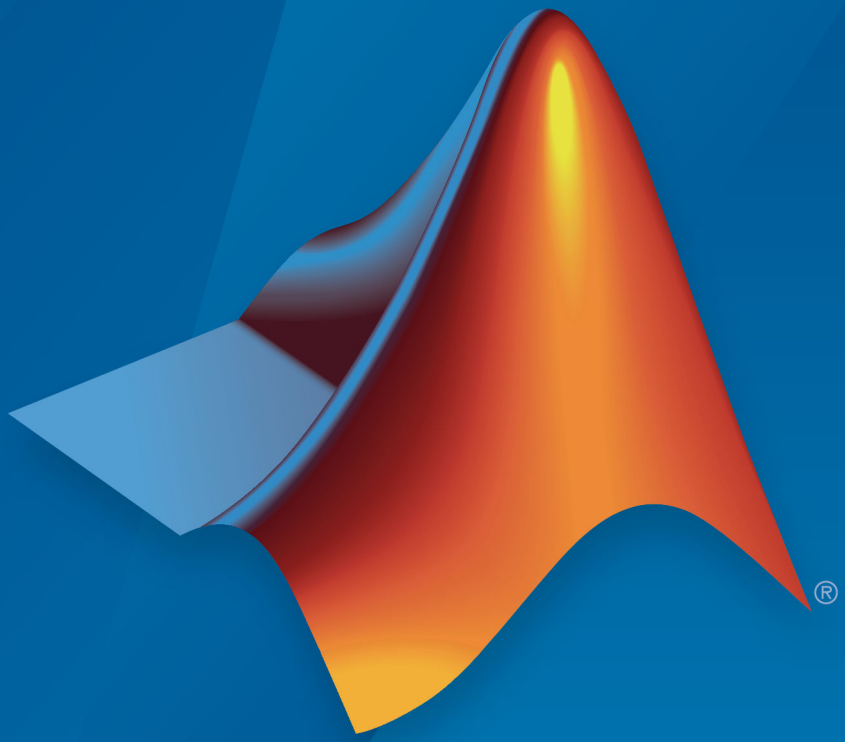
Data Acquisition Toolbox™

SDK User's Guide



MATLAB®

R2019b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Data Acquisition Toolbox™ SDK User's Guide

Revision History

March 2017	Online only	New for Version 1.1 (Release 2017a)
September 2017	Online only	Revised for Version 1.1 (Release 2017b)
March 2018	Online only	Revised for Version 1.2 (Release 2018a)
September 2018	Online only	Revised for Version 1.2 (Release 2018b)
March 2019	Online only	Rereleased for Version 1.2 (Release 2019a)
September 2019	Online only	Rereleased for Version 1.2 (Release 2019b)

