on. These strings are constructed in very similar fashions.

A switch channel or relay string can be either of the following:

- A combination of the switch device/channel name or a switch device/relay name (for example, Dev1/ch0).
- A switch relay name without the switch channel name (for example, ch0). This syntax is only valid as a shortcut and can only be used if a string with a switch device/channel name or a switch device/relay name (as shown above) was previously used to specify a device.

You can find the switch channels and switch relay strings in LabVIEW and LabWindows/CVI from the switch I/O name control. The name control should list valid channel/relay names for your current switch hardware configuration.

Connection and Disconnection List Syntax

Use connection lists and disconnection lists with DAQmx Switch Connect (Multiple) and DAQmx Switch Disconnect (Multiple), respectively. These function/VI list parameters use a rich and versatile syntax to describe the operation that the function/VI performs. Because these lists use the same syntax, they are referred to here as connection lists for simplicity. The connection list syntax is similar to the switch scanlist syntax with few exceptions.

Connection/disconnection lists are strings composed of one or more switching operations. For those lists that contain multiple operations, commas separate each operation:

Operation1, Operation2, Operation3

Switching operations can connect/disconnect channels in one of two ways:

 Specify the endpoints. When you specify the endpoints, NI-DAQmx searches for an available path between the endpoints to connect/disconnect. The syntax for specifying the endpoints is as follows:

channel1 -> channel2

where the channel names of a switch are separated by an arrow (->).

The two specified channels must reside on the same device and an available path between them must exist. If the path between the two channels includes one or more channels, these intermediate channels should have their usage mode marked as "reserved for routing."

For example, if a path exists between channel0, com0, and AB0, it is possible to connect channel0 to AB0 by marking the usage mode of com0 as "reserved for routing." The operation in the string would appear as the following:

channel0 -> AB0

When multiple intermediate channels exist and their usage is marked as "reserved for routing," NI-DAQmx selects the intermediate channel to use.

• Specify an explicit path. When you specify an explicit path, you define the endpoints as well as any intermediate channels of the path. Specifying an explicit path can be useful in applications in which you have calibrated your system based on a specific path through the switch.

The syntax for specifying an explicit path is as follows:

[channel1 -> channel2 -> channel3]

For example, if you want to connect column 1 and column 5 of a matrix and use row 2 to complete that connection, you would have to mark the usage of row 2 as "reserved for routing" and use the following string:

[c1 -> r2 -> c5]

The following is additional information on connection and disconnection lists:

 A connection/disconnection list string can contain switching operations on different switch modules. To do so, add the switch name and a / before the operation. Refer to the following example.

Switch1/ch1 -> com0 , [switch2/c0 -> r2 -> c5] , switch3/r0 -> c4

- Before execution, NI-DAQmx validates the connection/disconnection list. If any errors are returned, NI-DAQmx aborts execution of the list.
- The order of the operations in the connection/disconnection list does not guarantee their order of execution. To ensure a specific order, use multiple connection/disconnection lists.

• NI-DAQmx ignores any white space, and inputs are not case sensitive. You can use spaces and carriage returns to improve readability.

Scan Lists

A scan list is a string composed of device names, channel names, and characters that define connections, disconnections, triggering, and timing of the scan.

Switch Modules in NI-DAQmx

MAX supplies the default device name (or resource name) you use in a scan list. To find the default name, open MAX, and go to **Devices and Interfaces»NI-DAQmx Devices**. An example name is **SCXI-1130:"SC1Mod1"**. The part of the name in quotes—for instance, SC1Mod1 or Dev1—is what you use in the scan list.

Scan List Characters

The following characters can be used in a scan list.

Character	Definition
->	Used in a connect action (/Dev1/channel1->channel2). For example, the string /SC1Mod1/ch0->com0 connects CH0 to COM0.
~	Used with '->' in a disconnect action (~/Dev1/channel1- >channel2). Valid only in <u>No Action</u> mode. For example, ~/Dev1/ch0->com0 means disconnect CH0 and COM0.
- 7	Wait for debounce, send scan advanced output signal, then wait for trigger input.
&	Separates connect and/or disconnect actions. For example, the string /Dev1/ch0->com0 & /Dev1/ch9->com1 means connect CH0 to COM0 and CH9 to COM1 (in no particular order).
&&	Wait for debounce. For example, /Dev1/ch0->com0 && /Dev1/ch9->com1 means connect CH0 to COM0, wait for the relays to settle, then connect CH9 to COM1.
	Used in a channel range (channelX:Y, where <i>X</i> and <i>Y</i> are integers). Text containing a channel range represents multiple scan list entries. For example, the string /SC1Mod4/ch0:7->com0; represents 8 scan list entries. A semicolon must appear after the connect action using a channel range.
Tip N	NI-DAQmx ignores white space and line returns. Use these to

format the appearance of lengthy scan lists.

Scan List Entries

A scan list entry is the text delimited by semicolons (;). Scan lists are composed of one or more scan list entries. For example, the following scan list contains two scan list entries:

/Dev1/ch0->com0; /Dev1/ch1->com0;

Scan Modes

The scan mode affects how NI-DAQmx interprets the scan list string. Typical scanning applications use the Break Before Make scan mode.

Mode	Description
Break Before Make (default)	Connections from the previous <u>scan list entry</u> are automatically disconnected before executing the current scan list entry. Disconnect actions (~/Dev1/channel1->channel2) are not valid in this mode.
No Action	Connections remain connected until they are explicitly disconnected by a disconnect action.
Break After Make	Currently unsupported.

Scan List Examples

Example 1

Scan Mode: No Action

Scan List: /Dev1/ch0->com0; ~/Dev1/ch0->com0 && /Dev1/ch1->com0; ~/Dev1/ch1->com0 &&

Meaning:

- 1. Connect ch0 to com0.
- 2. Wait for debounce, send scan advanced signal, then wait for trigger input.
- 3. Disconnect ch0 from com0, and wait for debounce.
- 4. Connect ch1 to com0.
- 5. Wait for debounce, send scan advanced signal, then wait for trigger input.
- 6. Disconnect ch1 from com0, and wait for debounce.
- 7. If the scan is set to continuous, return to step 1; else, stop.

Example 2

Scan Mode: Break Before Make Scan List: /Dev1/ch0->com0; /Dev1/ch1->com0;

Meaning:

This scan list is equivalent to Example 1. Notice that the <u>disconnect</u> <u>actions</u> in Example 1 are no longer required.

Example 3

Scan Mode: Break Before Make Scan List: /Dev1/ch0:1->/Dev1/com0;

Meaning:

This scan list is equivalent to Example 1 and Example 2. This scan list uses a <u>channel range</u> to reduce typing.

Switch Scanning

Another method to operate relays in a switch module is through scanning. Scanning is typically used when timing of connections needs to be synchronized with an event of another device such as a measurement instrument.

Unlike the immediate operations, where the relay actuates immediately after calling DAQmx Switch Connect (or Disconnect) function/VI, scanning consists of setting up a list of connections to be made after an event.

Connection operations are entered in a scan list that is downloaded to the memory of the switch module. The first entry in the scan list is executed when the switch module is initiated using the DAQmx Start Task function/VI. The triggering settings determine how the switch advances through subsequent entries in the list. The scan list can be executed continuously or for a finite number of times.

After each connection, switch modules can generate a digital pulse called Advance Complete. This pulse is typically used to trigger another device, such as a DMM to take a measurement.

There are three scanning options—Software Trigger Scanning, Synchronous Scanning, and Handshaking—which ultimately determine the triggering scheme.

Software Trigger Scanning

In Software Trigger Scanning, the scan list starts when the DAQmx Start Task function/VI is called. Each subsequent entry is executed after each call of the DAQmx Send Software Trigger function/VI.

To write a software trigger scanning program, complete the following steps:

- 1. Set **Topology Name** of the switch module using DAQmx Switch Set Topology and Reset.
- 2. Set up the list of connections using the DAQmx Switch Create Scan List function/VI.
- 3. Set the **Trigger Type** attribute/property in the DAQmx Trigger class to Software.
- 4. Select number of times the scan list executes by setting the **Repeat Mode** attribute/property in DAQmx Switch Scan Class to Continuous or Finite.
- 5. Initialize the switch module using the DAQmx Start Task function/VI. The first entry in the scan list is executed and the switch waits for software triggers to execute the following entries in the list.
- 6. Execute each set of connections in the scan list by calling the DAQmx Send Software Trigger function/VI and specifying the Advance Trigger.
- 7. Terminate the scanning operation using the DAQmx Stop Task function/VI.
- 8. Release resources using the DAQmx Clear Task function/VI.

Refer to the Switch Scanning—Software Trigger VI for an example of software trigger scanning in LabVIEW.

Synchronous Scanning

In synchronous scanning, each entry in the scan list is executed after the switch receives a digital pulse. This digital pulse is the Advance Trigger. A common use of synchronous scanning is with a measurement device like a digital multimeter (DMM). The DMM is programmed to take measurements at regular intervals and generate a digital pulse. When the switch receives this digital pulse, it advances to the next entry in its scan list. You must program the DMM interval time to account for the time required by the switch to actuate and settle.

To write a synchronous scanning program, complete the following steps:

- 1. Set **Topology Name** of the switch module using the DAQmx Switch Set Topology and Reset function/VI.
- 2. Set up the list of connections using the DAQmx Switch Create Scan List function/VI.
- 3. Set the **Trigger Type** attribute/property to Digital Edge and **source** appropriately in the DAQmx Trigger function/VI. The source should coincide with the destination of the digital pulse sent by the measurement device.
- 4. Select number of times the scan list executes by setting the **Repeat Mode** attribute/property in DAQmx Switch Scan Class to continuous or finite.
- 5. Configure the digital pulse generated after each connection. In synchronous scanning, this digital pulse is not used. Set the **Advance Complete Event:Output Terminal** in DAQmx Export Signal class to an empty string.
- 6. Initialize the switch module using the DAQmx Start Task function/VI. The first entry in the scan list is executed and the switch waits for digital pulses to execute the following entries in the list. Each time the switch module receives a digital pulse, it executes an entry in its scan list.
- 7. Terminate the scanning operation using the DAQmx Stop Task function/VI.
- 8. Release resources using the DAQmx Clear Task function/VI.

Refer to the Switch Scanning with DMM—Synchronous VI for an example of synchronous scanning in LabVIEW.

Handshaking

Handshaking is very similar to synchronous scanning except the switch sends a digital pulse back to the measurement device after each set of connections. In this case, the measurement device accepts a pulse as a trigger for its measurements instead of taking measurements at regular intervals. If you use a DMM, it needs to be initialized and ready to accept a trigger for its first measurement. The switch is then initiated, executes its first entry in the scan list, waits for a digital pulse to execute its next entry, and generates a digital pulse (Advance Complete). When the DMM receives this digital pulse, it takes the first measurement and generates a digital pulse. When the switch receives this pulse, it executes the next entry in the scan list, generates another digital pulse, and so on.

To write a handshaking program, complete the following steps:

- 1. Set **Topology Name** of the switch module using the DAQmx Switch Set Topology and Reset function/VI.
- 2. Set up the list of connections using the DAQmx Switch Create Scan List function/VI.
- 3. Set the **Trigger Type** to Digital Edge and the **source** appropriately in the DAQmx Trigger function/VI. The source should coincide with the destination of the digital pulse sent by the measurement device.
- Configure the digital pulse generated after each set of connections. Set the Advance Complete Event:Output Terminal in DAQmx Export Signal class appropriately. This output terminal should coincide with the source of the DMM input trigger.
- 5. Select number of times the scan list executes by setting the **Repeat Mode** attribute/property in DAQmx Switch Scan Class to continuous or finite.
- 6. Initialize the switch module using the DAQmx Start Task function/VI. The first entry in the scan list is executed, and the switch waits for digital pulses to execute the following entries in the list. Each time the switch module receives a digital pulse, it executes an entry in its scan list and generates a digital pulse.
- 7. Terminate the scanning operation using the DAQmx Stop Task function/VI.

8. Release resources using the DAQmx Clear Task function/VI.

Refer to Switch Scanning with DMM—Handshaking VI for an example of handshaking in LabVIEW.

Topology

A switch topology is an abstract representation of the channels and relays in a switch module. The topology establishes the default states for all relays on a module. It also defines the channel names. Some switches can use multiple topologies. Notice that terminal blocks or accessories can force the switch to use a given topology or set of topologies.

The power up default topology for each switch is set by the hardware. You can, however, change the default topology that the switch goes into when device reset is called by configuring the topology in MAX.

The three major switch topologies are general purpose, multiplexer, and matrix.

Channel Usage

Every switch channel has a usage type associated with it. By default, most channels are considered load channels. In terms of usage, a load channel is essentially a channel with no special capabilities. The NI-DAQmx switch API offers two additional usage types for a given channel for added software protection against unintentional damage to your system. These additional types are source and reserved for routing. Setting a channel as source indicates to NI-DAQmx that a signal source is connected to this channel. NI-DAQmx does not allow two user-defined source channels to be directly or indirectly connected. Setting a channel as reserved for routing indicates to NI-DAQmx that you are not planning on wiring directly to the channel and that it is available for NI-DAQmx for routing as needed.

Note Configuring the usage of a channel provides additional software protection when using the Connect/Disconnect or scanning functions/VIs, but it does not provide additional protection when manipulating relays directly using the Open/Close Relay functions/VIs. National Instruments recommends against mixing the two types of function/VI calls.

You can change the default usage type for channels while configuring devices in MAX. By changing the default values for channel usage on each topology, you are setting the defaults that will be used for those attributes/properties when a reset is called.
Tasks in NI-DAQmx

A task is a collection of one or more virtual channels with timing, triggering, and other properties. Conceptually, a task represents a measurement or generation you want to perform. All channels in a task must be of the same I/O type, such as analog input or counter output. However, a task can include channels of different measurement types, such as an analog input temperature channel and an analog input voltage channel. With <u>some devices</u>, you can include channels from multiple devices in a task. To perform a measurement or a generation with a task, follow these steps:

- 1. Create or load a task. You can create tasks interactively with the DAQ Assistant or programmatically in your ADE such as LabVIEW or LabWindows/CVI.
- 2. Configure the channel, timing, and triggering properties as necessary.
- 3. Optionally, perform various <u>task state transitions</u> to prepare the task to perform the specified operation.
- 4. Read or write samples.
- 5. Clear the task.

If appropriate for your application, repeat steps 2 through 4. For instance, after reading or writing samples, you can reconfigure the virtual channel, timing, or triggering properties and then read or write additional samples based on this new configuration.

If properties need to be set to values other than their defaults for your task to be successful, your program must set these properties every time it executes. For example, if you run a program that sets property A to a nondefault value and follow that with a second program that does not set property A, the second program uses the default value of property A. The only way to avoid setting properties programmatically each time a program runs is to use virtual channels and/or tasks created in the DAQ Assistant.

See Also

Creating Tasks with the API

Creating Tasks with the DAQ Assistant

Creating Tasks with the API

The following example illustrates how to create a task with the API:

Problem

Create an NI-DAQmx task to measure temperature in the range 50°C to 200°C using a J-type thermocouple that is wired to channel 0 on an M Series device configured as Device 1. Sample the temperature 10 times per second, and acquire 10,000 samples. Use LabVIEW or LabWindows/CVI to write your application.

Solution

- 1. Call the AI Temp TC instance of the DAQmx Create Virtual Channel VI in LabVIEW (DAQmxCreateAIThrmcplChan function in LabWindows/CVI).
- 2. Specify Dev1/ai0 as the physical channel for the device connected to the thermocouple signal.
- 3. Specify myThermocoupleChannel as the name to assign to your virtual channel.
- 4. Select the appropriate values for the thermocouple type and range inputs. NI-DAQmx applies these attributes to the virtual channel.
- 5. Call the Sample Clock instance of DAQmx Timing VI in LabVIEW (or DAQmxCfgSampClkTiming function in LabWindows/CVI), specifying a **rate** of 10 Hz and a **sample mode** of finite.
- 6. Call the DAQmx Start Task VI (DAQmxStartTask in LabWindows/CVI).
- 7. Call the Analog 1D DBL 1Chan NSamp instance of DAQmx Read VI (DAQmxReadAnalogF64 in LabWindows/CVI), specifying **number of samples per channel** as 10,000.
- 8. Call the DAQmx Stop Task VI (DAQmxStopTask function in LabWindows/CVI) after the desired number of samples have been acquired.
- 9. Call the DAQmx Clear Task VI (DAQmxClearTask function in LabWindows/CVI).

You have now created a task called myTemperatureTask that uses a local virtual channel called myThermocoupleChannel.



Note You also can use the DAQ Assistant to create the same task and generate the code to run the task.

See Also

Creating Tasks with the DAQ Assistant

Choosing Whether to Use the API or the DAQ Assistant

Using the Start Task function/VI

To explicitly start a <u>task</u>, call the Start Task function/VI. You auto-start a task when you perform some other operation that implicitly starts the task. For instance, calling a Read function/VI or a Write function/VI might implicitly start the task if one is not already started. How to specify this behavior depends on the operation that your task performs. By default, the Read function/VI and the Write function/VI for a single sample automatically starts a task.

Starting a Finite Measurement Task

If you have specified a task to perform a finite measurement, you do not need to call the Start Task function/VI, nor do you need to change the default behavior of the DAQmx Read function/VI. Calling the Read function/VI starts your task, performs the finite measurement, and stops the task after the last sample is read. The task returns to its <u>state</u> before you called the read operation. However, if you need to perform additional read operations after the task has been stopped (in other words, if you want to read earlier locations in the buffer), the default behavior is insufficient for two reasons:

- 1. The task is returned to the <u>Verified state</u> and the samples are no longer accessible.
- 2. Future calls of the Read function/VI start new read operations rather than reading from the completed operation.

For this situation, explicitly commit the task by calling the Control Task function/VI with the **action** parameter set to Commit. Then, after performing the initial read operation and before performing the subsequent read operations, set the **Auto-Start Read** attribute/property to False.

Starting a Continuous Measurement Task

For a continuous measurement, explicitly call the Start Task function/VI, perform the desired read operations, and call the Stop Task function/VI to stop the continuous measurement. When you perform a read operation in a loop—regardless if the read operation performs a single-sample, on-demand read, or a multiple-sample, hardware-timed read—call the Start Task function/VI before entering the loop and call the Stop Task function/VI after leaving the loop.

Starting an Analog Output Task

The behavior of the Write function/VI is more complicated. Calling the Write function/VI always results in the task transitioning to at least the <u>Committed state</u>. Whether the task transitions to the <u>Running state</u> depends on the value of the **Auto-Start** parameter.

For single-sample write operation, call a single-sample version of the Write function/VI. This call implicitly starts the task, writes the single sample, and stops the task. For a multiple-sample, on-demand write operation, call the Write function/VI, but also set the **Auto-Start** parameter to True, which by default is set to False. This call implicitly starts the task, writes the multiple samples, and stops the task.

For a multiple-sample, hardware-timed write operation, first call the Write function/VI to write the samples to generate, explicitly call the Start Task function/VI, wait for the samples to be generated by calling the Wait Until Done function/VI, and then explicitly call the Stop Task function/VI.

If you attempt to perform a hardware-timed generation with the **Auto-Start** parameter of the Write function/VI set to True either because you explicitly set it to true or because you are using a single-sample Write function/VI, the operation might fail because the samples that you write are not transferred to the device in time to generate the waveform. As a result, when performing hardware-timed generations, always write at least part of the waveform to generate before starting the task.

Improving Performance with the Start Task function/VI

There are other situations in which you should explicitly call the DAQmx Start Task function/VI and the DAQmx Stop Task function/VI, even though you are not required to do so. When you call the Read function/VI or the Write function/VI in a loop, you can significantly improve performance if you explicitly call the Start Task function/VI before entering the loop and call the Stop Task function/VI after exiting the loop. Without explicitly calling the Start Task function/VI before entering the loop, the task must implicitly transition from its current state to the Running state before performing the read or write operation. After the read or write operation is complete, the task must implicitly transition from the Running state back to its previous state. These implicit state transitions occur for every iteration of the loop, which is inefficient.

Aborting a Task

Several conditions cause a task to abort:

- To explicitly abort a task, call the Control Task function/VI with the **Action** parameter set to Abort. In general, aborting a task is not a normal operation. It is intended for exceptional situations.
- In LabVIEW, you also can abort a task by clicking the **Abort Execution** button. Doing so results in all tasks created in that VI hierarchy to be aborted and then cleared.
- If you remove a device from the system, all tasks currently using the resources of that device are aborted.
- If you call the Reset Device function/VI to restore a device to its initial configuration, all tasks currently utilizing the resources of that device are aborted.