Contents

1 SDK Overview

2 Explore the SDK Demo Adaptor

3 Custom Adaptor Creation

4 API Reference

5 State and Sequence Diagrams

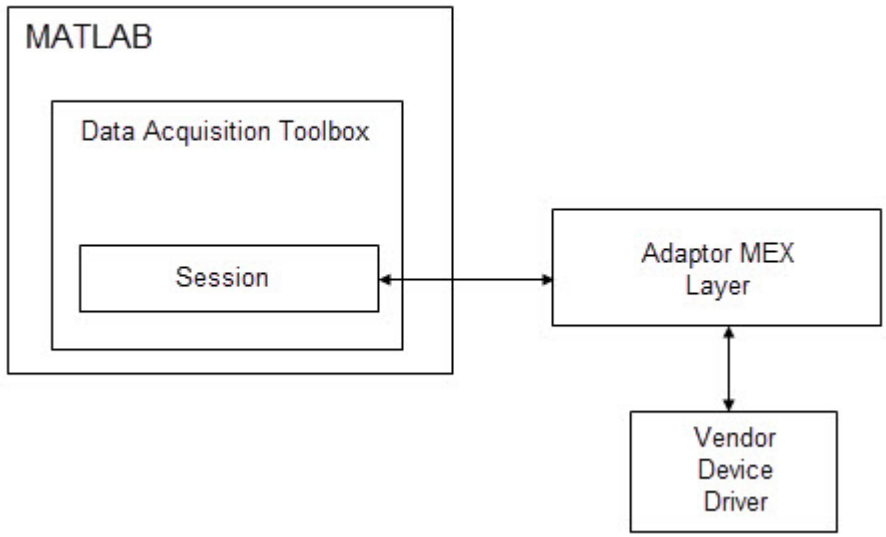
6 Functions — Alphabetical List

SDK Overview

Toolbox and Adaptor

Hardware Driver Adaptor

The hardware driver adaptor is the interface between the data acquisition engine and the hardware driver. The adaptor's main purpose is to pass information between MATLAB® and your hardware device via its driver.



Hardware drivers are provided by the device vendor. For example, to acquire data using a National Instruments® board, the appropriate version of the driver must be installed on your system. Hardware drivers are not installed as part of the toolbox, but a suitable driver is usually installed on PCs that are equipped with a sound card. For any other devices, the drivers must be installed.

See Also

Related Examples

- “Adaptor Creation Summary” on page 1-5

SDK Contents

An installation of Data Acquisition Toolbox includes the following folders to support its SDK. Your MATLAB installation location is referred to as *`matlabroot`*.

Folder	Description
`matlabroot``\toolbox\daq\daqsdk\bin\win64`	Built executable files for demo and vendor adaptors.
`matlabroot``\toolbox\daq\daqsdk\tests\+daq\+sdk\+tests`	Test files for adaptors. The installed set of test files is for the demo adaptor.
`matlabroot``\toolbox\daq\daqsdk\src\daqadaptor`	Adaptor C++ source files, one folder for each adaptor, and one other folder named `Shared` for elements common to all adaptors. `DemoAdaptor` contains all the source files for the demo adaptor. `VendorAdaptor` contains a set of templates.

See Also

Related Examples

- “Adaptor Creation Summary” on page 1-5
- “Create Your Adaptor from the Demo Adaptor” on page 3-2
- “Modify Demo Tests for Your Adaptor” on page 3-8
- “Vendor Adaptor Templates” on page 3-20

Adaptor Creation Summary

This topic provides a summary of adaptor creation with the SDK. For details and examples of these steps, see “Create Your Adaptor from the Demo Adaptor” on page 3-2.

1. Copy the demo adaptor or vendor adaptor source files into your working folder.
2. Change the names of the source files to reflect your own adaptor name.
3. Update the content of the source files so that the new names are used for references to other files, the adaptor, devices, and vendor.
4. Update the source file functions to use your driver code. For more information, see “Adaptor API Reference” on page 4-2.
5. Build the adaptor with the `buildAdaptor` function. Add the folder containing the built MEX-file to your MATLAB path.
6. Copy the demo adaptor tests and modify them for your adaptor. Add the test package folder to your MATLAB path.
7. Run the tests on your adaptor.
8. Deliver the finished adaptor MEX-file with your device driver and supporting files.

Tips

- Update, build, and test your adaptor iteratively one step at a time. Develop and test in small increments, proceeding upon the success of each step.
- When modifying the source files, do not remove any of the functions. Even if you do not use all the functions, they must be present when using `buildAdaptor`.

See Also

Related Examples

- “Create Your Adaptor from the Demo Adaptor” on page 3-2
- “Modify Demo Tests for Your Adaptor” on page 3-8

More About

- “Vendor Adaptor Templates” on page 3-20

- “Deliver Your Adaptor” on page 3-22

Explore the SDK Demo Adaptor

Demo Adaptor Description

The demo adaptor installed with Data Acquisition Toolbox consists of the files described in the following tables.

Source Files

The demo adaptor source files are in *matlabroot*\toolbox\daq\daqsdk\src\daqadaptor\DemoAdaptor.

File	Description
demoadaptor.hpp, demoadaptor.cpp	Wraps device driver code in methods that allow you to configure, discover, and enumerate the hardware in MATLAB.
daqstream_analog.hpp, daqstream_analog.cpp, daqstream_digital.hpp, daqstream_digital.cpp	Implement DAQStream objects for an analog and digital subsystems, that allow you to stream data to and from the hardware.
custom_demo.cpp	Dispatches calls from MATLAB to custom functions in the demo adaptor. At a minimum this must contain a customizeMap function.

In addition to these files, the demo adaptor also uses some of the source files in *matlabroot*\toolbox\daq\daqsdk\src\daqadaptor\Shared:

Files	Purpose
adaptorfactory.cpp, adaptorfactory.hpp	Create adaptor for dispatch and streaming.
daqadaptor.cpp, daqadaptor.hpp	Implement adaptor class.
daqapi.h	C interface.
daqdatatypes.hpp	C++ equivalents of session data types.
daqinterfaces.hpp	IAdaptor/IDriver, called before streaming.
daqstream.cpp, daqstream.hpp	Transfer streaming data between MATLAB and device driver

Files	Purpose
`dispatcher.cpp`, `dispatcher.hpp`, `dispatcher_common.hpp`	MATLAB calls to convert data and call adaptor functions.
`fakevendordriver.hpp`	Fake or virtual driver for testing and demonstrations.
`globals.h`	Global settings.
`mxconvert.hpp`	Utility functions for data type conversions.

Class Definitions in MATLAB

The demo adaptor class definitions are in *`matlabroot`*`\toolbox\daq\daqsdk\+daq\+demoadaptor`.

File	Description
`Session.m`	Defines `daq.demoadaptor.Session` class.
`VendorInfo.m`	Defines vendor driver class for `daq.getVendor`.

In addition to the files in this table, the demo adaptor also uses some of the class definition files in *`matlabroot`*`\toolbox\daq\daqsdk\+daq\+sdk`.

Executables

The following demo adaptor executables are in *`matlabroot`*`\toolbox\daq\daqsdk\bin\win64`.

File	Description
`DemoAdaptor.mexw64`	Built demo adaptor.
`daqasyncio.dll`	Accommodates streaming channel communication.
`daqmlconverter.dll`	Handles data type conversion.

See Also

Functions
`enableDemoAdaptorDiscovery`

Related Examples

- “Enable the Demo Adaptor” on page 2-5
- “Session Workflows with the Demo Adaptor” on page 2-6
- “Test the Demo Adaptor” on page 2-13

Enable the Demo Adaptor

By default, the demo adaptor is disabled when first installed. Use the following MATLAB command to enable it.

```
daq.sdk.utility.enableDemoAdaptorDiscovery
```

The adaptor is now ready for use. Confirm this with the command:

```
daq.getVendors
```

The output includes an entry for the demo adaptor with the vendor ID of `mw`:

```
index ID Operational         Comment
----- -- ----------- ----------------------
1     mw true        MathWorks
```

See Also

Functions
`enableDemoAdaptorDiscovery`

Related Examples

- “Demo Adaptor Description” on page 2-2
- “Session Workflows with the Demo Adaptor” on page 2-6
- “Test the Demo Adaptor” on page 2-13

Session Workflows with the Demo Adaptor

In this section...

Device Discovery and Configuration

When you create a data acquisition session, it applies to a specific vendor, and allows you to add applicable devices and channels. Discovery and configuration is part of setting up your session. This example shows a typical session setup with the demo adaptor.

Note To enable the demo adaptor in your installation, see the instructions in “Enable the Demo Adaptor” on page 2-5.

```
v = daq.getVendors
```

```
v =

Number of vendors: 2

index ID Operational         Comment
----- -- ----------- ------------------------
1     ni false       Click here for more info
2     mw true        MathWorks

Properties, Methods, Events

Additional data acquisition vendors may be available as
downloadable support packages.
Open the Support Package Installer to install additional vendors.
```

```
d = daq.getDevices
```

```
d =

Data acquisition devices:
```

```
index Vendor Device ID    Description
----- ------ --------- ------------------
1     mw     MWDev0    MathWorks MW314159
2     mw     MWDev1    MathWorks MW314159
3     mw     MWDev2    MathWorks MW628318
```

With a listing of available vendors and devices, you can create a session and add channels to it.

```
s = daq.createSession('mw')
```

```
s =

Data acquisition session using MathWorks hardware:
   Will run for 1 second (1000 scans) at 1000 scans/second.
   No channels have been added.

Properties, Methods, Events
```