When a task is aborted, it is returned to the <u>Verified state</u>. If the task is running, it is stopped as soon as possible and is then unreserved. After a task has been aborted, you can continue to use the task. However, you might need to transition the task back to its previous state before continuing the specified operation.

Using Is Task Done

You can use the Is Task Done function/VI for applications in which you need to <u>monitor the progress of a task</u> running in one section of your application from another section of your application.

In general, use the Is Task Done function/VI with continuous measurements and generations when you are not actively reading or writing samples but want to monitor for errors.

Using Wait Until Done

You might need to call the Wait Until Done function/VI to ensure that the specified operation is complete before you stop the task.

The most common example is a finite generation. If you start a task that performs a finite generation and then immediately stop the task, the generation <u>probably has not completed</u> when you stop the task. As a result, the generation does not complete as expected. To ensure that the finite generation completes as expected, call the Wait Until Done function/VI before stopping the task. After the Wait Until Done function/VI executes, the finite generation has been completed, and you can stop the task.

In general, use the Wait Until Task Done function/VI with finite measurements and generations.

When Is A Task Done?

If the measurement or generation is finite, the task is done when you acquire or generate the final sample. If the measurement or generation is continuous (including on-demand timing), the task is not done until you call the Stop Task function/VI. In addition, the task is done if a fatal error is generated while performing the measurement or generation, or you abort the measurement or generation. Check for errors and warnings to verify the task completed successfully.

Task State Model

NI-DAQmx uses a task state model to improve ease of use and speed up driver performance.

The task state model consists of five states—<u>Unverified</u>, <u>Verified</u>, <u>Reserved</u>, <u>Committed</u>, and <u>Running</u>. You call the Start Task function/VI, Stop Task function/VI, and Control Task function/VI to transition the task from one state to another. The task state model is very flexible. You can choose to interact with as little or as much of the task state model as your application requires.



To find out more about each of these states, refer to the following topics:

- <u>Unverified</u>
- <u>Verified</u>
- <u>Reserved</u>
- <u>Committed</u>
- Running

If you explicitly invoke a state transition that has already occurred, it is not repeated and an error is not returned. For example, if the task has already reserved its resources and, therefore, is in the Reserved state, calling the Control Task function/VI with the **Action** parameter set to Reserve does not reserve the resources again.

Unverified State

When a task is created or loaded, either explicitly or implicitly, it is in the Unverified state. In this state, you configure the timing, triggering, and channel attributes/properties of the task.

Verified State

NI-DAQmx checks the timing, triggering, and channel attributes/properties for correctness when the task transitions from the Unverified to the Verified state. You can explicitly perform this transition by calling the Control Task function/VI with **Action** set to Verify. While NI-DAQmx detects and verifies some invalid values for attributes/properties immediately when you set the attribute/property, NI-DAQmx cannot verify other values immediately because they depend on other attributes/properties and the devices being used. NI-DAQmx checks the value of these attributes/properties during the verify transition and reports any invalid values at that time. If NI-DAQmx finds no invalid values, the task is successfully verified and transitions to the Verified state. Otherwise, it remains in the Unverified state.

In certain cases, NI-DAQmx will <u>coerce</u> the values of attributes/properties when successfully verifying a task rather than generating an error. This is done when the value set on the attribute/property cannot be met exactly as specified and coercing it to a legal value has little functional impact on the task.

Reserved State

The resources a task uses to perform the specified operation are acquired exclusively when the task transitions from the Verified state to the Reserved state. These resources can be clocks or channels on a device, trigger lines on a PXI chassis, or buffer memory in the computer. Reserving these resources prevents other tasks from using these resources, which interferes with this task performing the specified operation. You can explicitly perform this transition by calling the Control Task function/VI with **Action** set to Reserve. This transition fails if some task resources are currently reserved by another task. If the task can gain access to all the resources it uses, the task is successfully reserved and transitions to the Reserved state. Otherwise, it remains in the Verified state.

Committed State

NI-DAQmx programs some of the settings for the resources when the task is committed. These settings might be the rate of a clock or the input limits of a channel on a device, the direction of a trigger line on a PXI chassis, or the size of the buffer memory in the computer. Other settings, such as the sample counter, cannot be programmed when the task is committed because they need to be programmed every time the task is started. When a task is committed, it transitions from the Reserved state to the Committed state. You can explicitly perform this transition by invoking the Control Task function/VI with **Action** set to Commit. In general, the commit transition should not fail. If it does, it is an exceptional condition and the task remains in the Reserved state. If the settings for the resources used by the task are programmed, the task is successfully committed and transitions to the Committed state.

Running State

When the task begins to perform the specified operation, the task transitions from the Committed state to the Running state. You can explicitly perform this transition by invoking the Start Task function/VI. Notice that starting a task does not necessarily start acquiring samples or generating a waveform. You might have specified the timing and triggering attributes/properties such that a sample is not acquired until you call the Read function/VI or a waveform is not generated until a trigger is detected. In general, the start transition does not fail. If it does, it is an exceptional condition, and the task remains in the Committed state. If the task begins to perform the specified operation, the task is successfully started and transitions to the Running state.

Running to Committed State

The task ceases to perform the specified operation when the task transitions from the Running state to the Committed state. To explicitly perform this transition, call the Stop Task function/VI. Notice that you might have specified the timing and triggering attributes/properties such that all the samples are acquired before this transition occurs. For output operations, the last value written will typically continue to be generated after the task is stopped. In this situation, despite the fact that no additional samples are acquired, the task is still in the Running state until this transition occurs. In general, the stop transition does not fail. If it does, it is an exceptional condition, and the task is returned to the Reserved state. If the task is stopped, the task successfully transitions back to the Committed state.

Committed to Verified State

When the task resources that perform the specified operation are released, the task transitions from the Committed state to the Verified state. These resources may be clocks or channels on a device, trigger lines on a PXI chassis, or buffer memory in the computer. To explicitly perform this transition, call the Control Task function/VI with **Action** set to Unreserve. After the task releases all of its resources, it successfully transitions back to the Verified state.

Explicit Versus Implicit State Transitions

When should you perform explicit state transitions, and when should you rely on the task to perform implicit state transitions? The answer depends on your application. The following list identifies instances in which you should use explicit state transitions:

- **Verify**—If in your application users interactively configure a task by setting various channel, timing, and triggering attributes/properties, explicitly verify the task occasionally to inform the users if they have set an attribute/property to an invalid value.
- **Reserve**—If the following is true, explicitly reserve a task: your application contains many different tasks that use the same set of resources, one of these tasks repeatedly performs its operation, and you want to ensure that none of the other tasks acquires these resources after the task begins its sequence of operations.

Reserving the task exclusively acquires the resources that the task uses, ensuring that other tasks cannot acquire these resources. For example, if your application contains two tasks that each perform a sequence of measurements and you want to ensure that each sequence is completed before the other sequence begins, you can explicitly reserve each task before it begins its sequence of measurements.

• **Commit**—If your application performs multiple measurements or generations by repeatedly starting and stopping a task, explicitly commit a task. Committing the task exclusively acquires the resources that the task uses and programs some of the settings for these resources. By explicitly committing the task, these operations are performed once, not each time the task is started, which can considerably decrease the time needed to start your task. For example, if your application repeatedly performs finite, hardware-timed measurements, the time required to start the task can dramatically decrease if you explicitly commit the task before repeatedly performing these measurements. Explicitly committing a task also is required if you need to perform additional read operations of the samples acquired by the task after stopping the

task. For more information, refer to <u>Using the Start Task</u> <u>Function/VI</u>

 Start—If your application repeatedly performs read or write operations, explicitly start a task. Starting the task reserves the resources that the task uses, programs some of the settings for these resources, and begins to perform the specified operation. By explicitly starting the task, these operations are performed once, not each time the read or write operation is performed. This process can considerably decrease the time required to perform each read or write operation. For example, if your application repeatedly performs single-sample, software-timed read operations, the time required for each read operation can dramatically decrease if you explicitly start the task before repeatedly performing these read operations.

Implicit Task State Transitions

Although you can explicitly transition a task through each of its states as described in <u>Task State Model</u>, you rarely need this level of detailed control. Two scenarios exist in which a task is implicitly transitioned from one state to another:

- Moving the task through multiple states at the same time
- Operations that require state transitions

Task Moves Through Multiple States at the Same Time

Some state transitions require the task to move through one or more states to reach the specified state. For example, if the task is in the Unverified state, and you call the Control Task function/VI, setting **Action** to Reserve, the task is verified and reserved. The task transitions from the Unverified state to the Verified state and to the Reserved state. In most applications, it is not helpful to explicitly transition the task to each state. Instead, invoke only those transitions that are necessary, and the task implicitly handles the rest.

Operations That Require State Transitions

You implicitly transition the task to a new state when you perform an operation that requires that the task be in a specific state and it is not. If this occurs, the task is implicitly transitioned to the required state. Some operations that require state transitions include the following:

- Querying the value of an attribute/property implicitly verifies the task. This verification is required to return accurate coerced values of attributes/properties. Because the coerced value of a attribute/property often depends on the values of other attributes/properties, the task as a whole must be verified to calculate the value. Because the task might be implicitly verified when you query the value of an attribute/property, NI-DAQmx may return an error specifying that the value of attribute/property is invalid.
- Calling the Read function/VI implicitly commits the task if the task is not already committed. If the value of the DAQmx Read **Auto Start** attribute/property is True and the task has not been started, the task also is implicitly started. For more information regarding the auto-start behavior of read operations, refer to <u>Using the Start</u> <u>Task Function/VI</u>.
- Calling the Write function/VI commits the task. If the value of the **Auto-Start** parameter is True, the task also is started. For more information regarding the auto-start behavior of write operations, refer to <u>Using the Start Task Function/VI</u>.

For example, if the task is in the Reserved state, the value of the DAQmx Read **Auto Start** attribute/property is True, and you call the Read function/VI, the task is implicitly committed and started. The task transitions from the Reserved state to the Committed state and to the Running state before the read operation is performed.

In some applications, it is not necessary to explicitly transition the task to any state. Instead, invoke the desired operation and the task implicitly handles everything else.

Transitioning the State Backwards

When a task is implicitly transitioned backwards, it returns to the state of the task prior to the last operation that resulted in a forward state transition. For example, if the task was in the Verified state and you called the Start Task function/VI to start the task, the task is reserved, committed, and started, transitioning to the Reserved state and to the Committed state before transitioning to the Running state. When you invoke the Stop Task function/VI, the task is not just stopped and transitioned from the Running state to the Committed state. If this were the case, the result is unexpected because the task still has its resources reserved despite the fact that you never explicitly reserved them. Instead, the task is stopped, uncommitted, and unreserved, returning to the Verified state, its state immediately before you performed the last operation that resulted in the state transition, calling the Start Task function/VI.

As another example, suppose the task is in the Reserved state, and you call the Read function/VI to perform a finite measurement. This results in the task implicitly transitioning from the Reserved state to the Committed state and then to the Running state before performing the read operation. When the read operation completes, the task does not remain in the running state. If this were the case, the result is unexpected behavior, because you need to stop the task and unreserve its resources despite the fact you never explicitly reserved the resources or started the task. Instead, after the finite read operation completes, the task is implicitly transitioned from the Running state to the Committed state to the Reserved state. This results in the task returning to the state before you performed the read operation.

Keep in mind that setting the value of a channel, timing, or triggering attribute/property does not implicitly transition the task back to the Unverified state. Instead, the task remains in its current state and is implicitly verified when the next state transition occurs. For example, if the task is in the Reserved state and you set the value of timing attribute/property, the task remains in the Reserved state. The next time the task, either implicitly or explicitly, is committed, the task is verified. Because the task is implicitly verified when the next state transition occurs, NI-DAQmx can return an error specifying that the value of attribute/property is invalid.

Creating Channels and Tasks with the DAQ Assistant

You can launch the DAQ Assistant from your NI application software or from MAX. The DAQ Assistant is a graphical interface for configuring channels, tasks, and scales.

After you launch the DAQ Assistant, follow the wizard instructions to create your new task or channel. When the wizard is done, you can configure measurement-specific settings, scaling, and, if necessary, timing and triggering.

LabVIEW

In LabVIEW, there are several ways to open the DAQ Assistant. A couple of common ones are the following:

- Drop the DAQ Assistant Express VI from the Express Input palette, as described in *Getting Started with LabVIEW*.
- Use the DAQmx Task Name control to open the DAQ Assistant. *Taking an NI-DAQmx Measurement in LabVIEW*, which is included in the *LabVIEW Help*, has step-by-step instructions on how to create a task from the DAQmx Task Name control and generate code from the task.

LabWindows/CVI

In LabWindows/CVI, select **Tools»Create/Edit DAQmx Tasks**. You also can launch the DAQ Assistant by clicking the **Task Name** control of the DAQmx LoadTask function panel and selecting **New Task**.

Measurement Studio

In Measurement Studio, open Visual Studio .NET and select **Project»Add New Item** to open the Add New Item dialog box. In the Categories pane, select **Measurement Studio»Assistants**. In the Templates pane, select **DAQmx Task Class**.

MAX

In MAX, right-click **Data Neighborhood**, and select **Create New** from the shortcut menu. Select **NI-DAQmx Task** or **NI-DAQmx Global Virtual Channel** in the **Create New** window, and click **Next**.

See Also

Choosing Whether to Use the API or the DAQ Assistant

Choosing Whether to Use the API or the DAQ Assistant

When creating a new application, you can choose to use DAQ Assistant or the API.

Advantages of Using the DAQ Assistant

- The DAQ Assistant requires no programming. You can configure channels, timing, triggering, and scales interactively.
- The DAQ Assistant can decrease development time. You can create a complete application in a matter of minutes.
- If you create your application using the DAQ Assistant and later need functionality that it doesn't expose, you can easily generated the equivalent API code from your DAQ Assistant task if you use an NI ADE such as LabVIEW, LabWindows/CVI, or Measurement Studio.

Advantages of Using the API

- The API contains advanced features not exposed by the DAQ Assistant.
- The API provides additional flexibility, allowing you to customize your application to suit your needs.
- The API gives you tighter control over the performance of your application.

Timing and Triggering

Timing and triggering are important in NI-DAQmx. The <u>clocks</u> section explains clocks and handshaking. The <u>triggering</u> section goes over the triggers—such as a Start Trigger and a Reference Trigger—and common trigger types—such as an analog edge trigger or a digital edge trigger.
Timing, Hardware Versus Software

You can use software timing or hardware timing to control when a signal is generated. With hardware timing, a digital signal, such as a clock on your device, controls the rate of generation. With software timing, the rate at which the samples are generated is determined by the software and operating system instead of by the measurement device. A hardware clock can run much faster than a software loop. A hardware clock is also more accurate than a software loop.

In NI-DAQmx, select hardware timing with the Sample Clock Timing function/VI or by setting the **Sample Timing Type** attribute/property to Sample Clock. If you do neither of these things, or you set the **Sample Timing Type** attribute/property to On Demand, you are selecting software timing.



Note Some devices do not support hardware timing. Refer to your device documentation if you are unsure if your device supports hardware timing.

Clocks

Periodic digital edges measure time and are called *clocks*. Clocks such as a sample timebase clock and the 20 MHz timebase clock mark the passing of time or are used to align other signals in time. Clocks usually do not cause actions in the sense that triggers do. The names of clocks usually do not refer to actions. The sample clock is a notable exception.

Clocks in NI-DAQmx

The following are some common clocks used by DAQ devices. Refer to your device documentation for all the clocks on your device.

- AI Convert Clock—The clock on a <u>multiplexed device</u> that directly causes ADC conversions. The default AI Convert Clock rate uses 10 µs of additional settling time between channels, compared to the fastest AI Convert Clock rate for the device. When the Sample Clock rate is too high to allow for 10 µs of additional settling time, the default AI Convert Clock rate uses as much settling time as is allowed by the Sample Clock rate. If there are multiple devices in the same task, the same amount of additional settling time is used for all devices in the task, even if their maximum AI Convert Clock rates differ.
- AI Convert Clock Timebase—The clock that is divided down to produce the AI convert clock.
- Al Sample Clock—The clock that controls the time interval between samples. Each time the sample clock ticks (produces a pulse), one sample per channel is acquired.
- Al Sample Clock Timebase—The clock used as the onboard clock source of the sample clock. When the source of the sample clock is set to the onboard clock, the Sample Clock Timebase is divided down to produce the sample clock. When the source of the Sample Clock Timebase is also the onboard clock, the master timebase is divided down to produce the Sample Clock Timebase.
- **AO Sample Clock**—The clock that controls the time interval between samples. Each time the sample clock ticks (and produces a pulse), one sample per channel is generated.
- AO Sample Clock Timebase—The onboard clock used as the source of the AO sample clock. The AO Sample Clock Timebase is divided down to produce the AO sample clock.
- **Counter Timebase**—The clock connected to the source terminal of a counter (Ctr0Source, for example).
- **Master Timebase**—An onboard clock used by other counters on the device. The master timebase is divided down to produce a slower clock or to measure elapsed time. This timebase is the

onboard clock used as the source of the AI Sample Clock timebase, the AO Sample Clock timebase, and the counter timebases, for example.

- **20 MHz Timebase**—The onboard clock source for the master timebase from which other timebases are derived, if the device does not support an 80 MHz Timebase. Otherwise, the clock produced by dividing the 80 MHz Timebase by 4.
- **80 MHz Timebase**—The onboard clock source for the master timebase from which other timebases are derived.
- **100 kHz Timebase**—The clock produced by dividing the 20 MHz Timebase by 200.
- $\overline{\mathbb{N}}$

Note M Series and C Series devices do not have a master timebase of an arbitrary frequency. These devices use the 20 MHz/80 MHz/100 kHz timebase directly.

The following diagram illustrates the M Series clocks that comprise analog input and analog output timing. The black circles in the diagram represent terminals.



The following diagram illustrates the C Series clocks that comprise analog input and analog output timing.



The following diagram illustrates the E Series clocks that comprise analog input and analog output timing. The black circles in the diagram represent terminals.



Trigger and Clock Distinction

The distinction between triggers and clocks is blurred when the digital edges used as a trigger are periodic. In such a case, a clock causes the device to perform an action. The sample clock is the primary example. The stimulus for the action of producing a sample is so often a clock that NI-DAQmx configures the sample clock instead of the sample trigger. The distinction is made clear when you consider the sample clock is in fact just one way of providing the source of a sample trigger.

Sample Timing Types

NI-DAQmx introduces the concept of a *sample timing type*. Each sample timing type is a different stimulus for triggering the action of producing a sample. When you select a Timing function/VI, you select your sample timing type. There also is an attribute/property for setting the following sample timing types:

- **Sample Clock**—A digital edge produces each sample. Nearly all devices have an onboard clock that is dedicated to producing these edges periodically. Even when the edges are not periodic, as they might be when the clock source is something other than the dedicated onboard clock, you still use sample clock timing. Sample clock timing is a type of <u>hardware timing</u>.
- **On Demand**—Every time the Read or Write function/VI executes, the device produces the requested samples as fast as possible. In this mode, the **Sample Quantity** attributes/properties are ignored. On-demand timing is a type of software timing.
- **Change Detection**—Change detection timing captures samples from digital physical channels when NI-DAQmx detects a change —a rising edge, a falling edge, or both rising and falling edges— on one or more digital lines or ports. Change detection timing reduces the digital data an application has to process.

One issue to be aware of with change detection on <u>some devices</u> is overflow. Overflow occurs when NI-DAQmx cannot read a sample prior to the next change detection event. The effect is that one or more samples can be missed.

Programmatically, you include the Change Detection Timing function/VI, specifying the physical channels for rising and falling edges on which to detect changes. You can query for an overflow by using the **Overflow** attribute/property in your application after the task starts.

- **Handshake**—The handshake sample timing type is used to acquire or generate digital data with the 8255 protocol. Many devices have an 8255 chip, and other devices emulate the 8255 protocol by default with the handshake timing type.
- **Burst Handshake**—Burst handshake timing acquires or generates digital data on the data lines with a clocked protocol.

This timing type involves three control signals: the <u>sample clock</u>, the <u>Pause Trigger</u>, and the <u>Ready for Transfer Event</u>. Data is transferred on each active sample clock edge if the peripheral device deasserts the Pause Trigger and the DAQ device asserts the Ready for Transfer Event.