To see details about one of the devices, use its index to access the array of devices.

```
d(1)
```

```
mw: MathWorks MW314159 (Device ID: 'MWDev0')
   Analog input subsystem supports:
      3 ranges supported
      Rates from 0.1 to 1000000.0 scans/sec
      2 channels ('ai0','ai1')
      'Voltage','Current' measurement types

   Analog output subsystem supports:
      3 ranges supported
      Rates from 0.1 to 1000000.0 scans/sec
      2 channels ('ao0','ao1')
      'Voltage','Current' measurement types
```

The first and second devices are the same model, so this example uses one for input (`'ai0'` and the other for output (`'ao0'`).

```
ch1 = addAnalogInputChannel(s,'MWDev0','ai0','voltage')
```

```
ch1 =

Data acquisition analog input voltage channel 'ai0' on device 'MWDev0':
```

```
       Coupling: DC
 TerminalConfig: Differential
          Range: -10 to +10 Volts
           Name: ''
             ID: 'ai0'
         Device: [1x1 daq.sdk.DeviceInfo]
MeasurementType: 'Voltage'

ch2 = addAnalogOutputChannel(s,'MWDev1','ao0','voltage')

ch2 =

Data acquisition analog output voltage channel 'ao0' on device 'MWDev1':

 TerminalConfig: SingleEnded
          Range: -10 to +10 Volts
           Name: ''
             ID: 'ao0'
         Device: [1x1 daq.sdk.DeviceInfo]
MeasurementType: 'Voltage'
```

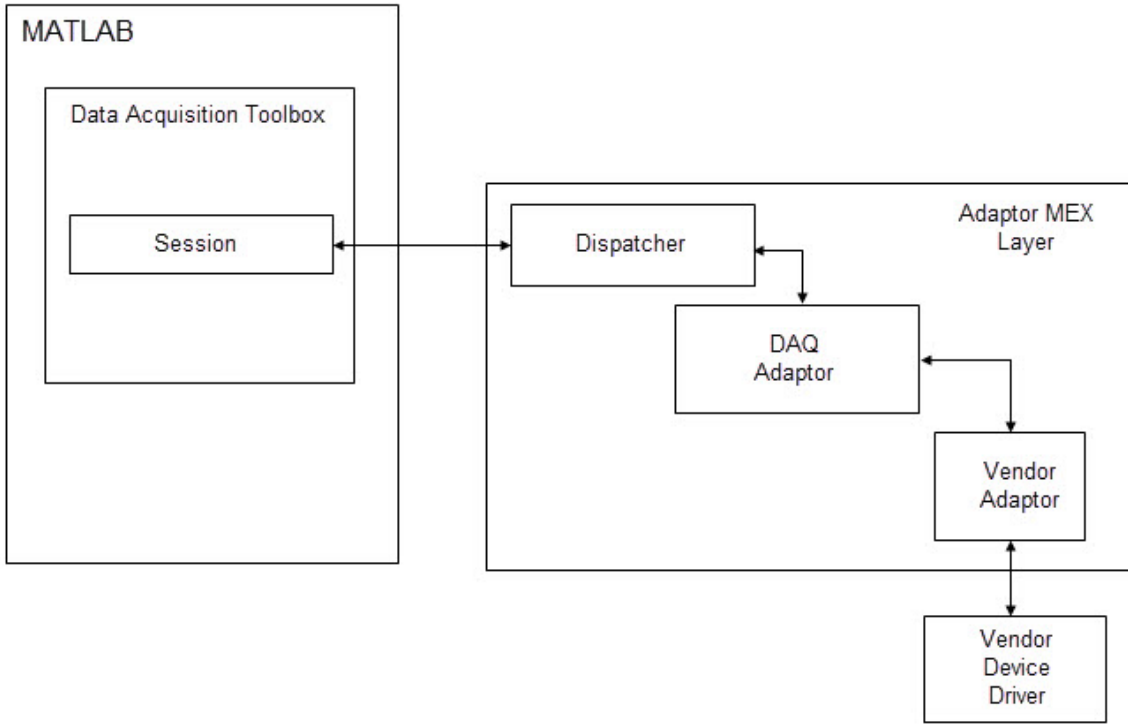
View the session to see the configuration.

```
s

s =

Data acquisition session using MathWorks hardware:
   No data queued.  Will run at 1000 scans/second.
   Number of channels: 2
      index Type Device Channel   MeasurementType        Range       Name
      ----- ---- ------ ------- ------------------- ----------------- ----
      1     ai   MWDev0 ai0     Voltage (Diff)      -10 to +10 Volts
      2     ao   MWDev1 ao0     Voltage (SingleEnd) -10 to +10 Volts
```

The session is now ready to send and receive single scans or streams of data.

Single Scan Input and Output

A single scan is when you send an output or read input from the channels at one moment in time. The data transfer is handled by the adaptor MEX layer. For the demo adaptor this is contained in the MEX-file *`matlabroot`*`\toolbox\daq\daqsdk\bin\win64\DemoAdaptor.mexw64`, which provides the functionality shown in the following diagram.



To generate a single scan analog output of 1.25 V, enter the following code.

```
outputSingleScan(s,1.25)
```

```
DemoDriver output: 1.25
```

To read a single scan of analog input:

```
data = inputSingleScan(s)
```

With only one input channel, this returns only a single value. If you add more analog input channels to the session, `inputSingleScan` returns a vector, with an element for each channel.

```
ch3 = addAnalogInputChannel(s,'MWDev2','ai0','voltage');
data = inputSingleScan(s)
```

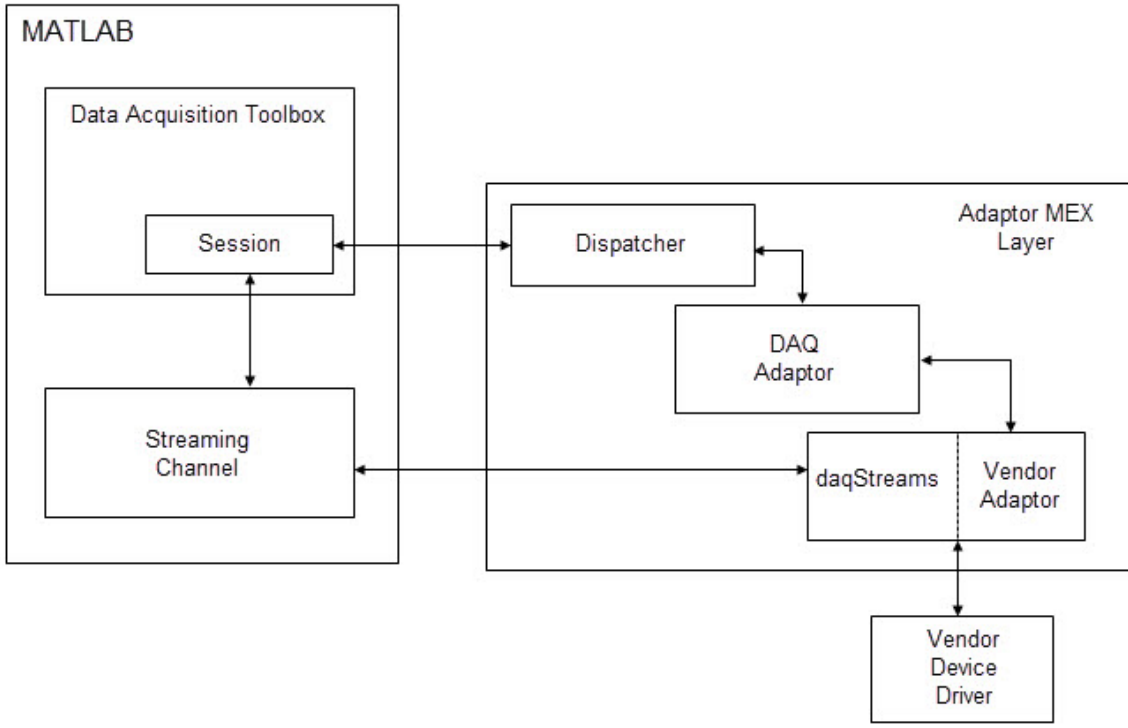
```
data =

     0   200
```

Remove channels from the session you no longer need.

```
removeChannel(s,[1 2 3]);
```