There are separate Burst Handshake Timing functions/VIs based on whether you import or export a sample clock. Using the appropriate function/VI is important because there are timing restrictions (such as <u>setup and hold times</u>) when sharing a clock between the two devices.

• **Implicit**—The implicit sample timing type is used for acquiring period or frequency samples using counters. It is also used for generating pulses. This timing type is called *implicit* because the signal being measured is itself the timing signal or the timing is implicit in the rate of the generated pulse train.

Sample Clock

Your device uses a sample clock to control the rate at which samples are acquired and generated. This sample clock sets the time interval between samples. Each tick of this clock initiates the acquisition or generation of one sample per channel. In Traditional NI-DAQ (Legacy), the sample clock is called the scan clock or the scan interval counter. You also can connect an outside source as your clock. In software, you can specify the interval (how fast the clock acquires or generates signals) by specifying the sample rate, which is called the scan rate or update rate in Traditional NI-DAQ (Legacy). You can limit the sample rate by the signal conditioning you apply to the signals or the number of channels in your application. However, the number of channels affects your measurement only if you are sampling close to the maximum sample rate for your device.



Note Sample clock timing for digital I/O is not supported on all <u>devices</u>.

Handshaking

If you want to communicate with an external device using an exchange of signals to request and acknowledge each data transfer, use handshaking.

For example, you might want to acquire an image from a scanner. The process involves the following steps:

- 1. The scanner sends a pulse to your measurement device after it scans the image and is ready to transfer the data.
- 2. Your measurement device reads an 8-, 16-, or 32-bit digital sample.
- 3. Your measurement device then sends a pulse to the scanner to inform the scanner that the digital sample has been read.
- 4. The scanner sends out another pulse when the scanner is ready to send another digital sample.
- 5. After your measurement device receives this digital pulse, the device reads the sample.

This process repeats until all the samples are transferred.

Note Not all devices support handshaking. Refer to your device documentation to see if handshaking is supported on your device. For E Series devices, only those devices with more than eight digital lines—those devices that have an additional 8255 chip onboard—support handshaking.

Handshaking Signals (Devices with an Additional 8255 Chip Onboard)

Burst Handshaking Signals

For devices that support burst handshake timing, three signals are used:

- Pause Trigger (formerly called REQ)
- Ready for Transfer Event (formerly called ACK)
- sample clock

For digital input tasks, when the Pause Trigger signal is logic low and the Ready for Transfer Event is logic high, the samples are sent to the measurement device. For digital output tasks, when the Pause Trigger signal is logic low and the Ready for Transfer Event is logic high, the NI-DAQmx device sends the samples to a peripheral device. The sample clock, either onboard or external, controls the timing. Data is transferred or acquired on either the rising or falling edge of the sample clock.

The default terminals used for burst handshaking signals vary from device to device. Refer to <u>Burst Handshaking Timing Signal Defaults</u> for your device.

Handshaking Signals for Devices That Emulate the 8255 Protocol

Devices that emulate the 8255 protocol support two handshaking signals:

- Handshake Trigger—Also called Strobe Input (STB) and Acknowledge Input (ACK)
- Handshake Event—Also called Input Buffer Full (IBF) and Output Buffer Full (OBF)

For input tasks, when the Handshake Trigger signal is low, the samples are sent to the measurement device. After the samples have been sent, Handshake Event is high, which tells the peripheral device that the data has been read. For digital output, Handshake Event is low while the NI-DAQmx device sends the samples to a peripheral device. After the peripheral device receives the samples, it sends a low pulse back on the Handshake Trigger line. Refer to your device documentation to determine which digital ports you can configure for handshaking signals.

The default terminals used for handshaking signals vary from device to device. Refer to <u>Handshaking Timing Signal Defaults</u> for your device.

Handshaking Signals for 8255-Based Devices

8255-based devices that perform handshaking support four handshaking signals:

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

Use the STB and IBF signals for digital input operations and the OBF and ACK signals for digital output operations. When the STB line is low, the samples are sent to the measurement device. After the samples have been sent, IBF is high, which tells the peripheral device that the data has been read. For digital output, OBF is low while the software sends the samples to an peripheral device. After the peripheral device receives the samples, it sends a low pulse back on the ACK line. Refer to your device documentation to determine which digital ports you can configure for handshaking signals.

Digital Data on Multiple Ports

For 8255-based devices, the ports in the task affect which handshaking lines are used. Always use the handshaking lines associated with the highest order port in the task. For instance, if you want to group ports 1 and 2 into a single task, use the handshaking lines associated with port 2.

Connect all the STB lines together if you are grouping ports for digital input, as shown in the following figure. Connect only the IBF line of the highest order port in the task to the other device. No connection is needed for the IBF signals for the other ports.



(Highest Order Port)

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



(Highest Order Port)

When performing handshaking, some lines are automatically reserved for control purposes and are unavailable for use. The control lines used depend on the ports you are using and whether you are handshaking with input or output channels. The remaining lines in the port not used for control are still available for use. If you are transferring data across any line in a port in a handshaking task, the entire port is reserved for handshaking data and the remaining lines in the port are unavailable for use.

Hardware-Timed Single Point Sample Mode

In hardware-timed single point sample mode, samples are acquired or generated continuously using <u>hardware timing</u> and no buffer. You must use the sample clock or change detection timing types. No other timing types are supported.

Use hardware-timed single point sample mode if you need to know if a loop executes in a given amount of time, such as in a control application.

Because there is no buffer if you use hardware-timed single point sample mode, you should ensure that reads or writes execute fast enough to keep up with hardware timing. If a read or write executes late, it returns a warning.

Continuous Pulses (HW Timed Updates) is hardware-timed single point for counter output. Refer to <u>Hardware-Timed Counter Tasks</u> for more information.

Multiplexed Versus Simultaneous Sampling

S Series devices use simultaneous sampling. These devices have an ADC for each analog channel, which allows you to sample from all channels at the same time, as shown in the following figure.



Other devices, such as M Series and E Series devices, use multiplexed sampling. With this type of sampling, a single ADC is used for all analog input channels. These devices use both a <u>sample clock</u> and a <u>convert</u> <u>clock</u>. The sample clock initiates the acquisition of a sample from all channels in the scan list. The convert clock causes the ADC conversion for each individual channel. The following figure depicts a three-channel analog input task on a device that uses multiplexed sampling. Notice that, unlike S Series devices, the samples are not digitized simultaneously.

Sample Clock	
Convert Clock	Ch1 Ch2 Ch3

The convert clock must run faster than the sample clock to achieve the specified sample rate. For instance, if you specify a sample rate of 10 S/s for 8 analog input channels, the convert clock must run at least eight times the sample rate (80 Hz) to ensure that each channel is sampled 10 times a second. At faster sampling rates, you must also take settling time between channels into account.

C Series devices in a NI cDAQ-9172 chassis use both simultaneous and multiplexed sampling, where all devices in the chassis share the same sample clock. Devices, such as the NI 9215, with an ADC for each analog channel use simultaneous sampling. Devices with a single ADC sample in sequence, using multiplexed sampling. An example of such a device is the NI 9205.

Each multiplexed C Series device has a separate convert clock. The convert clock timing is based on the number of channels for that device in the task, not the total number of channels in the task. You can set the convert clock rate on a per-device basis using the **Active Devices** and **AI Convert Rate** attributes/properties on the DAQmx Timing property node.

The following figure depicts a ten-channel analog input task on two simultaneous sampling C Series devices and two multiplexed sampling C Series devices with different AI convert rates:



Setup and Hold Times

When a DAQ device samples a digital signal, the signal must remain stable for a period of time before and after the assertion of the clock edge used for timing. The amount of time before the assertion of the clock is called the *setup time*. The amount of time after the assertion of the clock edge is called the *hold time*. Refer to your device documentation for minimum setup and hold times.

Simultaneous Analog Output On-Demand Timing

Typically, when you use software timing to output samples on multiple AO channels, NI-DAQmx writes a sample to the first DAC, and the sample is generated. Then, NI-DAQmx writes a sample to the second DAC, and that sample is generated, and so on. However, with the simultaneous single-point on-demand timing, all of the data is generated at the same time after NI-DAQmx writes to each DAC. You set this timing with the **Simultaneous Analog Output Enable** attribute/property.

Timing Response Modes

Digital I/O and DAQ devices typically use the single-cycle timing response mode, meaning the device responds to an external signal by the next active sample clock edge.

Devices that support the pipelined timing response mode, such as the NI PCIe-6536 and NI PCIe-6537, can respond to an external signal a few sample clock edges later. This mode uses a source-synchronous clock scheme, which simultaneously returns the clock and data to the acquiring device. With a source-synchronous data transfer, you can acquire and generate data at much higher rates than with single-cycle timing response mode.

With the pipelined timing response mode, you can configure external sample clocks, but the sample clock must be free-running and started before the task commits. If you export the sample clock, the export occurs during a task commit. As with other events, when the task uncommits, the signal remains exported.

Triggering

When a device controlled by NI-DAQmx does something, it performs an *action*. Two very common actions are producing a sample and starting a waveform acquisition. Every NI-DAQmx action needs a stimulus or cause. When the stimulus occurs, the action is performed. Causes for actions are called *triggers*. Triggers are named after the actions they cause:

- Advance Trigger
- Expiration Trigger
- Handshake Trigger
- Pause Trigger
- <u>Reference Trigger</u>
- <u>Start Trigger</u>
- Arm Start Trigger

In addition to specifying the action you want a trigger to cause, you must select the type of trigger to use, which determines how the trigger is produced.

Advance Trigger

An Advance Trigger causes a switch device to execute the next entry in its instruction (scan) list. You can configure this trigger to occur on a digital edge or when the Send Software Trigger function/VI runs.

Arm Start Trigger

When you configure an Arm Start Trigger, a counter task does not respond to any Start Triggers until after the Arm Start Trigger occurs. You can configure this trigger to occur on a digital edge. The Arm Start Trigger is separate from a Start Trigger and is typically used in advanced counter/timer applications. You might use an Arm Start Trigger to synchronize multiple tasks, such as counting edges and pulse generation. The Start Trigger then would be used to start the acquisition or generation.

Expiration Trigger

An Expiration Trigger expires a watchdog task. You can use this trigger instead of the <u>watchdog timer</u> to signal an expiration. You can configure this trigger to occur on a digital edge.

Handshake Trigger

A Handshake Trigger is a control signal from a peripheral device. The peripheral device asserts the Handshake Trigger to signal to the DAQ device that it has acquired a sample (for output tasks) or generated a sample (for input tasks). For input tasks, the DAQ device latches data, by default, at the trigger position specified by the **sample when** attribute/property when the peripheral device deasserts the Handshake Trigger.

Pause Trigger

With sample clock timing or burst handshake timing, the Pause Trigger pauses an ongoing acquisition or generation. Deasserting this trigger resumes an acquisition or generation. Depending on your device, there are some <u>additional issues</u> you need to remember.

Reference Trigger

A Reference Trigger establishes the reference point in a set of input samples. You can configure this trigger to occur on a digital edge, a digital pattern, an analog edge, or when an analog signal enters or leaves a window. Data acquired up to the reference point is pretrigger data. Data acquired after this reference point is posttrigger data.



Start Trigger

A Start Trigger begins an acquisition or generation. You can configure this trigger to occur on a digital edge, a digital pattern, an analog edge, or when an analog signal enters or leaves a window.

Trigger Types

In addition to specifying the action you want a trigger to cause, you must select the type of trigger to use, which determines how the trigger is produced. If you need to trigger off an analog signal, use an <u>analog edge</u> trigger or an <u>analog window trigger</u>. If the trigger signal is digital, choose a <u>digital edge trigger</u> with the source typically being one of the PFI pins.

- Analog Edge
- Analog Level
- <u>Analog Window</u>
- Digital Edge
- Digital Level
- Digital Pattern
- Software

Analog Edge Triggering

For analog edge triggering, you configure the measurement device to look for a certain signal level and slope (either rising or falling). After the device identifies the trigger condition, the device performs the specified action associated with the trigger, such as starting the measurement or marking which sample was acquired when the trigger occurred. You connect analog trigger signals to any analog input channel or terminal capable of accepting analog signals. Refer to the <u>device-specific analog</u> <u>triggering considerations</u> for your device for additional information.

In the following figure, the trigger is set to capture data for a rising edge signal when the signal reaches 3.2.



Hysteresis

Hysteresis adds a window above or below the trigger level and often is used to reduce false triggering due to noise or jitter in the signal. When using hysteresis with a rising slope, the trigger asserts when the signal starts below **level** (or **threshold level**) minus hysteresis and then crosses above **level**. The trigger deasserts when the signal crosses below **level** minus **hysteresis**.

For example, if you add a **hysteresis** of 1 to the previous example, which used a **level** of 3.2, the signal must start at or drop below 2.2 for triggering to occur. The trigger then asserts as the signal rises above 3.2 and deasserts when it falls below 2.2.



When using hysteresis with a falling slope, the trigger asserts when the signal starts above **level** (or **threshold level**) plus hysteresis and then crosses below **level**. The trigger deasserts when the signal crosses above **level** plus **hysteresis**. If you instead trigger on a falling edge at 3.2 with a hysteresis of 1, the signal must start at or rise above 4.2 for triggering to occur. The trigger will then assert as the signal falls below 3.2 and deassert when it rises above 4.2.



See Also

Device-Specific Analog Triggering Considerations

Analog Level Triggering

An analog level trigger is similar to an analog edge trigger. With both trigger types, you specify the edge—rising or falling—and the trigger level. With an analog edge trigger, you are interested in the point at which the trigger condition is met. With an analog level trigger, on the other hand, you are interested in the *duration* that the signal remains above or below the trigger level. An analog level trigger is typically used with a Pause Trigger. The Pause Trigger asserts or deasserts when the trigger condition is met. In the following illustration, a trigger asserts when it drops below it. The deassertion of the trigger could correspond to a Pause Trigger.



Analog Window Triggering

A window trigger occurs when an analog signal either passes into (enters) or passes out of (leaves) a window defined by two voltage levels. Specify the levels by setting the window top value and the window bottom value. The following image demonstrates a trigger that acquires data when the signal enters the window.



The following image demonstrates a trigger that acquires data when the signal leaves the window.



See Also

Device-Specific Analog Triggering Considerations
Digital Edge Triggering

A digital trigger is usually a TTL signal with two discrete levels: a high and a low level. When the signal moves from high to low or from low to high, a digital edge is created. There are two types of edges: rising and falling. You can produce Start or Reference Triggers from the rising or falling edge of your digital signal.

In the following figure, the acquisition begins after the falling edge of the digital trigger signal. Usually, digital trigger signals are connected to PFI pins on your measurement device.



Digital Pattern Triggering

For digital pattern triggering, you configure the device to detect a specific digital pattern on specific physical channels. After detecting this condition, the device performs the action associated with the trigger, such as starting the task or marking which sample was acquired when the trigger occurred.

The digital pattern is specified using the following characters:

- X: ignore the physical channel
- 0: Match on a logic low level on the physical channel
- 1: Match on a logic high level on the physical channel
- R: Match on rising edge on the physical channel
- E: Match on either rising or falling edge on the physical channel
- F: Match on falling edge on the physical channel

For instance, if you specify a pattern of "X11100" and a source of "dev1/line0:4,dev1/line6," the pattern match occurs when physical channels "dev1/line1," "dev1/line2," and "dev1/line3" are logic high and when physical channels "dev1/line4" and "dev1/line6" are logic low. "dev1/line0" is ignored.



For pattern triggers on ports, the pattern match occurs in reverse order. For instance, if you specify a pattern of "11000000" and a source of "dev1/port0," the pattern match occurs when physical channels "dev1/line0" and "dev1/line1" are logic high and the other six lines are logic low.

Events

Triggers and clocks are input signals. Exportable triggers and clocks, such as the sample clock, also can be output signals. Output signals that do not have a trigger or clock counterpart are called *events*. Events are emitted to signify a device state change, the arrival of a certain kind of sample, the production of a certain amount of samples, or the passage of time.

NI-DAQmx includes the following events:

- Advance Complete Event—A signal emitted by a switch when it has finished executing an instruction in its scan list.
- AI Hold Complete Event—A signal emitted by a multiplexed analog input circuit when the analog signal at the physical channel being measured has been latched or held. The AI Hold Complete Event is designed to signal an external multiplexer to switch to the next channel. This signal was previously known as SCANCLK, which is the legacy name of the external terminal where this signal can be emitted.
- **Change Detection Event**—A signal a DIO device generates after it detects a change—a rising edge, a falling edge, or both rising and falling edges—on the data lines.
- **Counter Output Event**—A signal produced by a counter when it reaches terminal count.
- Handshake Event—A signal generated by a DAQ device that is used for handshaking. The assertion and deassertion times for this event are configurable within a handshaking cycle for some devices. For these devices, the default configuration is to mimic the 8255 protocol, which means that for input tasks, this event asserts after the device has space available in its FIFO; for output tasks, it asserts after valid data has been driven on the data lines; and in both input and output tasks, the event deasserts after the <u>Handshake Trigger</u> has been asserted.
- **Ready For Start Event**—A signal produced when a device is ready to accept a <u>Start Trigger</u>.
- **Ready For Transfer Event**—A signal sent to the peripheral device that signals that the DAQ device is ready for a transfer. For burst handshake output tasks, this means that the data is on

the data lines. For input tasks, this means that there is space available in the device FIFO. This event is used by <u>devices</u> that support burst handshake timing.

- Sample Complete Event—A signal produced when the device acquires a sample from every channel in a task.
- Watchdog Timer Expired Event—A signal produced when a watchdog timer expires. Watchdog timers are hardware features that can detect failures in the software controlling the device.
- Note The Sample Complete Event is not exportable.

Exported Signal Behaviors

You can export clocks, triggers, and events. The exported signal can exhibit one of three behaviors. It can rapidly change from its current state and then back again (pulse), change from its current state and remain at that state (toggle), or change from its current state and remain at that new state for a period of time determined by the configuration of the task before reverting back to the initial state (level). Most exported signals have pulse behavior, but some signals have programmable output behavior. For example, the Counter Output Event supports toggle as well as pulse behaviors. The Sample Clock supports pulse and level behaviors. You specify the behavior through the **Output Behavior** attribute/property for the exported signal.

Most exported signals exhibit the pulse behavior. When the event occurs, a finite pulse is generated. The pulse width of some exported triggers and events is configurable. The polarity of a signal exported as a pulse is also sometimes configurable. In the following illustration, the polarity is set to active high, meaning the initial state change of the signal is from low to high. When an event is exported as a pulse, each time the event occurs, the exported signal pulses.

When an event is exported as a toggle, each time the event occurs the exported signal changes state just once and remains at its new state until the next occurrence of the event. You can also set the initial state. In the following illustration, the initial state is set to high. The Counter Output Event is an example of a signal that can toggle.

For level behavior, the signal changes state and remains at that state for a period of time that is dependent on some configurable aspect of your task. If you are exporting the Sample Clock, the exported signal goes high at the beginning of the sample and goes low when the last AI Convert Clock pulse begins, as shown in the illustration.





Note On some devices, the exported signal can go low at the

beginning of the sample and then high when the last AI Convert Clock pulse begins. Refer to your device documentation for additional information.

Software Events

Software events provide an asynchronous notification mechanism for a set of DAQ events. Unlike hardware <u>events</u>, software events do not require you to use a thread to wait until data is available. Using event-based programming, you can write an application that continues to perform work while waiting for data without resorting to developing a multi-threaded application.

NI-DAQmx includes the following software events:

- Every N Samples Acquired Into Buffer Event—Occurs when the user-defined number of samples is written from the device to the PC <u>buffer</u>. This event works only with devices that support buffered tasks.
- Note The value you set for this event must be evenly divisible into the <u>buffer size</u> if you are using <u>DMA</u> as your data transfer mechanism. For instance, if the buffer size is 1,000 samples, specifying 102 for this software event generates an error. Specifying 100, however, would not generate an error. If you are using IRQ as the data transfer method, the value does not need to be evenly divisible. With IRQ, however, the **Data Transfer Request Condition** attribute/property can affect when this software event occurs.
 - Every N Samples Transferred From Buffer Event—Occurs when the user-defined number of samples is written from the PC <u>buffer</u> to the device. This event works only with devices that support buffered tasks.
- Note The value you set for this event must be evenly divisible into the <u>buffer size</u> if you are using <u>DMA</u> as your data transfer mechanism. For instance, if the buffer size is 1,000 samples, specifying 102 for this software event generates an error. Specifying 100, however, would not generate an error. If you are using IRQ as the data transfer method, the value does not need to be evenly divisible. With IRQ, however, the **Data Transfer Request Condition** attribute/property can affect when this software event occurs.
 - Done Event—Occurs when the task completes execution or

when an error causes the task to finish. Recoverable errors that do not cause the task to finish do not cause this event to fire. Calling the Stop Task function/VI to complete execution similarly does not cause this event to fire.

• **Signal Event**—Occurs when the specified hardware signal occurs. Supported signals include the <u>counter output event</u>, <u>change detection event</u>, <u>sample complete event</u>, and <u>the sample clock</u>.

Reading and Writing Data

This section covers <u>buffering</u> and <u>selecting data formats and</u> <u>organization</u>.

Selecting Read and Write Data Format and Organization

NI-DAQmx provides multiple VIs and functions for reading and writing data. In many cases, you can use multiple options. This section outlines the options and provides some guidelines to follow to select the best option. Some data formats and organizations are not supported in all ADEs.

The read and write VIs have two major selection criteria: <u>data format</u> and <u>data organization</u>. Data format deals with the type of the data that is returned. For example, counter reads can return integers or floating-point formats. The second category, data organization, deals with the structure the data is returned in. For example, analog reads have a variety of array and scalar organizations.

Data Formats in NI-DAQmx

Data format deals with the type of the data that is read or written.

Analog Channel Data Formats

Waveform

The waveform data format includes the channel name, timing, and unit information with the actual 64-bit scaled floating-point data. Your ADE provides a mechanism for extracting and setting individual parts of the waveform.

For input tasks, you can use the additional information for a variety of purposes. For example, you can update graphs to show the timing information and include labels with the channel names. Analysis routines can use the timing information for calculations such as FFTs. Because there is overhead associated with including this additional information, NI-DAQmx allows you to configure the information you want to include.

For output tasks, the timing information is the primary field that is useful. A library that generates a waveform can include timing information that sets up the timing for your output task.

When reading data, the waveform data includes the time when the first sample in the waveform was acquired, t0, and the amount of time that elapsed between each sample, dt. However, there are <u>limitations on</u> <u>these two values</u>.

64-Bit Floating-Point Numbers

The 64-bit floating-point number format allows you to read or write scaled data with no additional information. Use this format to work with scaled data that requires higher performance than the waveform format provides. You might also use this format because it is a better match for the libraries you plan to use.

Unsigned and Signed Integers

The unsigned and signed integer format reads or writes data in the native format of the device. Use this format for maximum performance. The tradeoff is that your application has to understand how to interpret and manipulate data that is not in engineering units.

Digital Channel Data Formats

Waveform

The waveform data format includes the channel name and timing information with the actual data represented in a dedicated digital format. Your ADE provides a mechanism for extracting and setting individual parts of the waveform.

The dedicated digital format represents digital data similar to logic analyzers and digital simulation tools. Each channel has no limits on the number of lines. In addition, the digital format allows for additional states beyond basic 1s and 0s. The ADE can take advantage of this format by tailoring data and graph displays for the digital data.

For input tasks, you can use the additional information for a variety of purposes. For example, you can update graphs to show the timing information and include labels with the channel names. Because there is overhead associated with including this additional information, NI-DAQmx allows you to configure the information you want to include.

For output tasks, the timing information is the primary field that is useful. A waveform generated by a library may include timing information that you can use to set up the timing for your output task.

When reading data, the waveform data includes the time when the first sample in the waveform was acquired, t0, and the amount of time that elapsed between each sample, dt. However, there are <u>limitations on these two values</u>.

Line Format (Boolean)

The line format represents each line within a channel as a single Boolean value (a single byte). The states of the data are limited to 1s (true) and 0s (false). Line formats are only provided for single sample reads and writes.

Use the line format when it is convenient for manipulating or displaying the digital data. A typical application is controlling or reading back relay states. For high-speed digital applications, you should generally not use the line format.

Port Format (Integer)

The port format matches the native format of digital devices that can represent only two digital states and organize individual lines into

collections known as ports. For more information, refer to <u>Digital Data</u><u>Integer Format</u>

The port format is the most efficient in terms of space, as it requires only a bit of memory per line. In addition, the port format is often the most efficient in time as it matches the native format of many devices.

The largest integer supported is 32 bits; therefore, you can read and write digital channels with no more than 32 lines when using the port format.

Counter Channel Data Formats

64-Bit Floating-Point Numbers

The 64-bit floating-point number format reads scaled data. This format is best when you want to work with data in engineering units.

Unsigned Integers

The unsigned integer format reads data in the native format of the device. Use this format for maximum performance. The tradeoff is that your application will have to understand how to interpret and manipulate data that is not in engineering units.

Raw Data Formats

The <u>raw data</u> format is defined by the native data format of the device.

Data Organization

The number of channels and the number of samples being read generally affect data organization. For example, if 100 samples are read for eight analog channels using 64-bit floating-point format, a two-dimensional array is used with one index selecting the channel and the second index selecting the sample. On the other hand, a simple floating-point scalar value is sufficient to read one sample for one analog channel using the 64-bit floating-point format. In general, the data organization for a particular read or write call is the simplest reasonable format that can handle the number of channels and samples requested.

There are often multiple legal data organizations to choose from. The main tradeoff to consider for data organization is difficulty in manipulation of the data. You can use data organizations that can handle multiple channels and multiple samples, but they are generally the most complicated to manipulate.

Performance is not significantly different for equivalent operations using read or writes with different data organizations.

Waveform Data Organization

A waveform can contain one or more samples.

1D Waveform Array Data Organization

The single dimension of a waveform array selects the channel. Each waveform can contain multiple samples, so a second dimension is not required.

Scalar Data Organization

Use scalars when you read or write a single sample on a single channel. Scalar data is easy to manipulate. It is a good match when data is read and or written to individual channels as needed.

Scalar data is generally not a good match for high-speed multiple sample applications.

Scalar data also is not a good choice if multiple channels are acquired or generated simultaneously. Using a multiple channel organization is easier and in the case of output operations is actually a requirement.

Array Data Organization

Array formats allow you to read or write data for multiple channels and/or multiple samples at the same time. If you acquire or generate on multiple channels simultaneously, reading and writing them at the same time is easiest. Reading and writing multiple samples in one call is more efficient than reading and writing samples one at a time.

Raw Data Organization

The <u>raw data</u> organization is defined by the native organization of the device.

Digital Data (Integer Format)

You typically use an integer format to read or write entire ports. In integer format, each digital channel you read or write must fit into one integer. For example, if the largest channel in a task consists of one 8-line port, you can use the 8-bit, 16-bit, or 32-bit format. If you have more than one 8-line port or a port with more than eight lines in any channel within a task, you must use the 32-bit format.

Each byte in the integer maps to a port in the channel, in the order in which you added the ports to the channel. The least significant byte maps to the first port added to the channel, with all unused bytes zeroed out. Therefore, if a channel contains two 8-line ports, port0 and port1, and you added port0 to the channel before port1, the channel uses a 32-bit representation:

```
unused unused port1 port0
```

Within a particular byte, each bit in the integer maps to a line in the corresponding port. NI-DAQmx orders the bits by line number, with the least significant bit mapping to the lowest line number. Therefore, with these values assigned to the lines in the channel, you might get the following:

port0/line0	0
port0/line1	0
port0/line2	1
port0/line3	0
port0/line4	1
port0/line5	1
port0/line6	0
port0/line7	1
port1/line0	1
port1/line1	1
port1/line2	1
port1/line3	0
port1/line4	1

port1/line5 0 port1/line6 0 port1/line7 1

The 32-bit binary representation of the channel is the following:

000000000000000000010010111010100100

with an integer value of 38836.

If you specify only certain lines in a port to read or write, the full length of the integer is still used, but all unused bits are zeroed out. Therefore, the following lines and values:

port0/line0	1
port0/line3	1

yield the following 8-bit representation:

00001001

with an integer value of 9.

Interleaving

Interleaved samples prioritize samples before channels, such that the array lists the first sample from every channel in the task, then the second sample from every channel, up to the last sample from every channel.

Channel 0—Sample 1
Channel 1—Sample 1
Channel 2—Sample 1
Channel 0—Sample 2
Channel 1—Sample 2
Channel 2—Sample 2
Channel 0—Sample N
Channel 1—Sample N
Channel 2—Sample N

Non-interleaved samples prioritize channels before samples, such that the array lists all samples from the first channel in the task, then all samples from the second channel, up to all samples from the last channel.

```
Channel 0—Sample 1
Channel 0—Sample 2
...
Channel 0—Sample N
Channel 1—Sample 1
Channel 1—Sample 2
...
Channel 1—Sample N
Channel 2—Sample 1
Channel 2—Sample 2
...
```

Channel 2—Sample N

Raw Data

Raw data is in the native format and *organization* of the device, read directly from the device or buffer without scaling or reordering. The native format of a device can be an 8-, 16-, or 32-bit integer, signed or unsigned.

If you use a different integer size than the native format of the device, one integer can contain multiple samples or one sample can stretch across multiple integers. For example, if you use 32-bit integers, but the device uses 8-bit samples, one integer contains up to four samples. If you use 8-bit integers, but the device uses 16-bit samples, a sample might require two integers. This behavior varies from device to device. Refer to your device documentation for more information.

NI-DAQmx does not separate raw data into channels. It returns data in an <u>interleaved or non-interleaved</u> 1D array, depending on the raw ordering of the device. Refer to your device documentation for more information.



Note If your device supports software calibration, NI-DAQmx does not calibrate raw samples. Refer to <u>calibration</u> to find out if your device uses software or hardware calibration.

Unscaled Data

Unscaled data is in the native format of the device, read directly from the device or buffer without scaling. The native format of a device can be an 8-, 16-, or 32-bit integer, signed or unsigned.



Note If your device supports software calibration, NI-DAQmx does not calibrate unscaled samples. Refer to <u>calibration</u> to find out if your device uses software or hardware calibration.

Waveform Timing Limitations

The limitation on t0 is that NI-DAQmx calculates the starting time for the task when data is read the first time. At this time, NI-DAQmx calculates the starting time for the task by reading the current system time and subtracting the number of samples acquired × dt from the system. Therefore, if you call read after the acquisition is complete, the calculated start time for the task is not accurate. This inaccuracy is reflected in the t0 returned with the waveform data.

The limitation on dt is that for certain timing types, NI-DAQmx cannot calculate the value of dt. When you use sample clock timing, NI-DAQmx calculates dt based on the rate of the clock. Because NI-DAQmx does not know the rate when handshake, implicit, on demand, or change detection timing is specified, NI-DAQmx returns dt as 0. Waveforms with a dt of 0 often do not work with the waveform analysis functions. However, you can always update the value of dt in your application if you know the expected rate of the timing source. Your ADE has an interface to update the value of dt.

Note The waveform data only supports symmetric timing between samples. If your timing is not symmetric such as if each sample has a time stamp, the waveform data format cannot contain the timing information. However, you can use your ADE's analysis library to resample the data using a constant dt. You can the use the resampled data with the waveform based analysis library.

Buffering

A buffer is a temporary storage in computer memory for acquired or tobe-generated samples. Typically this storage is allocated from your computer's memory and is also called the task buffer. For input operations, a data transfer mechanism transfers samples from your device into the buffer where they wait for a call to the Read function/VI to copy the samples to your application. For output operations, the Write function/VI copies samples into the buffer where they wait for the <u>data</u> <u>transfer mechanism</u> to transfer them to your device.

When Is a Buffer Created?

If you use the Timing function/VI and set the **sample mode** to finite or continuous, NI-DAQmx creates a buffer. If you set **sample mode** to hardware timed single point, NI-DAQmx does not create a buffer.

If you set the Data Transfer Mechanism to Programmed I/O or set the buffer size to zero by using either the Input or Output Buffer Config function/VIs, NI-DAQmx does not create a buffer (even if you also used the Timing function/VI). A data transfer mechanism of programmed I/O means there is no buffer.

See Also

How Is Buffer Size Determined?

Reference Triggering Impact on Buffers

Continuous Acquisition and Generation with Finite Buffer Size

Controlling Where in the Buffer to Read Samples

Read Status Attributes/Properties and Buffers

Controlling Where in the Buffer to Write Samples

Write Status Attributes/Properties and Buffers

How Is Buffer Size Determined?

Input Tasks

If your acquisition is finite (**sample mode** on the Timing function/VI set to Finite Samples), NI-DAQmx allocates a buffer equal in size to the value of the **samples per channel** attribute/property. For example, if you specify samples per channel of 1,000 samples and your application uses two channels, the buffer size would be 2,000 samples. Thus, the buffer is exactly big enough to hold all the samples you want to acquire.

If the acquisition is continuous (**sample mode** on the Timing function/VI set to Continuous Samples), NI-DAQmx allocates a buffer equal in size to the value of the **samples per channel** attribute/property, unless that value is less than the value listed in the following table. If the value of the **samples per channel** attribute/property is less than the value in the table, NI-DAQmx uses the value in the table.

Sample Rate	Buffer Size
no rate specified	10 kS
0–100 S/s	1 kS
100–10,000 S/s	10 kS
10,000–1,000,000 S/s	100 kS
>1,000,000 S/s	1 MS

You can override the default buffer size by calling the Input Buffer Config function/VI.

NI-DAQmx does not create a buffer when the **sample mode** on the Timing function/VI is set to hardware-timed single point.



Note Using very large buffers may result in diminished system performance due to excessive reading and writing between memory and the hard disk. Reducing the size of the buffer or adding more memory to the system can reduce the severity of these problems.

Output Tasks

For generations, the amount of data you write before starting a generation determines the size of the buffer. The first call to a Multiple Samples version of the Write function/VI creates a buffer and determines its size.

You also can use the Output Buffer Config function/VI to create an output buffer. If you use this function/VI, you must use it before writing any data.

The **samples per channel** attribute/property on the Timing function/VI does not determine the buffer size for output. Instead it is the total number of samples to generate. If n is your buffer size, setting **samples per channel** to $3 \times n$ generates the data in the buffer exactly three times. To generate the data exactly once, set **samples per channel** to n.

NI-DAQmx does not create a buffer when the **sample mode** on the Timing function/VI is set to hardware-timed single point.

Continuous Acquisition and Generation with Finite Buffer Size

The NI-DAQmx API uses circular buffers as shown in the following figure. For input operations, portions of data are read from the buffer while the buffer is filled. Likewise for output operations, portions of the buffer can be written to while the buffer is emptied. Using a circular buffer, you can set up your device to continuously acquire data in the background while NI-DAQmx retrieves the acquired data.



When a continuous operation reaches the end of the buffer, it returns to the beginning and fills up (or in the case of output operations, reads from) the same buffer again. Your input application must retrieve data in blocks, from one location in the buffer, while the data enters the circular buffer at a different location, so newer data does not overwrite unread data.

While a circular buffer works well in many applications, two possible problems can occur with this type of acquisition: Your application might try to retrieve data from the buffer faster than data is placed into it, or your application might not retrieve data from the buffer before NI-DAQmx overwrites the data into the buffer. When your application tries to read data from the buffer that has not yet been collected, NI-DAQmx waits for the data to be acquired and then returns the data. If your application does not read the data from the circular buffer fast enough, you receive an error, stating that some data has been overwritten and lost. If losing data in this way is not important to you, change the setting of the **OverWrite** Mode attribute/property.

Reference Triggering Impact on Buffers

Until the Reference Trigger occurs, the acquisition runs continuously even though you must set the **sample mode** parameter on the Timing function/VI to Finite Samples. The number of posttrigger samples in your buffer after the acquisition has finished is equal to the value of the **samples per channel** parameter from the Timing function/VI minus the number of pretrigger samples from the Trigger function/VI. When using a Reference Trigger, the default read position is **Relative To** First Pretrigger Sample with a read **Offset** of 0.
Controlling Where in the Buffer to Read Samples

Default read behavior depends on if a Reference Trigger is configured. If there is no Reference Trigger, NI-DAQmx reads samples beginning with the first sample acquired with each subsequent read beginning where the previous one left off. If there is a Reference Trigger, NI-DAQmx reads samples beginning with the first pretrigger sample and cannot begin reading until the acquisition has finished. This default behavior can be changed by using the **Relative To** and **Offset** attributes/properties.

The place where a read begins is called the Current Read Position. Each time data is read, the Current Read Position is computed based on the settings of the **Relative To** and **Offset** attributes/properties. When there is no Reference Trigger, the default for **Relative To** is Current Read Position. When there is a Reference Trigger, the default for **Relative To** is First Pretrigger Sample. In either case, the default for **Offset** is 0. Changing the settings of these two attributes/properties controls where in the buffer data is read.

During a continuous acquisition, for example, you can always read the most recent 1000 points by setting **Relative To** to Most Recent Sample and **Offset** to -1000. Even when a Reference Trigger is configured, you can begin reading samples immediately by setting **Relative To** to First Sample.

Read Status Attributes/Properties and Buffers

The three Read Status attributes/properties are useful for observing the progress of your acquisition. The **Current Read Position** is the place in the buffer where the next read begins if the **Relative To** attribute/property is Current Read Position and the **Offset** is 0. In any case, the Current Read Position is always where the last read left it. **Total Samples per Channel Acquired** is the total number of samples per channel acquired by the device and transferred into the buffer. **Available Samples per Channel** is computed by first calculating the Current Read Position based on the settings of the **Relative To** and **Offset** attributes/properties and then subtracting this number from **Total Samples per Channel Acquired**.

Controlling Where in the Buffer to Write Samples

By default, NI-DAQmx writes samples sequentially beginning with the first sample in the buffer, and each write begins where the previous one left off. The sample where a write begins is called the Current Write Position. Each time data is written, the Current Write Position is computed based on the settings of the **Relative To** and **Offset** attributes/properties. The default write behavior results from the default settings of these two attributes/properties. The default for **Relative To** is Current Write Position and the default for **Offset** is 0. Changing the settings of these two attributes/properties controls where in the buffer data is written.

Write Status Attributes/Properties and Buffers

The three Write Status attributes/properties are useful for observing the progress of your generation. The **Current Write Position** is the place in the buffer where the next write begins if the **Relative To** attribute/property is Current Write Position and the **Offset** is 0. In any case, the **Current Write Position** is always where the last write left it. **Total Samples per Channel Generated** is the total number of samples per channel generated by your device since the task started. **Space Available in Buffer** is computed by first calculating the Current Write Position based on the settings of the **Relative To** and **Offset** attributes/properties and then subtracting this number from the sum of **Total Samples per Channel Generated** and the buffer size. If regeneration is allowed, the **Space Available in Buffer** value is capped at the buffer size and grows from 0 to the buffer size repeatedly.

Glitching

Glitching refers to the generation of a waveform in which, when transitioning from old samples in the buffer to new samples, a mixture of old and new samples is generated rather than just the new samples. This situation may occur when continuously generating samples if the **Regeneration Mode** write attribute/property is set to Allow Regeneration. Glitching occurs when, while you write new samples, a subset of these new samples are generated and then, since you have not finished writing all of the new samples, a subset of the old samples is generated. After your write operation completes, only the new samples are generated.

NI-DAQmx reduces the likelihood of glitching by ensuring that the writing of new samples does not overtake the generation. This glitching protection works by pausing the write until the total samples generated is more than one buffer ahead of the current write position. However, NI-DAQmx does not ensure that the generation does not overtake the new samples being written. If this occurs, a glitch results, and NI-DAQmx reports the kWarningPotentialGlitchDuringWrite warning (error 200015). The following suggestions can help you to avoid generating glitches:

- Write new samples that are almost one buffer ahead of the total samples generated. By writing the new samples almost one buffer ahead of the total samples generated, there is less of a chance that the generation overtakes the new samples that are being written. If you are updating the entire buffer at a time, wait to write the new samples until the total samples generated attribute/property is one sample greater than an integral number of buffer sizes. For example, if the buffer size is 1000 samples, wait to write new samples until the total samples generated is either 1001, 2001, 3001, and so on.
- Increase the buffer size. If the buffer size is larger, there is less of a chance that the generation overtakes the new samples that are being written.
- Decrease the sample clock rate. If the sample clock rate is slower, there is less of a chance that the generation overtakes the new samples that are being written.

In the following graphs, the sine wave is generated from old samples and the square wave is generated from the new samples. The first graph depicts glitching.



The second graph depicts the same waveforms without glitching.



Data Transfer Mechanisms

There are four primary ways to transfer data across the PCI bus: Direct Memory Access (DMA), Interrupt Request (IRQ), Programmed I/O, and USB Bulk.

Direct Memory Access (DMA)

DMA is a mechanism to transfer data between the device and computer memory without the involvement of the CPU. This mechanism makes DMA the fastest available data transfer mechanism. National Instruments uses DMA hardware and software technology to achieve high throughput rates and to increase system utilization. DMA is the default method of data transfer for DAQ devices that support it.



Note DAQCard and USB devices do not support DMA.

Interrupt Request (IRQ)

IRQ transfers rely on the CPU to service data transfer requests. The device notifies the CPU when it is ready to transfer data. The data transfer speed is tightly coupled to the rate at which the CPU can service the interrupt requests. If you are using interrupts to acquire data at a rate faster than the rate the CPU can service the interrupts, your systems may start to freeze.

Programmed I/O

Programmed I/O is a data transfer mechanism in which a buffer is not used and instead the computer reads and writes directly to the device. Software-timed (on-demand) operations typically use programmed I/O.

USB Bulk

USB Bulk is a buffered, message-based streaming mechanism for data transfer. This high-speed method is the default transfer mechanism for USB devices.

Memory Mapping

Memory mapping is a technique for reading and writing to a device directly from your program, which avoids the overhead of delegating the reads and writes to kernel-level software. Delegation to the kernel is safer, but slower. Memory mapping is less safe because an entire 4 KB page of memory must be exposed to your program for this to work, but it is faster. Memory mapping is set by default if your device supports it.

Changing Data Transfer Mechanisms between DMA and IRQ

There are a limited number of DMA channels per device (refer to your device documentation). Each operation (AI, AO, and so on) that requires a DMA channel uses that mechanism until all of the DMA channels are used. After all of the DMA channels are used, you receive an error if you try to run another operation requesting a DMA channel. If appropriate, you can change one of the operations to use interrupts. For NI-DAQmx, use the **Data Transfer Mechanism** channel attribute/property.

Regeneration

Generating the same data more than once is called *regeneration*. You can configure NI-DAQmx to allow or disallow regeneration by setting the **Regeneration Mode** attribute/property. By default, NI-DAQmx allows regeneration for sample clock timing and disallows it for handshaking or burst handshaking timing. When regeneration is disallowed, new data must be continuously written to the device.

Allowing Regeneration and Using Onboard Memory

When the **Use Only Onboard Memory** attribute/property is true, NI-DAQmx transfers data only once to the device and that data is continually regenerated from there. Attempting to write new data to the device after starting the task returns an error. In addition, the amount of data written to the device before staring the task must fit in the onboard memory of the device.

When the **Use Only Onboard Memory** attribute/property is false, NI-DAQmx continuously transfers data from the host memory buffer to the device even though this data is not changing. Thus, if you write new data to the device after starting the task, that new data is generated and regenerated until you write more new data. This type of regeneration is sometimes called PC memory or user buffer regeneration.

When this attribute/property is false, you can also set the **Data Transfer Request Condition** attribute/property to specify when to transfer data from the host buffer to the device.

Signal Routing

A single routing API now controls all the digital routing for NI measurements devices. Signal routing controls the mapping of digital signals or triggers across hardware such as digital multiplexers or public trigger buses.

Here is a basic list of features in the signal routing API:

- A single unified signal routing API for all devices supported in NI-DAQmx
- Multi-device routing: a single route will be able to span two devices
- Logical inverter support
- Double driving prevention across public trigger buses

Specifying a Route

A route is a connection between a pair of terminals. The source and destination terminals make a terminal pair. Any time the source or destination terminal of a signal is specified, a route is created. Usually, you specify only one terminal for the route. For example, if you export a signal to the I/O connector, you set the destination terminal, but the source terminal is predetermined by the name of the signal. If you import a hardware trigger for a task, you can set the source terminal, but the destination terminal is predetermined by the name of the trigger.

Single-Device Routing Versus Multi-Device Routing

A single-device route is a connection between two terminals on the same physical device. Before NI-DAQmx, all routes were single-device routes. NI-DAQmx introduces multi-device routing. An example is specifying a terminal on a device as the source of a Start Trigger for a second device.

Creating Multi-Device Routes

NI-DAQmx supports multi-device routing. You simply specify the source terminal and destination terminal. If the two terminals are on different devices, NI-DAQmx uses the trigger bus to route the signal from the source device to the destination device. NI-DAQmx also selects and reserves an available trigger line on the trigger bus.

Plugging in and Registering Your RTSI Cable in MAX

To create a multi-device route, the source and destination devices must share a trigger bus both physically and logically in MAX. For PCI devices, you must register your RTSI cable in MAX. For more information on how to register (or add) a RTSI cable in MAX, refer to *Measurement & Automation Explorer Help for NI-DAQmx*. If you do not register your RTSI cable, NI-DAQmx fails to create a route. PXI trigger backplanes are automatically registered when you identify your chassis type in MAX.

Dynamically Selecting Trigger Bus Lines

Management of trigger lines is another feature of NI-DAQmx routing. If you hard-code two measurement tasks to the same trigger line for different signals, at least one of the measurement tasks causes a resource conflict. Multi-device routing allows you to dynamically select trigger lines at run time. This means that NI-DAQmx selects any available trigger line. You can still select a specific trigger bus line by splitting your multi-device route into two single-device routes. However, the two static routes lose the ability to dynamically choose an available trigger at run time.

Task-Based Routing

Task-based routing is the most common form of routing. When you create a hardware trigger or export a hardware signal, you create a task-based route. These routes are embedded in a task. You can use Export Signal function/VI to explicitly make a task-based route. When the task is committed, the route is committed. When the task is cleared, the route is unreserved. Clearing the task does not always clear the route. Refer to Lazy Line Transitions for more information.

Immediate Routing

Immediate routing is not associated with any task. An immediate route is a pair of <u>fully qualified terminal names</u> specifying the source and destination of the route. When an immediate route is created, the route gets committed to hardware immediately. Because an immediate route does not have a task governing its lifetime, you need to actively destroy the route. Create an immediate route with the Connect Terminals function/VI and destroy it with the Disconnect Terminals function/VI. Also, if you make an immediate route multiple times with several calls to Connect Terminals, only one call to Disconnect Terminals releases the route. There are other ways to destroy routes such as resetting the device. Refer to <u>Device Resetting and Interactions with Routing</u> for more information.

Logical Inversion of Signals

If you route a signal to or from an external device, you might need to invert the polarity of the signal. For example, you may need to change a high gating signal to a low gating one or look at falling edges instead of rising edges. With routing in NI-DAQmx, you can invert a signal during the routing. If there is an inverter available along the route, the inversion of the signal takes place. Inversion could fail if an overlapping route has previously reserved the inverter with an incompatible configuration.

Routing and Hardware Sharing

Two or more routes might overlap in a compatible fashion—especially if these two routes have the same source and destination. When routes overlap in a compatible fashion, the routing software handles this situation.

As an example, assume that two separate tasks make the same route. The resources associated with the routes are not released until both tasks have been unreserved. Mixing task-based and immediate routes is acceptable, too. However, the hardware resources are not released until all task-based routes have been released and the immediate route has been disconnected.

Releasing a task-based route using the Disconnect Terminals function/VI is not possible. You must release task-based routes by unreserving or clearing the task. In LabVIEW, if you explicitly create your task with the Create Task VI, you must clear it with the Clear Task VI. Otherwise, LabVIEW clears your task for you when the top level VI of your program stops executing.

Line Tristating Issues

During device initialization, all terminals on the I/O connector and trigger buses are tristated. Tristated means the terminal is floating or at high impedance. For the terminal to be driven from the device, the tristate buffer associated with the terminal must be enabled.

For instance, assume that you have a device with a single bidirectional terminal on the I/O connector. The terminal on the I/O connector is called the trigger terminal for reference purposes. Also, the trigger terminal of the device is bidirectional because it can accept an external trigger signal or export the internal trigger signal. The exported internal trigger signal could be different from the external trigger signal.

Scenario	Usage and Consequences
The trigger terminal is being driven by an external trigger signal only.	This is a common case for triggering an operation from an external source. As a result of this operation, you must disable the tristate buffer associated with the trigger terminal so that the internal trigger signal does not drive the trigger terminal, too.
The trigger terminal is being driven by the internal device trigger only.	In this case, an internally generated trigger is starting the device. This signal could be useful for other devices, too. To export this trigger signal, you must enable the tristate buffer associated with the trigger pin, so the device can drive the pin with the trigger signal. It is important that there is no external signal hooked up to the trigger terminal. If it is inconvenient to unhook the external signal, you must make sure the external signal is at least tristated.
The trigger terminal is being driven by both the internal device trigger AND an external trigger	Driving the trigger pin both internally and externally is called double driving. If the internal and external sources drive the signal in opposite directions, it signals problems. Usually the driving hardware is damaged, but more extreme consequences can occur as well. Remember to be very careful to avoid double driving any terminals on your I/O connectors.

signal.

Lazy Line Transitions

When a task-based route gets created and released, it does not necessarily go away. The hardware resources associated with the route are released, but the configuration might remain so that glitches are minimized.

By default, all tristate buffers associated with I/O connector terminals are disabled. When a task-based route with a destination on the I/O connector is released, the tristate buffer associated with the I/O connector terminal is not disabled. This means that even though the route was released, glitches are minimized on the destination terminal on the I/O connector. If you do not want this behavior, you can disable the tristate buffer associated with the I/O terminal function/VI. Or, if you initially created the route by exporting a signal with the Export Signal function/VI, you can also disable the tristate buffer by calling the Export Signal function/VI with the same signal name but with an empty string as the output terminal. Putting the terminal back into a tristate mode is necessary if an external signal must be connected to the I/O terminal. If the terminal is not tristated first, double driving the terminal damages the hardware.

All other connectors, such as the RTSI connector, use a different rule. When the task-based route associated with the RTSI connector is released, the tristate buffer associated with the RTSI terminal is disabled. The RTSI bus is a public bus that is shared by multiple devices. All drivers using the RTSI bus assume that all devices on the bus are tristated. The I/O connector is different because you have full control of it. You must keep track of which terminals are tristated or being driven by internal or external signals.

Device Resetting and Interactions with Routing

When you reset a device in NI-DAQmx, every immediate route and task associated with the device is invalidated. When the task is invalidated, all the routes are invalidated, too. If a task-based route is invalidated using a device reset, its parent task also is invalidated.

For instance, device A is running a task that performs an analog input operation. This same analog input operation receives its Start Trigger from device B. This task spans across device A and B due to the multidevice routing. If device B gets reset, all routes on device B are destroyed. The invalidation of the task-based route on device B causes its parent task on device A to be invalidated, too. You must consider these possible consequences when issuing a device reset. If the route between device A and B is an immediate route, there is not a relationship between the immediate route and the task. This could result in the task not being invalidated. You need to decide if you need to preserve the task.

Device Routing in MAX

NI-DAQmx exposes much more of your device's routing functionality than Traditional NI-DAQ (Legacy). You can discover what routes are possible by referring to a table of possible routes in MAX.

To find the device routing table for your device, launch MAX and select **Devices and Interfaces**»**NI-DAQmx Devices**. Click a device to open a tabbed window in the middle pane. Click the Device Routes tab at the bottom of the pane to display the device routing table.



Note MAX does not display the device routing table for SCXI chassis, SCC connector blocks, or RTSI cable devices.

Each cell in the table is an index with the valid source and destination terminal for the device. These are the same terminal names you can find in the Terminal Name I/O control in LabVIEW.

If a route is possible between a source and destination terminal, the intersecting cell is colored green or yellow. A green cell indicates the route can be made without consuming any important resource of your device. A yellow cell indicates that although the route is possible, something important must be consumed to create the route. Placing the cursor over a yellow square reveals the resource used in the **subsystem used** indicator. Usually, the sacrificed resource is a counter.

When you display the device routing table for a cDAQ chassis or a C Series device, the table contains all of the terminals for the chassis and all devices installed in the chassis.

Counters

This section provides an <u>overview of counters in NI-DAQmx</u> and the <u>two</u> <u>counter measurement method</u> for period and frequency measurements.

Paired Counters

For more complex and accurate measurements and generations, a counter is paired with another counter with dedicated connections to and from each counter. This pairing allows you to perform such operations as finite pulse-train generations, higher accuracy frequency and period measurements, and cascaded edge counting. Paired counters are generally numbered sequentially. For example, ctr0 and ctr1 are a pair, ctr2 and ctr3 are a pair, and so on.

Two Counter Measurement Method

For period and frequency measurements, you also can use a second counter. For most applications, the low frequency with one counter method is sufficient and desirable because it uses fewer resources. However, if you have a high-frequency or widely varying signal, you can use one of the two counter measurement methods—the high-frequency measurement method or the large-range measurement method. Depending on the rate of your input signal and measurement method used, your measurement is subject to different amounts of quantization error. In two counter applications, you only need to call the Create Channel function/VI once, specifying only the counter channel to which you want to connect your input signal. NI-DAQmx automatically takes care of making the internal routes necessary to perform the measurement across paired counters.

High Frequency Two-Counter Measurement Method

Use this high-frequency measurement method if you measure a digital frequency or period of a signal with a high frequency component. To perform measurements using this method in NI-DAQmx, a paired counter generates a pulse train with a period specified using the **measurement time** attribute/property. The **measurement time** is generally much larger than the period of the input signal being measured to reduce <u>quantization</u> error. However, the **measurement time** must be small enough to keep the counter from rolling over. The measurement counter counts the number of periods of the input signal that occur during the **measurement time**, averages the results, and returns the averaged value in the Read function/VI. The value returned is calculated as follows:

Period (in seconds) = Measurement Time / Number of Periods Counted

Frequency (in Hz) = Number of Periods Counted / Measurement Time

To determine if you should use the high-frequency measurement method, refer to the <u>quantization error</u> tables. If the quantization error listed for the one-counter method is too high, use the high-frequency measurement method instead.

Large-Range Two Counter Measurement Method

If you measure the digital frequency or the period of a counter signal, you can use this two-counter method to measure signals with large ranges. This method is useful when you have a widely varying signal to measure and would like increased accuracy throughout the entire range. Refer to the <u>quantization error</u> section for more information on increasing measurement accuracy with the large-range measurement method. You can also use this method to measure signal frequencies that are faster than your counter timebase rate as long as the input signal does not exceed the maximum input frequency supported by the counter.

To perform measurements using this method in NI-DAQmx, a paired <u>counter</u> is used to divide the input signal by a value specified using the **Divisor** attribute/property. However, you need to be careful the **Divisor** you choose does not cause the counter to roll over. This divisor has the effect of shifting the measurable frequency range upward. The **Divisor** scales the measured period and returns data according to the following equations:

```
Period = Measured Period / Divisor
```

Frequency = Divisor × Measured Period

For example, if you use a 24-bit counter and the Counter Timebase Rate is 100 kHZ, the measurable frequency range is approximately 0.006 Hz to 50 kHz because

 $\begin{aligned} & \text{Frequency} = \left(\frac{\text{Counter Timebase}}{\text{Count}}\right) \times \text{Divisor} \\ & \text{Frequency} = \left(\frac{100 \text{ kHz}}{2^{24}}\right) \times 1 = .006 \text{ Hz and} \left(\frac{100 \text{ kHz}}{2}\right) \times 1 = 50 \text{ kHz} \end{aligned}$

However, with a divisor of 4, the measurable frequency range is 0.024 Hz to 200 kHz because

$$\begin{aligned} & \text{Frequency} = \left(\frac{\text{Counter Timebase Rate}}{\text{Count}}\right) \times \text{Divisor} \\ & \text{Frequency} = \left(\frac{100 \text{ kHz}}{2^{24}}\right) \times 4 = .024 \text{ Hz and} \left(\frac{100 \text{ kHz}}{2}\right) \times 4 = 200 \text{ kHz} \end{aligned}$$

To determine if you should use the large-range measurement method, refer to the <u>quantization error</u> tables. If the quantization error listed for the one-counter method in that section is too high, use the large-range measurement method instead.

Quantization Error

Quantization error is the inherent uncertainty in digitizing an analog value as a result of the finite resolution of the conversion process. Quantization error depends on the number of bits in the converter, along with its errors, noise, and nonlinearities. Quantization error occurs due to phase differences between the input signal and the counter timebase. Depending on how the phase of the input signal and counter timebase align, the count measured has three possibilities:

- **Miss Both Edges**—The counter recognizes neither the first rising edge nor the last rising edge of the counter timebase, giving a count of one less than the expected value.
- Miss One, Catch One—The counter only recognizes the first rising edge or the last rising edge of the counter timebase, giving the expected value.
- **Catch Both Edges**—The counter recognizes both the first rising edge and the last rising edge of the counter timebase, giving a count of one more than the expected value.

GATE						-
SOURCE						
Miss Both Edges	0	1	2	з	з	
Miss One, Catch One	0	1	2	з	4	
Catch Both Edges	1	2	з	4	5	

For example, if the counter timebase rate is 20 MHz, and the frequency of the input signal is 5 MHz, the measured value can be 3, 4, or 5 due to quantization error. This corresponds to a measured frequency of 6.67 MHz, 5 MHz, or 4 MHz, resulting in a quantization error of as much as 33%.

Quantization Error with One Counter Time Measurements

For one counter time measurements, the following equation gives the quantization error.

```
Err<sub>Quantization</sub> = Actual Frequency / (Counter Timebase Rate - Actual Frequency)
```

You can reduce the quantization error for single counter time measurements by increasing the counter timebase rate. The following table shows the quantization error for various timebase rates with given input signal frequencies:

Actual Frequency of Input Signal	Counter Timebase Rate	Quantization Error
10 Hz	100 kHz	0.01%
100 Hz	100 kHz	0.10%
1 kHz	100 kHz	1.01%
10 kHz	100 kHz	11.11%
10 kHz	20 MHz	0.05%
100 kHz	20 MHz	0.50%
1 MHz	20 MHz	5.26%
2 MHz	20 MHz	11.11%
5 MHz	20 MHz	33.33%

For period and frequency measurements, if the quantization error is too large for your input signal, you might consider using one of the two counter period and frequency measurements.

Quantization Error with High Frequency Two Counter Method

For two counter high-frequency measurements, the following equations give the quantization error.

Err_{Quantization} = Actual Period / Measurement Time

Err_{Quantization} = 1 / (Measurement Time × Actual Frequency)

Increasing the measurement time reduces the quantization error. The quantization error also decreases with higher frequency input signals. The following table shows the quantization error for various measurement times and input signal frequencies:

Actual Frequency of Input Signal	Measurement Time	Quantization Error
10 kHz	1 ms	10.00%
100 kHz	1 ms	1.00%
1 MHz	1 ms	0.10%
5 MHz	1 ms	0.02%
10 MHz	1 ms	0.01%
10 kHz	10 ms	1.00%
---------	--------	----------
100 kHz	10 ms	0.10%
1 MHz	10 ms	0.01%
5 MHz	10 ms	0.002%
10 MHz	10 ms	0.001%
10 kHz	100 ms	0.10%
100 kHz	100 ms	0.010%
1 MHz	100 ms	0.001%
5 MHz	100 ms	0.0002%
10 MHz	100 ms	0.0001%
10 kHz	1 s	0.010%
100 kHz	1 s	0.0010%
1 MHz	1 s	0.0001%
5 MHz	1 s	0.00002%
10 MHz	1 s	0.00001%

As the table shows, quantization error is reduced at higher frequencies of the input signal. However, the advantage of this measurement method disappears at lower frequency input signals because you need to measure longer to gain accuracy, and you use up more resources.

Quantization Error with Large Range Two-Counter Measurement Method

For two counter large-range measurements, the following equations give the quantization error.

 $Err_{Quantization} = 1 / (Divisor \times Counter Timebase Rate \times Actual Period - 1)$

Err_{Quantization} = Actual Frequency / (Divisor × Counter Timebase Rate – Actual Frequency)

Increasing the divisor, increasing the counter timebase rate, or lowering the input signal frequency reduces the quantization error. The table lists the quantization error for various divisors and input signal frequencies assuming a counter timebase rate of 20 MHz.

Actual Frequency of Input Signal	Divisor	Quantization Error
1 kHz	4	0.00125%
100 kHz	4	0.125%
1 MHz	4	1.266%
1 kHz	10	0.0005%
100 kHz	10	0.05%
1 MHz	10	0.5%
1 kHz	100	0.00005%
100 kHz	100	0.005%
1 MHz	100	0.05%

1 .

Notice that the use of a divisor reduces the quantization error. Although the high frequency two-counter measurement method is more accurate at higher frequencies, the large range two-counter measurement method is more accurate throughout the range in a shorter amount of time. For example, if the input signal varies between 1 kHz and 1 MHz and you require a maximum quantization error of 2.0% at any signal range, you need a minimum measurement time of 50 ms using the high frequency two-counter measurement method. To gain the same accuracy using the large range two-counter method requires a maximum measurement time of 4 ms for any one measurement.

Counter Parts in NI-DAQmx

A counter contains several advanced terminals that you can use to perform time measurements and generate pulses. For most applications, NI-DAQmx automatically makes the proper routes from the terminal connector block to the correct advanced terminal with no additional routing required. For advanced applications, you might need to make explicit routes to the internal counter terminals.

Advanced Terminals and Common Counter Applications

The following lists the names of the advanced terminals as well as their common uses:

CtrnGate—The signal at this advanced terminal is used as the Start Trigger, pause trigger, sample clock, or the input signal being measured. The following table lists how this terminal is used in various applications:

Application	Purpose of Gate Terminal
Pulse Generation	Pause or Start Trigger
One Counter Time Measurements	Input Signal
Two Counter Time Measurements	Unused
Nonbuffered Edge Counting	Pause Trigger
Buffered Edge Counting	Sample Clock
Two-Edge Separation	Second Input Terminal
Position	Z Input Terminal

CtrnSource—The signal at this advanced terminal is either the input terminal for the measurement or the counter timebase. The following table lists how this terminal is used in various applications:

Application	Purpose of Source Terminal
Pulse Generation	Counter Timebase
One Counter Time Measurements	Counter Timebase
Two Counter Time Measurements	Input Terminal
Nonbuffered Edge Counting	Input Terminal
Buffered Edge Counting	Input Terminal
Two-Edge Separation	Counter Timebase

CtrnInternalOutput—The signal at this advanced terminal is where the pulsed or toggled output of the counter appears. The output of a counter pulses or toggles when the counter reaches terminal count. When counting down, the counter reaches terminal count when the count reaches zero. When counting up, the counter reaches terminal count when the counter rolls over. To configure the counter to toggle or generate pulses, use the Export Signal function/VI with Counter Output Event as the signal name.

For the output of a counter to appear at the I/O connector (Ctr0Out, for example), the signal on the internal output terminal must be routed to a terminal on the I/O connector. For pulse generations, this route is automatically made to the dedicated counter output terminal on the I/O connector. For measurements, if you are interested in observing this signal, you need to manually make this route to the appropriate pin on the I/O connector using the Export Signal function/VI with Counter Output Event as the signal name. After you route the internal output of a counter to the I/O connector, the signal remains on the I/O connector until the device is reset or you explicitly tristate the terminal.

CtrnAux (TIO-, STC II-, and STC III-Based Devices)—The following table lists how this terminal is used in various applications:

Application	Purpose of Source Terminal
Pulse Generation	Unused
One Counter Time Measurements	Unused
Two Counter Time Measurements	Unused
Nonbuffered Edge Counting	Optional Count Direction Terminal
Buffered Edge Counting	Optional Count Direction Terminal
Two-Edge Separation	First Input Terminal
Position	B Input Terminal

The main parts of a counter include the following:

A GATE input terminal controls when counting occurs. A GATE input is similar to a trigger because it starts or stops a count.

A SOURCE (CLK) input terminal is the timebase for a measurement or the signal to count.

A count register increments or decrements the number of edges to count. If the count register decrements, it counts down to zero. The count register size is the number of bits in the counter, and you calculate it as Count Register = $2^{no. of bits}$.

An OUT signal terminal can output a pulse or a pulse train, which is a series of pulses.



Configuring a Time Measurement in NI-DAQmx

To configure a measurement, you specify the expected range of the input signal. Based on this range, NI-DAQmx automatically picks the internal timebase that provides the highest resolution for your measurement and uses it as the counter timebase. You also can explicitly specify the source of the counter timebase by setting the **Counter Timebase Source** attribute/property and the rate of the timebase by setting the **Counter Timebase Rate** attribute/property. For more information on where to connect input signals, refer to <u>Connecting Counter Signals</u>.

To perform buffered time measurements, use the Timing function/VI with the Implicit timing type. After the acquisition begins, NI-DAQmx consecutively measures each sample of the input signal and stores it in the input buffer. Due to this consecutive measurement, the rate of the input signal implicitly determines the rate of the acquisition. Depending on the phase of the input signal in relation to the start of the measurement, the first sample of buffered measurements is often invalid. For instance, if you are performing a buffered period measurement, and you start the measurement when the input signal is halfway through its current cycle, the measured period for the first sample is half its expected value. Subsequent samples indicate the correct values because they are guaranteed to be taken after a full period of the input signal. For this reason, the first sample of buffered period, pulse width, and semi-period measurements often indicates a smaller value than the actual value. For buffered frequency measurements, the first sample often indicates a higher frequency than the actual frequency.

With bus-powered M Series USB devices, such as the NI 6210, NI 6211, NI 6212, NI 6215, NI 6216, and NI 6218, you can take buffered time measurements with the sample clock timing type. After the acquisition begins, your device consecutively measures each sample of the input signal but does not store it to the input buffer unless there is an active edge of the sample clock source signal. Using this timing type, the sample clock rate determines the acquisition rate rather than the input signal. With this buffered time measurement method, all measurements returned are a valid, complete cycle of your input signal. Using this method, you can measure signals that are much faster than your sample rate, which minimizes the amount of data transferred from your device to NI-DAQmx.

For non-buffered time measurements, calling the Read function/VI initiates the measurement and returns the next valid sample. Calling the Read function/VI repeatedly does not return consecutive measurements of the input signal.

Configuring a Displacement Measurement with NI-DAQmx

To configure a measurement, specify the initial sensor position through the **Initial Angle** attribute/property. You also can specify if the Z Input Terminal is used with the **Z Index Enable** attribute/property. You can configure the reload position on a Z index, and when a Z index position should cause a reload to occur in relation to the A and B signals, by using the **Z Index Phase** and **Z Index Value** attributes/properties, respectively.

When performing a single point, or on-demand, displacement measurement, you first arm the counter by calling the Start Task function/VI. Each subsequent read returns the current position of the encoder. If you perform multiple reads without first starting the counter, the counter implicitly starts and stops with each Read function/VI call, and the position is not recorded properly between read calls.

With a buffered displacement measurement, the device latches the current position onto each active edge of the sample clock and stores the position in the buffer. There is no onboard clock for buffered displacement measurement, so you must supply an external sample clock.

Terminals

A *terminal* is a named location where a signal is either generated (output or produced) or acquired (input or consumed). A terminal that can output only one signal is often named after that signal. A terminal with an input that can be used only for one signal is often named after the clock or trigger that the signal is used for. Terminals that are used for many signals have generic names such as RTSI, PXITrig, or PFI.

See Also

Signal and Terminal Confusion Terminal Names Signal Routing

Signal Versus Terminal

A signal is a means of conveying information. An analog waveform and a digital edge are both examples of signals. The word signal, in this section, refers to the digital edge variety, also known as hardware signals. A terminal, on the other hand, is a named location where a signal is either generated (output or produced) or acquired (input or consumed).

When a terminal shares a name with a signal, it is not always clear which is being referred to—the terminal or the signal. The sample clock provides a good example.

Within most devices, there is a terminal such that the signal at that terminal is always used as the sample clock. So when you refer to the sample clock signal, you refer to this terminal. For instance, for M Series analog input tasks, this terminal is named the ai/SampleClock terminal. For analog output tasks, this terminal is named the ao/SampleClock terminal.

When you use the Timing function/VI to select the source of the sample clock signal for your analog input task on an M Series device, you choose a signal at some other terminal to act as the source for the ai/SampleClock terminal. In other words, NI-DAQmx connects your chosen terminal (a PFI terminal pin, for instance) to the ai/SampleClock terminal. Selecting the ai/SampleClock terminal as the sample clock source returns an error because a terminal cannot be connected to itself.

Terminal Names

OnboardClock	An alias for the terminal within a device where the default source for a clock can be found. If your application does not set the source of a clock (or uses an empty string as the source), the clock's particular onboard clock is used. For example, the onboard clock for the ai sample clock is the ai Sample Clock Timebase.
PFIn	Programmable Function Interface — general-purpose input terminals, fixed- purpose output terminals. The name of the fixed output signal is often placed on the I/O connector next to the terminal as a hint.
PXITrig <i>n</i>	PXI Trigger bus —general-purpose input/output lines.
RTSIn	Real Time System Integration bus — general-purpose input/output lines. RTSI7 is the exception. It is the only line to use for the 20 MHz Timebase signal.
ai/SampleClock	A terminal within a device where the analog input sample clock can be found.
ai/StartTrigger	A terminal within a device where the analog input Start Trigger can be found.
ai/ReferenceTrigger	A terminal within a device where the analog input Reference Trigger can be found.
ao/SampleClock	A terminal within a device where the analog output sample clock can be found.
ao/StartTrigger	A terminal within a device where the analog output Start Trigger can be found.
20MHzTimebase	A terminal within a device where the onboard clock source for the master timebase can be found.
80MHzTimebase	A terminal within a device where the onboard

	clock source for the master timebase can be found.
MasterTimebase	A terminal within a device where the master timebase signal can be found. This signal originates either from the 20MHzTimebase terminal or the RTSI7 terminal. This signal is the onboard source for the Sample Clock Timebases and is one of the possible sources for the AI convert clock timebase.
100kHzTimebase	A terminal within a device where the 100 kHz Timebase signal can be found. This signal is created by dividing the signal at the 20MHzTimebase terminal by 200 and is one of the possible sources for the Sample Clock Timebases.
ai/ConvertClock	A terminal within a device where the AI Convert Clock can be found.
ai/ConvertClockTimebase	A terminal within a device where the AI Convert Clock Timebase can be found. This is the onboard clock source for the AI convert clock.
ai/HoldCompleteEvent	A terminal within a device where the AI Hold Complete Event signal can be found.
AIHoldComplete	The terminal at the I/O connector (external to the device) where the AI Hold Complete Event signal can be emitted.
ai/PauseTrigger	A terminal within a device where the analog input pause trigger can be found.
ai/SampleClockTimebase	A terminal within a device where the AI Sample Clock Timebase can be found. This is the onboard clock source for the AI sample clock.
AnalogComparisonEvent	A terminal within a device where the output of the analog comparison circuit, the Analog Comparison Event signal, can be found. This circuit is active whenever an analog edge or

	window trigger is configured.
ao/PauseTrigger	A terminal within a device where the analog output pause trigger can be found.
ao/SampleClockTimebase	A terminal within a device where the AO Sample Clock Timebase can be found. This is the onboard clock source for the AO sample clock.
Ctr0Out, Ctr1Out	Terminals at the I/O connector where the output of counter 0 or counter 1 can be emitted. You also can use Ctr0Out as a terminal for driving an external signal onto the RTSI bus.
Ctr0Gate, Ctr1Gate	Terminals within a device whose purpose depends on the application. Refer to <u>Counter</u> <u>Parts in NI-DAQmx</u> for more information on how the gate terminal is used in various applications.
Ctr0Source, Ctr1Source	Terminals within a device whose purpose depends on the application. Refer to <u>Counter</u> <u>Parts in NI-DAQmx</u> for more information on how the source terminal is used in various applications.
Ctr0InternalOutput, Ctr1InternalOutput	Terminals within a device where you can choose the pulsed or toggled output of the counters. Refer to <u>Counter Parts in NI-</u> <u>DAQmx</u> for more information on internal output terminals.
PairedCtrInternalOutput	A terminal within a device that chains counters together, creating a <u>paired counter</u> without using any external connections. If your application uses counter 0, PairedCtrInternalOutput refers to the output of counter 1. If your application uses counter 1, PairedCtrInternalOutput refers to the output of counter 0.

A terminal within a device that chains PairedCtrOutputPulse counters together without using any external connections. If you configure counter 0, PairedCtrOutputPulse refers to the pulsed output of counter 1. If you configure counter 1, PairedCtrOutputPulse refers to the pulsed output of counter 0. Refer to Paired Counters for more information. When the counter reaches terminal count (zero when counting down, its maximum count when counting up), the output of the PairedCtrOutputPulse pulses. By using this terminal, you can chain counters together to create a wider counter, perform buffered edge counting using the other counter as your clock source, perform finite pulse-train generation, and create other custom applications.

Note M Series and C Series devices do not have a master timebase of an arbitrary frequency. These devices use the 20 MHz/80 MHz/100 kHz timebase directly.

Analog Input Accessory Terminal Names

The following table lists the revised names for analog input terminal names.

Original Terminal Names	Revised Terminal Names	Explanation
AIGND, ACHGND	AIGND	The reference point for referenced single- ended measurements and the bias current return point for differential measurements
ACH#	AI#	AIO, AI1, and so on; the analog input channels
AISENSE	AISENSE	The reference point for <u>NRSE</u> measurements using channels 0–15
AISENSE2	AISENSE2	The reference point for <u>NRSE</u> measurements using channels 16–63
SCANCLK	AI HOLD COMP	The terminal where the AI Hold Complete Event signal appears
TRIG1	AI START TRIG	Placed as a hint next to the PFI terminal where the AI Start Trigger can be emitted
TRIG2	AI REF TRIG	Placed as a hint next to the PFI terminal where the AI Reference Trigger can be emitted
CONVERT*	AI CONV CLK	Placed as a hint next to the PFI terminal where the AI Convert Clock can be emitted
STARTSCAN	AI SAMP CLK	Placed as a hint next to the PFI terminal where the AI Sample Clock can be emitted

Analog Output Accessory Terminal Names

The following table lists the revised names for analog output terminal names.

Original Terminal Names	Revised Terminal Names	Explanation
DAC0OUT	AO0	An analog output channel
DAC1OUT	AO1	An analog output channel
EXTREF	AO EXT REF	AO external reference
AOGND	AO GND	The analog output ground
UPDATE*	AO SAMP CLK	Placed as a hint next to the PFI terminal where the AO Sample Clock can be emitted
WFTRIG	AO START TRIG	Placed as a hint next to the PFI terminal where the AO Start Trigger can be emitted

Counter Accessory Terminal Names

The following table lists the revised names for counter terminal names.

Original Terminal Names	Revised Terminal Names	Explanation
GPCTR1_SOURCE	CTR1SOURCE	Placed as a hint next to the PFI terminal where the Ctr1Source signal can be emitted
GPCTR1_GATE	CTR1GATE	Placed as a hint next to the PFI terminal where the Ctr1Gate signal can be emitted
GPCTR1_OUT	CTR1OUT	The name of the terminal where the Ctr1Out signal appears
GPCTR0_SOURCE	CTR0SOURCE	Placed as a hint next to the PFI terminal where the Ctr0Source signal can be emitted
GPCTR0_GATE	CTR0GATE	Placed as a hint next to the PFI terminal where the Ctr0Gate signal can be emitted
GPCTR0_OUT	CTR0OUT	The name of the terminal where the Ctr0Out signal appears
FREQ_OUT	FREQ OUT	The name of the terminal where the output of the 4-bit clock divider signal appears

Digital Accessory Terminal Names

The following table lists the revised names for digital terminal names.

Original Terminal Names	Revised Terminal Names	Explanation
DIO#	P0.#	Ports on devices are referred to by a number. Port A is called port 0, for instance. The # symbol refers to a single digital line
PA#, PB#, and so on	P0.#, P1.#, and so on	Ports on devices are referred to by a number. Port A is called port 0, for instance. The # symbol refers to a single digital line
DIOA#, DIOB#	P0.#, P1.#, and so on	The # symbol refers to a single digital line

Syntax for Terminal Names

The syntax for terminal names is a unique identifier that refers to a physical terminal in your system. To guarantee the uniqueness of a terminal name across multiple devices, terminal names begin with a forward slash, followed by the name of the device as configured in MAX, such as Dev1. A forward slash and the name of the terminal follow the device identifier, such as PFI3. For example, the fully qualified terminal name for PFI3 on Dev1 is /Dev1/PFI3.

Many fully qualified terminal names might have multiple forward slash delimiters. For example, the Start Trigger for the analog input subsection on Dev1 is /Dev1/ai/StartTrigger.

Coercion

When a value you set cannot be met exactly, NI-DAQmx sometimes adjusts—or coerces—that value to a valid one. Coercion often occurs when an attribute/property supports a set of discrete ranges.

After you set an attribute/property, you can query that attribute/property to determine its actual value after coercion.

See Also

Input Limit Coercion

Clock/Pulse Frequency Coercion

Input Limit Coercion

Some devices support only a discrete set of <u>device ranges</u>. When you specify <u>input limits</u>, NI-DAQmx <u>coerces</u> those values to fit within one of the supported device ranges.

For instance, suppose your device only supports ranges of 0 to 10 V, -5 to 5 V, and -10 to 0 V. If you set a maximum value of 8 V, NI-DAQmx coerces the maximum value to 10 V. NI-DAQmx also coerces scaled values, including <u>custom scaling</u>. If you have a temperature sensor that outputs 100 mV for every 1 °C, NI-DAQmx coerces a maximum value of 80 °C to 100 °C.

Because NI-DAQmx coerces input limits, <u>code width</u> is calculated based on the coerced values, which can be outside the minimum and maximum values you expect to measure.

Clock/Pulse Frequency Coercion

Frequencies of clocks or pulse trains must be evenly divisible into the frequency of its timebase. For example, the rate of the Sample Clock must be evenly divisible into the frequency of the Sample Clock Timebase. If you specify a Sample Clock rate that is not evenly divisible into the frequency of the Sample Clock Timebase, NI-DAQmx <u>coerces</u> the Sample Clock rate to one that is valid.

Calibration

There are two types of calibration, <u>channel calibration</u> and <u>device</u> <u>calibration</u>.

Device Calibration

What Is Device Calibration?

Device calibration consists of verifying the measurement accuracy of a device and adjusting for any measurement error. Verification consists of measuring the performance of the device and comparing these measurements to the published specifications. During calibration, you supply and read voltage levels or other signals using external standards, then you adjust the device calibration constants. The new calibration constants are stored in the EEPROM. These calibration constants are loaded from memory as needed to adjust for the error in the measurements taken by the device. There are two kinds of calibration, external and self. For more information on calibrating your device with NI-DAQmx, refer to Device Calibration Considerations. For detailed external calibration procedures, refer to ni.com/calibration.

External Calibration

External calibration, which is typically performed by a metrology lab, requires using a high-precision voltage source to verify and adjust calibration constants. This procedure replaces all calibration constants in the EEPROM and is equivalent to a factory calibration. Because the external calibration procedure changes all EEPROM constants, it invalidates the original calibration certificate. If an external calibration is done with a NIST-certified voltage source, a new NIST traceability certificate can be issued.

Self-Calibration (Internal Calibration)

Self-calibration adjusts the calibration constants with respect to an onboard reference stored on the device. The new calibration constants are defined with respect to the calibration constants created during an external calibration to ensure that the measurements are traceable to these external standards. The new calibration constants do not affect the constants created during an external calibration because they are stored in a different area of the device memory. You can perform a selfcalibration at any time to adjust the device for use in environments other than those in which the device was externally calibrated. You use the DAQmx Self Calibrate (or DAQmxSelfCal) function/VI to perform a selfcalibration.

Note Self-calibration does not require any external connections.

Channel Calibration

Channel calibration is a technique used to achieve higher measurement accuracy. In most applications, <u>device calibration</u> provides sufficient accuracy. However, in applications where the highest degree of accuracy is critical, channel calibration is necessary, but it does not replace device calibration. Channel calibration compensates for various errors, including those introduced by cabling, wiring, and sensors.

Control in NI-DAQmx

This section explains control concepts as implemented in NI-DAQmx. <u>Timing control loops</u>, <u>synchronizing analog input and output</u>, <u>using</u> <u>control algorithms</u>, <u>single-point real-time applications</u>, and <u>setting</u> <u>priorities for control applications in LabVIEW</u> are described.

For a general introduction to control, independent of the software you are using, refer to the <u>control overview</u>.

NI-DAQmx Single-Point Real-Time Applications

This section describes sample applications that demonstrate the functionality for hardware-timed single-point operations on real-time platforms.

Most of these applications use the Wait For Next Sample Clock function/VI, which guarantees tight synchronization between the hardware layer and the software layers for hardware-timed single-point tasks. Wait For Next Sample Clock provides an accurate way to correlate application execution to hardware signals, such as the sample clock for the given task, while providing feedback on the overall real-time execution of the control loop.

The following sections present common control applications:

- Hardware-Timed Simultaneously Updated I/O
- Hardware-Timed Simultaneously Updated I/O with Data
 Exchanges between Time-Critical and Non-Time-Critical Loops
- Hardware-Timed Input, Software-Timed Output
- Hardware-Timed Counter Tasks
- <u>Software-Timed I/O</u>
- Hardware-Timed Simultaneously Updated I/O Using the Timed Loop (LabVIEW Only)

Hardware-Timed Simultaneously Updated I/O

Requirement

The I/O must be hardware-timed. All output values need to simultaneously update at the arrival of the sample clock edge.

Solution

Use the Wait For Next Sample Clock function/VI to verify that a new sample clock edge has not yet occurred.

Advantages

- The current iteration's output samples are guaranteed to be aligned with the next iteration's input samples.
- NI-DAQmx returns an error if the Wait For Next Sample Clock function/VI does not start before the next sample clock edge occurs.
- I/O jitter is confined to the jitter of the hardware clock, which is on the order of a few nanoseconds.

Restrictions

Read, process, and write operations are confined to the time available between the moment the device starts acquiring data and the moment the next sample clock edge arrives.

Sample Application

An example of this kind of application is an analog control loop that reads samples from a specific number of analog input channels, processes the data using a control algorithm (such as PID), and writes new control values to the analog output channels.



Related Topic

Hardware-Timed Simultaneously Updated I/O

LabVIEW Example—Hardware-Timed Simultaneously Updated I/O

Note Although this example is written for LabVIEW users, the principles apply if you are using another ADE, such as LabWindows/CVI.

LabVIEW Example

- Wire the Wait For Next Sample Clock VI to one of the hardwaretimed tasks. Use dataflow wiring to guarantee that the Wait For Next Sample Clock VI executes after the AO Write call.
- If the Wait For Next Sample Clock VI does not start before the arrival of the next sample clock edge, it returns an error.

Sample Block Diagram



Notes

- Use only one Wait For Next Sample Clock VI within a LabVIEW loop. If you have multiple hardware-timed single-point I/O tasks within the same LabVIEW loop, you can connect the Wait For Next Sample Clock VI to any one hardware-timed single point task within that loop.
- If, when a cycle overflow occurs, you want to receive a warning rather than an error, set the **DAQmx Real-Time»Convert LateErrors to Warnings** property to True.
- Wait For Next Sample Clock has two modes of operation: Polling and Wait For Interrupt. Wait For Interrupt mode, which is the default, allows lower priority processes to execute while the timecritical loop waits for the next sample clock. Polling mode allows for higher sampling rates, but it prevents lower priority processes in the system from executing while the time-critical loop waits for the next sample clock.
- Analog DAQmx Read calls have two modes of operation: Polling and Wait For Interrupt. Wait For Interrupt mode allows lower priority processes to execute while the time-critical loop waits for all the requested samples to be converted. Polling mode allows for higher sampling rates, but it prevents lower priority processes in the system from executing while the time-critical loop waits for the converted analog samples.
- The specific application shown in this section assumes the use of Wait For Interrupt mode for both the Wait For Next Sample Clock VI and the DAQmx Analog Read VI. Too change these values, use the DAQmx Read»Advanced»Wait Mode and/or DAQmx Real-Time»Wait For Next Sample Clock Wait Mode properties.
Hardware-Timed Simultaneously Updated I/O with Data Exchanges between Time-Critical and Non-Time-Critical Loops

Requirement

The I/O needs to be hardware-timed. All output values need to simultaneously update at the arrival of the sample clock edge. Data needs to be exchanged between a time-critical loop and lower-priority processes.

Solution

- Use the Wait For Next Sample Clock function/VI to verify that a new sample clock edge has not yet occurred.
- Place the communication code (usually real-time FIFOs in LabVIEW or a thread-safe queue in LabWindows/CVI) after the Wait For Next Sample Clock function/VI.

Advantages

- The current iteration's output samples are guaranteed to be aligned with the next iteration's input samples.
- NI-DAQmx returns an error if the Wait For Next Sample Clock function/VI does not start before the next sample clock edge occurs.
- I/O jitter is confined to the jitter of the hardware clock, which is on the order of a few nanoseconds.
- Hardware-timed counter input operations have no conversion period similar to that of multiplexed analog input. Therefore, you can place the real-time FIFO, or the thread-safe queue, anywhere within the loop.

Restrictions

Read, process, and write operations are confined to the amount of time available between the moment the device starts acquiring data and the moment the next sample clock edge arrives.

Sample Application

An example of this kind of application is an analog control loop that reads samples from a specific number of analog input channels, processes the data using a control algorithm (such as PID), and writes the new control values to the analog output channels. The application uses a real-time FIFO to stop the control loop based on a Boolean value provided by a lower-priority process. A similar approach can employ the use of real-time FIFOs to vary the PID parameters on the fly, or to transfer acquired and control output values to lower-priority processes for data logging and remote monitoring.

Timing Diagram



Related Topic

LabVIEW example for Hardware-Timed Simultaneously Updated I/O with Data Exchanges between Time-Critical and Non-Time-Critical Loops

LabVIEW Example—Hardware-Timed Simultaneously Updated I/O with Data Exchanges between Time-Critical and Non-Time-Critical Loops



- **Note** Although this example is written for LabVIEW users, the principles apply if you are using another ADE, such as LabWindows/CVI.
 - Wire the Wait For Next Sample Clock VI to one of the hardwaretimed tasks. Use dataflow wiring to guarantee that the Wait For Next Sample Clock VI executes after the AO Write call.
 - Use dataflow wiring to guarantee that the real-time FIFO operations execute after the Wait For Next Sample Clock VI executes.
 - If the Wait For Next Sample Clock VI does not execute before the arrival of the next sample clock edge, it returns an error.

Sample Block Diagram



Notes

- Use only one Wait For Next Sample Clock VI within a LabVIEW loop. If you have multiple hardware-timed I/O tasks within the same LabVIEW loop, you can connect the Wait For Next Sample Clock VI to any one hardware-timed single-point task within that loop.
- If, when a cycle overflow occurs, you want to receive a warning rather than an error, set the **DAQmx Real-Time»Convert LateErrors to Warnings** property to True.
- Although you do not have to place the real-time FIFO code after the Wait For Next Sample Clock VI, it is highly recommended that you do so when dealing with multiple-channel analog input operations on multiplexed devices (such as E and M Series).
 Because the device can spend up to 50% of the sample period converting samples on the analog input channels, executing the FIFO code during this conversion period has the advantage of using up otherwise idle time.
- You can increase the Analog Input Conversion Rate manually through the DAQmx Timing Property Node. This reduces the total amount of time spent converting the requested number of samples. It is important to consider the minimum settling time specifications for the complete data acquisition system to avoid signal degradation and interference.

Hardware-Timed Input, Software-Timed Output

Requirement

An analog input task must be hardware-timed. The output task does not need hardware synchronization with the sample clock edge.

Solution

Use the DAQmx real-time **Report Missed Samples** attribute/property, which returns an error if new samples are available before the read operation finishes converting samples from the previous iteration.

Advantages

- Input samples are hardware-timed.
- Read, process, and write operations can overflow into the next sample period, as long as enough time remains for the subsequent read operation to complete on the next set of input samples. An application that acquires data from multiple channels on multiplexed devices (such as E Series and M Series) has to wait for the device to convert input samples before the read operation can return. By allowing process and write operations to overflow into the next sample period, the application takes advantage of otherwise idle time. This enables the application to achieve higher control-loop rates.

Restrictions

Output updates suffer from software jitter because they are not hardwaretimed.

Sample Application

An example of this kind of application is an analog control loop that reads samples from a specific number of multiplexed analog input channels, processes the data using a control algorithm (such as PID), and writes the new control values to the analog output channels using a softwaretimed task.

Timing Diagram



Related Topic

LabVIEW example for hardware-timed input, software-timed output

LabVIEW Example—Hardware-Timed Input, Software-Timed Output

- Note Although this example is written for LabVIEW users, the principles apply if you are using another ADE, such as LabWindows/CVI.
 - Set the **Report Missed Samples** property for the analog input operation to True.
 - The analog input operation returns an error if new samples are available before the read operation finishes converting samples from the previous iteration.

Sample Block Diagram



Notes

- If, when an Analog Input Read overflow error occurs, you prefer to receive a warning rather than an error, set the **Convert Late Errors to Warnings** property to True.
- Do not use the Wait For Next Sample Clock VI and the **Report Missed Samples** property within the same LabVIEW loop.
- Only hardware-timed single-point analog input tasks support the **Report Missed Samples** property.
- Because the analog output task is software timed, the value is written out as soon as the write call is initiated. It does not wait for a hardware clock to output the data.

Hardware-Timed Counter Tasks

Requirement

Use hardware-timed counter input operations to drive a control loop.

Solution

Use the Wait For Next Sample Clock function/VI to synchronize the counter operations with the counter's sample clock.

Advantages

- Counter tasks allow for flexible timing and event detection operations that can drive the software processing of the control loop. In other words, the control loop can have a dynamic clock rate.
- NI-DAQmx returns an error if the Wait For Next Sample Clock function/VI does not start before the next sample clock edge arrives.

Restrictions

Read, process, and write operations are confined to the time available between the moment the device starts acquiring data and the moment the next sample clock edge arrives.

Sample Application

An example of this kind of application is a control loop that uses a counter input task, such as count edges, while controlling digital lines based on some predefined control logic. This sample application performs communication through the use of real-time FIFOs. In LabWindows/CVI, you can use a thread-safe queue instead of real-time FIFOs.

Timing Diagram



Sample Application 2

Another example application is a control loop that monitors discrete inputs and uses the values to update a counter output task, using pulse frequency mode to generate pulse-width modulation control signals. This example application performs communication through the use of realtime FIFOs. In LabWindows/CVI, you can use a thread-safe queue instead of real-time FIFOs.

Timing Diagram



Related Topic

LabVIEW example for Hardware-Timed Counter Tasks

LabVIEW Example—Hardware-Timed Counter Tasks

- Note Although this example is written for LabVIEW users, the principles apply if you are using another ADE, such as LabWindows/CVI.
 - Wire the Wait For Next Sample Clock VI to the counter input task.
 - If the Wait For Next Sample Clock VI does execute before the arrival of the next sample clock edge, it returns an error.

Sample Block Diagram



Example 2

- Wire the Wait For Next Sample Clock VI to the counter output task.
- If the Wait For Next Sample Clock VI does not execute before the arrival of the next sample clock edge, it returns an error.

Sample Block Diagram



Notes

- Use only one Wait For Next Sample Clock VI within a LabVIEW loop. If you have multiple hardware-timed I/O tasks within the same LabVIEW loop, you can connect the Wait For Next Sample Clock VI to any one hardware-timed single point task within that loop.
- If, when a cycle overflow occurs, you want to receive a warning rather than an error, set the **DAQmx Real-Time»Convert LateErrors to Warnings** property to True.
- Hardware-timed counter operations have no conversion period similar to that of multiplexed analog input. Therefore, the real-time FIFO can be placed anywhere within the LabVIEW loop.
- NI-DAQmx provides a mechanism to recover after missing a sample clock edge when performing counter writes. If this write recovery mechanism is not successful, NI-DAQmx returns an error, and subsequent operations on that task are no longer hardware timed.
- The **DAQmx Real-Time**»Write Recovery Mode property allows you to choose between Wait For Interrupt or Polling mode for the recovery mechanism. Wait For Interrupt, which is the default, allows lower priority processes to execute while NI-DAQmx attempts to recover. Polling mode, on the other hand, allows for higher sampling rates.

Software-Timed I/O

Requirement

The I/O tasks do not support hardware-timed operations.

Solution

Apply software timing to your time-critical loop by using the Timed Loop in LabVIEW or asynchronous timers in LabWindows/CVI. Configure your NI-DAQmx tasks to use on-demand timing.

Advantages

- You can perform I/O control loops with operations that are not hardware-timed.
- Read, process, and write operations are confined to the software timing period that you define with the Timed Loop or asynchronous timers.

Restrictions

I/O samples suffer from software jitter.

Sample Application

An example of this kind of application is a digital I/O control loop. The application monitors the state of several discrete inputs and toggles the corresponding output based on the control algorithm. Hardware timing is not available for single-point digital I/O tasks in NI-DAQmx.

Timing Diagram



Related Topic

LabVIEW example for software-timed I/O

LabVIEW Example—Software-Timed I/O

 $\overline{\mathbb{N}}$

Note Although this example is written for LabVIEW users, the principles apply if you are using another ADE, such as LabWindows/CVI.

- A Timed Loop running off the system's time sources (millisecond or microsecond resolution) accomplishes the task. Configure the Timed Loop to run at the desired rate.
- Configure all tasks to be software-timed (on demand).
- The Timed Loop provides feedback to the application as to whether the previous iteration completed in time. It does this through the "Finished Late [i-1]" node.

Sample Block Diagram



Notes

- The Timed Loop allows the application to adjust its period from within the loop, allowing the implementation of dynamic timing algorithms for control.
- Lower-priority processes can execute while the Timed Loop waits until its next iteration.
- Other software timing methods include the use of the Wait and Wait Until next multiple VIs (with microsecond or milliseconds resolution). These methods provide no feedback when the application falls behind.

Hardware-Timed Simultaneously Updated I/O Using the Timed Loop (LabVIEW Only)

Requirement

I/O needs to be hardware-timed. All output values need to simultaneously update at the arrival of the sample clock edge. The application uses the Timed Loop.

Solution

Use the DAQmx Create Timing Source function/VI to create a timing source that drives a Timed Loop that contains the I/O operations and the control algorithm.

Advantages

- Using a timing source allows you to specify an I/O signal (for example, the sample clock signal) to trigger the execution of Timed Loop iterations.
- Timing sources such as the **Control Loop From Task** provide strict lateness checking and allow other threads to execute while several analog channels are being multiplexed and sampled.
- The Timed Loop provides feedback as to whether the iterations complete in time.
- Multi-rate applications, using distinct I/O hardware subsystems, are possible by extending this approach to multiple Timed Loops.

Restrictions

- Minor increase in overhead when compared to a regular LabVIEW While Loop
- Requires additional code to handle warm-up iterations

Sample Application

An example of this kind of application is an analog control loop that reads samples from a specific number of analog input channels, processes the data using a control algorithm (such as PID), and writes the new control values to the analog output channels.

You can create such an application with the **Control Loop From Task** timing source. The **Control Loop From Task** timing source uses the sample clock signal from the analog input task, which allows strict lateness-checking of all tasks associated with that sample clock.

The **Control Loop From Task** timing source also allows you to specify a delay between the time the sample clock event is handled and the time the Timed Loop starts executing. This delay, or sleep time, keeps the Analog Input Read function/VI inside the Timed Loop from using 100% of the CPU time available while waiting for the analog input samples to be multiplexed and digitized.

Timing Diagram



Related Topic

LabVIEW example for hardware-timed simultaneously updated I/O using the Timed Loop
LabVIEW Example—Hardware-Timed Simultaneously Updated I/O Using the Timed Loop

- Create a **Control Loop From Task** timing source for the Timed Loop. This signal serves as the timebase that drives the execution of the Timed Loop.
- The Timed Loop provides feedback to the application as to whether the previous iteration completed in time. It does this through the "Finished Late [i-1]" node.
- Allow a few warm-up iterations to account for the effects of processor-caching and other events that may occur during the first iterations of the loop.

Sample Block Diagram



Notes

- The Analog Input Read VI is implicitly configured to polling mode when using the **Control Loop From Task** timing source. Polling mode avoids the additional scheduling overhead associated with interrupts inside the Timed Loop.
- You can increase the Analog Input Conversion Rate manually through DAQmx Timing properties. This reduces the total amount of time spent converting the requested number of samples. It is important to consider the minimum settling time specifications for the complete data acquisition system to avoid signal degradation and interference.
- Do not use the Wait For Next Sample Clock VI for any of these tasks.
- Lower-priority processes, including other Timed Loops with lower priorities, can execute while the Timed Loop waits until its next iteration.
- To optimize multi-channel control applications in which lowerpriority threads might require additional processing time, provide a non-zero value for the sleep time (us) parameter of the Create Timing Source VI. This non-zero value allows other threads to use the time spent converting analog input samples to perform other tasks such as communication or logging to disk.
- The maximum amount of sleep time you can set without impacting the overall rate of the application depends on several factors, including the number of analog channels being acquired, the sample conversion rate, and the system's specifications.

The following diagram shows, for multiple channel configurations, the effect of the amount of sleep over the maximum achievable rate and the amount of work lower-priority threads can execute at such rates.



*See benchmark configuration below.

The graph shows that, when acquiring 8 channels using a specific hardware and software configuration, the maximum achievable rate decreases as soon as the amount of sleep time increases from 0 to 5 μ s. This is not, however, the case for the 12- and 16-channel configurations, for which increasing the amount of sleep up to 10 and 15 μ s respectively has no visible effect on the maximum achievable I/O rates. In the 12- and 16-channel case, the additional sleep interval allows other threads to execute more work (refer to definition below) without affecting the overall I/O rate of the application.

Benchmark Configuration

Hardware Configuration:

- NI PXI-8196 RT Controller
- NI PXI-6070 E Series MIO device
- NI PXI-6723 Analog Output device

Software Configuration:

- LabVIEW Real-Time 8.0
- NI-DAQmx 8.0
- Ethernet driver set to polling mode

Benchmark details:

- A work unit is defined as the number of times a normal-priority loop can increment an unsigned 64-bit number while the I/O Timed Loop, depicted in the sample block diagram above, runs in parallel with it.
- The analog input conversion is not explicitly configured. This means that the DAQmx driver auto-calculates it based on the number of channels and desired sample clock rate.

Timing Control Loops

You can time control loops using software timing or hardware timing. You can also use the Timed Loop structure.

For software timing, the software and operating system determines the rate at which the loop executes. Software timing is not <u>deterministic</u>. Controlling a while loop and using the Wait Until Next ms Multiple VI to handle timing is an example of a software-timed loop. Hardware timing uses the DAQ device internal clock or an external clock to control when a read executes within a loop. The example block diagram shown in *Control Loops* in the Common Applications section uses hardware timing.

The Timed Loop structure is hardware timed. It is ideal for multirate applications. By default, the Timed Loop uses the 1 kHz clock of the Windows operating system as its timing source. Refer to your *LabVIEW Help* for more information about the Timed Loop structure.

 $\overline{\mathbb{N}}$

Note The Timed Loop is only available in LabVIEW 7.1 and later.

Control Algorithms

There are many data processing algorithms to consider when creating a control application. You can create custom control algorithms using LabVIEW. You also can use VIs, such as the LabVIEW Control Design and Simulation Module VIs and Functions, to process control application data. This module offers several libraries that you can use to design, analyze, simulate, and deploy dynamic system models, including controllers.

Synchronizing Analog Input and Output

You can time analog output events and synchronize the events to the analog input clock. You can share a common clock for A/D and D/A conversions to ensure that both occur simultaneously. The advantage of this method is that, so long as your software completes in time, you mask out jitter. One caveat is that outputs are always one loop cycle behind the inputs, as shown in the following figure.



See also

Synchronization

Setting Priorities for Control Applications in LabVIEW

The relative priority of parallel processes is important in a control application. Because LabVIEW is multi-threaded, you can separate your application into individual tasks, each with its own priority. By setting priorities, time-critical tasks can take precedence over non-time-critical tasks. The time-critical task must periodically yield processor resources to the lower-priority tasks so they can execute. By properly separating the time-critical task from lower priority tasks, you can reduce application jitter. Refer to the *LabVIEW Real-Time User Manual* for details on assigning priorities to tasks.

I/O Cycles

The input and output operations along with any processing performed during a single sample clock period are called *I/O cycles*. I/O cycles can consist of reads or writes only, but an I/O cycle in a typical control application consists of reading data, processing that data, then writing a result.

NI-DAQmx Simulated Devices

This section includes information about timing and triggering, task behavior, and reading and writing data with NI-DAQmx simulated devices.

Refer to the *Measurement & Automation Explorer Help for NI-DAQmx* for detailed instructions on creating NI-DAQmx simulated devices and importing NI-DAQmx simulated device configurations onto physical devices.

Timing and Triggering

Task Behavior

Reading and Writing Data

Timing and Triggering with NI-DAQmx Simulated Devices

With NI-DAQmx simulated devices, the following timing and triggering considerations exist:

- NI-DAQmx simulated devices simulate timing for continuous analog input, digital input, and all output tasks. Timing is not simulated for counter tasks.
- NI-DAQmx simulated devices do not cause a timed loop to execute.
- NI-DAQmx simulated devices support software events. However, events that rely on the hardware, such as a sample clock event, are not supported.
- Triggers always occur immediately.
- Watchdog timers do not expire.

Task Behavior of NI-DAQmx Simulated Devices

NI-DAQmx tasks using NI-DAQmx simulated devices are verified just as tasks are on physical devices. If a property is set to an invalid value, the error returned for an NI-DAQmx simulated device is identical to the error returned for a real device. All resources necessary for the task are reserved for NI-DAQmx simulated devices. RTSI lines, PXI Trigger lines, DMA channels, counters, and so on are counted and reserved just as on physical devices.



Note NI-DAQmx simulated devices cannot be included in the same task with physical devices.

Reading and Writing Data with NI-DAQmx Simulated Devices

All NI-DAQmx simulated devices return analog input data in the form of a full-scale sine wave with 3% of full-scale noise. When multiple channels are in the task, the data for each channel is offset 5 ° in time. Digital data is always returned as if each eight-bit port were a binary counter. Counter data is always returned as 0.

Data written to an NI-DAQmx simulated device is scaled as if the device were real.

Deployment

Deployment refers to developing an application so that it can be distributed, or deployed, on a different computer than the one on which the application was developed. To deploy an application, you need the saved application and any configuration information the application and system requires.

When deploying an application, you must coordinate the use of configuration items that can be shared among multiple tasks. This includes devices, scales, and global virtual channels.

Developing Applications for Deployment

You can deploy NI-DAQmx applications in several ways:

- You can use the MAX Export Wizard to deploy an entire setup to another computer, including tasks, channels, scales, and devices.
- You can use the MAX Export Wizard to deploy an entire setup, except the device configurations, to another computer. You might choose to do this if the target computer already has tasks that rely on existing device configurations. In this case, you might have to make modifications after deployment so that your tasks and channels refer to the device configurations on the target computer.
- You can use the MAX Export Wizard initially to deploy a fixed set of device, scale, and global virtual channel configurations among a group of users. Each member of the group can create tasks that rely on the shared configurations, then create applications that use these tasks, and finally share the applications within the group. In some cases, the tasks deploy with the applications automatically. In other cases, you must deploy the tasks separately from the applications. Refer to the *Deploying Tasks and Channels* section for more information.

Deploying Tasks and Channels

Your tasks and channels deploy automatically with your application under the following circumstances:

- You use LabVIEW Express VIs for your DAQ applications.
- You save your tasks within a LabWindows/CVI or Measurement project, and the tasks use local channels only.
- You create your tasks and channels programmatically using the NI-DAQmx API.

You must deploy your tasks and channels using the MAX Export Wizard if you create your tasks and channels in the following ways:

- You create your tasks and channels directly in MAX.
- You create your tasks and channels in LabVIEW from the Task Name and Channel Name controls and do not generate configuration code.
- You create your tasks in LabWindows/CVI and neither generate configuration code nor copy the tasks to your project.

DAQmx I/O Server and Virtual Channels

In NI-DAQmx 8.0 or later, you can set up distributed applications to combine remote data applications using the NI-DAQmx I/O Server. For instance, you can configure a central computer to monitor other computers that control hardware sensors by using shared variables. You can connect to the NI-DAQmx I/O Server through LabVIEW 8.0, but LabVIEW is not required. If you use a third-party OPC client, you also can access DAQ channels.



Note You must have at least one global virtual channel defined either in the project or in Measurement & Automation Explorer (MAX) to use the NI-DAQmx I/O Server. Global virtual channels of any I/O type can be bound to shared variables, but tasks cannot.

When using a third-party OPC client, connect to the variable engine server to access DAQ channels on the network.

Refer to the *LabVIEW Help* for information about binding to a DAQ channel using a shared variable in a LabVIEW Project.

Translation Guide—Traditional NI-DAQ (Legacy) to NI-DAQmx

The following topics cover terminology changes from Traditional NI-DAQ (Legacy) to NI-DAQmx. These topics are intended for users of Traditional NI-DAQ (Legacy) who are transitioning to NI-DAQmx. Terminology for both the C API and LabVIEW are included.

- <u>Common Clock Names</u>
- Analog Input Clock and Trigger Names
- Analog Input Read Terminology
- Analog Output Clock and Trigger Names
- <u>Analog Output Write Terminology</u>
- Analog Output Physical Channel Names
- <u>Counter Clock and Trigger Names</u>
- <u>Counter Application Names</u>
- <u>Counter Physical Channel Names</u>
- Digital Physical Channel Names

Common Clock Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx for the C API and the LabVIEW API.

Traditional NI-DAQ (Legacy) C API	Traditional NI- DAQ (Legacy) LabVIEW	NI- DAQmx	Explanation
Board Clock	Board Clock	20 MHz Timebase	The onboard clock source for the master timebase from which other timebases are derived

Analog Input Terminology in Traditional NI-DAQ (Legacy) and NI-DAQmx

This section explains the differences between Traditional NI-DAQ (Legacy) and NI-DAQmx in the following areas:

- Analog Input Clock and Trigger Names
- Analog Input Read Terminology

Analog Input Clock and Trigger Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx for the C API and the LabVIEW API.

Traditional NI-DAQ (Legacy) C API	Traditional NI-DAQ LabVIEW	NI-DAQmx	Explanation
Sample Interval Counter (and Channel Clock)	Channel Clock or Interchannel Delay	Al Convert Clock	The E Series clock that directly causes analog-to-digital conversions
Points per Second (as Sample Rate)	Channels per Second	Conversions per Second	Units for specifying AI convert clock rate
Sample Timebase (and Channel Clock Timebase)	Timebase	Al Convert Clock Timebase	The clock that is divided down to produce the AI convert clock
Sample Interval	Timebase Divisor	Al Convert Clock Timebase Divisor	The number of AI Convert Timebase ticks used to divide down the AI Convert Timebase
Scan Interval Counter (and Scan Clock)	Scan Clock	Sample Clock	The clock that controls the time interval between samples. Each time the sample clock ticks (produces a pulse) one sample per channel is acquired.
Scans per Second (as	Scans per Second	Samples per Channel	Units for specifying sample acquisition rate (the sample

scanRate)		per Second	rate)
Scan Timebase	Timebase	Sample Clock Timebase	The clock that is divided down to produce the sample clock
Scan Interval	Timebase Divisor	Sample Clock Timebase Divisor	The number of Sample Clock Timebase ticks used to divide down the Sample Clock Timebase
Start Trigger	Start Trigger	Start Trigger	The trigger that begins an acquisition
Stop Trigger	Stop Trigger	Reference Trigger	The trigger that creates the reference point between the pretrigger samples and the posttrigger samples
No Equivalent	Scan Clock Gating	Pause Trigger	The signal that pauses and resumes an acquisition
No Equivalent	ATCOUT	Analog Comparison Event	The signal from the analog comparator circuit, used most often for analog triggering
SCANCLK	SCANCLK	Al Hold Complete Event	A digital signal emitted by an E Series device when the analog signal to be converted by the analog-to-digital converter has been held

Analog Input Read Terminology in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx for the C API and the LabVIEW API.

Traditional NI- DAQ (Legacy) C API	Traditional NI-DAQ LabVIEW	NI- DAQmx	Explanation
Reading, Voltage, Sample	Sample	Sample	A single measurement from a single channel
Scan	Scan	No Equivalent	A set of samples, one from each channel in the task
Sample Count	Scans per Buffer	Buffer Size (in Samples per Channel)	Method for specifying the buffer size
No Equivalent— the Sequential parameter in DAQ_Monitor is similar	Read/Search Mode	Relative To	Where to place the read position prior to adding the offset
No Equivalent	Read/Search Offset	Offset	After placing the read position, this offset is added to determine where the read takes place; all subsequent reads use this offset until the offset is changed
No Equivalent	Read Mark Scan	Current Read Position	The current read position
No Equivalent	Scan Backlog	Available Samples per Channel	The amount of unread data in the buffer

Retrieved	End of Data Scan	Total Samples per Channel Acquired	The total number of samples per channel acquired since the start of the task
Count	Total Scans to Acquire	Samples per Channel	The total number of samples to acquire
numPts	Number to Read	Number of Samples per Channel	The number of samples to read

Analog Output Terminology in Traditional NI-DAQ (Legacy) and NI-DAQmx

This section explains the differences between Traditional NI-DAQ (Legacy) and NI-DAQmx in the following areas:

- Analog Output Clock and Trigger Names
- Analog Output Write Terminology
- Analog Output Physical Channel Names

Analog Output Clock and Trigger Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx for the C API and the LabVIEW API.

Traditional NI-DAQ C API	Traditional NI-DAQ LabVIEW	NI- DAQmx	Explanation
Update Clock	Update Clock	Sample Clock	The clock controlling the time interval between samples. Each time the sample clock ticks (produces a pulse) one sample per channel is generated.
Updates per Second	Updates per Second	Samples per Channel per Second	Units for specifying digital-to-analog conversion (DAC) rate
Timebase	Timebase	Sample Clock Timebase	The clock that is divided down to produce the sample clock
Interval	Timebase Divisor	Sample Clock Timebase Divisor	The number of Sample Clock Timebase ticks used to divide down the Sample Clock Timebase
Start Trigger	Start Trigger	Start Trigger	The trigger that begins a generation
No Equivalent	Gate	Pause Trigger	The signal that pauses and resumes a generation

Analog Output Write Terminology in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx for the C API and the LabVIEW API.

Traditional NI-DAQ C API	Traditional NI-DAQ LabVIEW	NI-DAQmx	Explanation
Voltage, Sample	Update	Sample	A single measurement generated at a single channel
No Equivalent	Update	No Equivalent	A set of samples, one for each channel in the task
No Equivalent	Write Mode	Relative To	Where to place the write position prior to adding the offset
No Equivalent	Write Offset	Offset	After placing the write position, this offset is added to determine where the write takes place. All subsequent writes use this offset until the offset changes
No Equivalent	Write Mark	Current Write Position	The place in the buffer where the next write begins if the Relative To attribute/property is Current Write Position and the Offset attribute/property is 0
No Equivalent	No Equivalent	Space Available in Buffer	The number of samples that can be written without overwriting a sample that has not been output
pointsDone, itersDone	Output Mark, Buffer Iterations	Total Samples Per Channel	The total number of samples that have been generated by the device

		Generated	since the task began
oldDataStop	Regeneration Mode	Regeneration	The attribute/property that controls whether old data is regenerated
partialTransfer	Regeneration Mode	No Equivalent	To stop a waveform generation after a specific number of samples are generated, set the NI- DAQmx attribute/property samples per channel to the desired number of samples.
Iterations	Iterations	Samples per Channel	To generate a finite number of iterations of a buffer, set the samples per channel attribute/property to $I \times N$, where I is the desired number of iterations, and N is the number of samples per channel in the buffer.

Analog Output Physical Channel Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx.

Traditional NI-DAQ	NI- DAQmx	Explanation
0, 1, and so on	Dev1/ao0, Dev1/ao1, and so on	In Traditional NI-DAQ (Legacy), physical channel names are numbers. In NI-DAQmx, physical channels are string names that combine the device name, the I/O type, and the physical channel number.

Counter Terminology in Traditional NI-DAQ (Legacy) and NI-DAQmx

This section explains the differences between Traditional NI-DAQ (Legacy) and NI-DAQmx in the following areas:

- Counter Clock and Trigger Names
- <u>Counter Application Names</u>
- <u>Counter Physical Channel Names</u>

Counter Clock and Trigger Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx for the C API and the LabVIEW API.

Traditional NI- DAQ (Legacy) C API	Traditional NI- DAQ (Legacy) LabVIEW	NI- DAQmx	Explanation
Start Trigger	Start Trigger	Start Trigger	The signal that begins a counter operation
Start Trigger for NI TIO-Based Devices	Start Trigger	Arm Start Trigger	The signal that begins a counter operation
Gate	Gate	Pause Trigger	The signal that pauses and resumes a counter operation

Counter Application Names

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx for both the C and the LabVIEW API.

Traditional NI- DAQ (Legacy) C API	Traditional NI- DAQ (Legacy) LabVIEW	NI-DAQmx	Explanation
Event Counting	Event Counting	Edge Counting	A counter tallies the total number of rising or falling edges
No Equivalent	Frequency Shift-Keying	No Equivalent	NI-DAQmx does not support frequency shift-keying
Two Signal Edge Separation Measurement	Two Signal Edge Separation Measurement	Two Edge Separation Measurement	The time between the rising or falling edge of one digital signal and the rising or falling edge of another digital signal
Pad Synchronization	Pad Synchronization	Digital Synchronization	Synchronizes the input signal on a particular PFI line to the maximum timebase available on the device

Counter Physical Channel Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx.

Traditional NI-DAQ (Legacy)	NI- DAQmx	Explanation
0, 1, and so on	Dev1/ctr0, Dev1/ctr1, and so on	In Traditional NI-DAQ (Legacy), physical channel names are numbers. In NI-DAQmx, physical channels are string names that combine the device name, the I/O type, and the physical channel number.

Counter Attribute/Property Names

Traditional NI-DAQ (Legacy)	NI-DAQmx	Explanation
Synchronous Counting	Duplication Count Prevention	Prevents incorrect duplicate counts caused by a timebase being slower than the signal you want to measure

Digital Terminology in Traditional NI-DAQ (Legacy) and NI-DAQmx

This section explains the differences between Traditional NI-DAQ (Legacy) and NI-DAQmx in the <u>physical channel names</u>.
Digital Physical Channel Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx.

Traditional NI-DAQ (Legacy)	NI-DAQmx	Explanation
0, 1 as port	Dev1/port0, Dev1/port1, and so on	In Traditional NI-DAQ (Legacy), physical channel names are numbers. In NI- DAQmx, physical channels are string names that combine the device name, the I/O type, and the physical channel number. If the line number is omitted from the NI-DAQmx name, all lines in the port are included.
0, 1 as line	Dev1/port0/line0, Dev1/port0/line1	When the line number is present in the NI- DAQmx name, only that line is used.
No Equivalent	Dev1/port1_16	Concatenated ports, port a_b, where a is the port number of the beginning port and b is the width in lines

Digital Clock and Trigger Names in Traditional NI-DAQ (Legacy) and NI-DAQmx

The following table translates terms from Traditional NI-DAQ (Legacy) to NI-DAQmx.

Traditional NI-DAQ (Legacy)	NI- DAQmx	Explanation
Start Trigger	Start Trigger	The trigger that begins an acquisition or generation.
Stop Trigger	Reference Trigger	The trigger that creates the reference point between the pretrigger samples and the posttrigger samples.

Functions, VIs, Properties, and Attributes

You program your device primarily with the functions (or VIs in LabVIEW) that make up the NI-DAQmx API. The functions contain the core functionality of the API, but for advanced or uncommon functionality, you can use the following:

- Properties for LabVIEW, Visual C++, Visual Basic .NET, and Visual C#
- Get and Set Attribute functions for ANSI C and LabWindows/CVI

Refer to your function or VI reference help for detailed information on available attributes and properties.

External Reference Sources for Generating Voltage

Devices that support an external voltage reference enable you to maximize the resolution of your device. If the voltages you want to generate do not exceed a certain level and you can supply an external reference voltage at that level, you achieve your device's maximum resolution. The external reference voltage settings are available as a Channel property in the **Analog Output**»General **Properties**»DAC»Reference Voltage.

You also can use external reference voltages to apply a gain to a DC voltage or to a time-varying waveform. For example, set your external reference voltage level to 1.0, and write a sine wave buffer with values from -1.0 to +1.0 V. When you apply an actual reference voltage of 2.0 V, your signal jumps to ± 2 V in amplitude. Increasing the reference voltage level to 3.0 again jumps the signal to ± 3 V. Applying a reference voltage level of 0.0 V immediately flat-lines your time-varying voltage signal at 0.0 V.

The terminal you use for external reference sources varies <u>depending on</u> <u>your device</u>.

Custom Scales

You can create scales to specify a conversion from the prescaled units measured by a channel to the scaled units associated with your transducer or actuator. For input channels, the scale converts samples read to the final scaled units. For example, a scale could convert a voltage to a linear position. For output channels, the scale converts samples written to the prescaled units of the channel. For example, a scale could convert a linear position to a voltage. You also can use scales to calibrate samples read or written so that the final scaled units are identical to the prescaled units of the channel.

Often, you do not need to create a scale because NI-DAQmx has explicit support for many of the most common transducers, sensors, and actuators. For example, when creating an analog input temperature channel, you can specify the type of transducer (for example, thermistor, RTD, or thermocouple) used to make the measurement when creating the channel. However, if NI-DAQmx does not explicitly support your transducer or actuator, you can create a scale that specifies how to convert from the prescaled units to the scaled units. You can associate the same scale with multiple channels. You do not need to create a scale for each channel if the scale is the same. After a scale is assigned to a channel, the scale applies to all attributes normally expressed in the prescaled units of the channel. For example, if a custom scale, which converts volts to meters, is assigned to a voltage channel, the channels minimum and maximum attributes are expressed in meters.

Prescaled Versus Scaled Units

Prescaled refers to values expressed in the unit of the channel prior to the custom scale being applied. Usually, these prescaled units are volts or amps since scales are most often associated with channels that natively measure or generate signals using these units. However, it is possible to associate a scale with a channel that contains a transducer explicitly supported by NI-DAQmx. In this case, the prescaled units are the units of the channel including the explicitly supported transducer. For example, you can create a analog input resistance channel and associate a scale with this channel. In this example, the prescaled units would be ohms and the scale would specify how to convert from ohms to the desired scaled units.

Scaled refers to values expressed in the final unit after NI-DAQmx applies the custom scale. For example, a linear-position-to-voltage scale is assigned to a voltage output channel. In this case, the prescaled samples are in volts while the scaled samples could be specified in meters. Scaled units are the units that are most convenient for your application. You have complete control over the scaled units when specifying your scale. The scale specifies the conversation from the prescaled units of the signal measured or generated by the channel to your specified scaled units. When you read samples from a channel associated with a scale, the samples are in scaled units. Likewise, when you write samples to a channel associated with a scale, the samples are in scaled units.

You can create scales in the DAQ Assistant or programmatically. When you programmatically associate a scale with a channel, you must set the **custom scale name** attribute/property to the name of the scale and set the **units** attribute/property to From Custom Scale.

Note Unscaled data is not synonymous with prescaled units. Unscaled data refers to an 8-, 16-, or 32-bit integer in the native format of the device. Prescaled refers to the units of measurement, such as volts or amps, before a custom scale is applied.

Example—Converting Volts to Revolutions/Minute

Imagine that you have connected an analog output voltage channel to a motor whose speed is proportional to the generated voltage, and you want to create a scale that specifies this conversion. The prescaled units in this case would be volts and the scaled units could be revolutions/minute. You would then specify the equation, table, or map that converts from volts to revolution/minute. After you have created this scale, you would associate the scale with an analog output voltage channel. Now, rather than having to convert between volts and revolutions/minute when operating your application or having to develop additional code in your application to perform this conversion, you can simply write samples in units of revolutions/minute directly to the channel associated with the scale and NI-DAQmx automatically performs the specified conversions. Scales can simplify your code and improve the usability of your application.

Applying Scales That Do Not Monotonically Increase or Decrease

Some scale types allow scales that do not monotonically increase or decrease. This is problematic because application of the scale may not produce the desired results. For example, if multiple prescaled values map to the same scaled value, the conversion from the scaled value to the prescaled values is ambiguous. The conversion is well defined and predictable even in these cases. While not disallowed, non-monotonically increasing scales should be avoided or used with caution.

Using Traditional NI-DAQ (Legacy) and NI-DAQmx in the Same Application

You can use both Traditional NI-DAQ (Legacy) and NI-DAQmx in the same computer, and in the same application, but there are some restrictions. After using a device in NI-DAQmx, you must unreserve all NI-DAQmx tasks that are using that device before you can use that device through Traditional NI-DAQ (Legacy). After using a device in Traditional NI-DAQ (Legacy), you must reset the device before you can use that device in NI-DAQmx.

Refer to <u>ni.com/support</u> for details and instructions about the following topics:

- How to add NI-DAQmx code to a Traditional NI-DAQ (Legacy) application
- How to run both Traditional NI-DAQ (Legacy) applications and NI-DAQmx applications that use the same device