Streaming Input and Output

Streaming involves a sequence of input or output data on each channel, typically a waveform, comprised of many scans. Streaming can be accomplished in the foreground (blocking MATLAB until the stream is complete), or in the background (running asynchronously while MATLAB continues). Streaming might involve more scans or samples than the device memory can hold. For these reasons, the toolbox uses streaming channels to accommodate data flow. This allows data to be sent and received without causing a memory overflow, and without interrupting MATLAB.



Use MWDev2 to generate a 100 Hz sine wave in the background for 10 seconds. The default sample rate is 1000 scans per second; that amounts to 1000 cycles for 10,000 samples.

```
ch4 = addAnalogOutputChannel(s,'MWDev2','ao1','voltage')
```

```
ch4 =

Data acquisition analog output voltage channel 'ao1' on device 'MWDev2':

 TerminalConfig: SingleEnded
          Range: -10 to +10 Volts
           Name: ''
             ID: 'ao1'
         Device: [1x1 daq.sdk.DeviceInfo]
MeasurementType: 'Voltage'
```

```
Y = sin(linspace(0,2*pi*1000,10000))' % 1000 cycles for 10000 samples;
queueOutputData(s,Y);
s
```

```
s =

Data acquisition session using MathWorks hardware:
   Will run for 10000 scans (10 seconds) at 1000 scans/second.
   Number of channels: 1
      index Type Device Channel   MeasurementType        Range       Name
      ----- ---- ------ ------- ------------------- ---------------- ----
      1     ao   MWDev2 ao1     Voltage (SingleEnd) -10 to +10 Volts
```

The session display now indicates the number of queued scans, and how long it will run to output all the data. You can start the output.

```
startBackground(s)
s.IsRunning
```

```
  logical

   1
```

```
pause(10)
s.IsRunning
```

```
  logical

   0
```

See Also

Functions
`enableDemoAdaptorDiscovery`

Related Examples

- “Demo Adaptor Description” on page 2-2
- “Enable the Demo Adaptor” on page 2-5
- “Test the Demo Adaptor” on page 2-13
- “Channel Groups” on page 3-14

Test the Demo Adaptor

In this section...

Run Individual Tests

A collection of tests is available for testing functionality of the demo adaptor. These are all contained in the subfolders of *`matlabroot`*`\toolbox\daq\daqsdk\tests\+daq\+sdk\+tests`. Each test file name begins with the letter `t` and has the extension `.m`.

To get help and information on running an individual test, use the MATLAB help command with the full package and test name. For example, to learn about the test defined in *`matlabroot`*`\toolbox\daq\daqsdk\tests\+daq\+sdk\+tests\+workflow\tinputsinglescan.m`, type:

```
help daq.sdk.tests.workflow.tinputsinglescan
```

As indicated in the display help, you can run this test with the following commands:

```
t = daq.sdk.tests.workflow.tinputsinglescan;
results = run(t);
table(results)
```

```
Running daq.sdk.tests.workflow.tinputsinglescan
..
Done daq.sdk.tests.workflow.tinputsinglescan
__________

                             Name                                    Passed    Failed    Incomplete    Duration
    ______________________________________________________________    ______    ______    __________    ________

    'daq.sdk.tests.workflow.tinputsinglescan/verifyInputSingleScan'       true      false     false         1.7093
    'daq.sdk.tests.workflow.tinputsinglescan/verifyInputSingleScanLoop'   true      false     false         1.3631
```

Tip When modifying functionality in your custom adaptor, you should also modify the corresponding test. Be sure that the test runs as expected before moving on to your next modification.

Run a Test Suite

You can run all the tests in a package folder using the `runtests` function. For example, to run all the tests contained in `daq\+sdk\+tests\+workflow`, use the following commands:

```
results = runtests('daq.sdk.tests.workflow','Verbosity','Concise');
table(results)
```

```
                                                                Name                                                                 Passed    Failed    Incomplete
    _____________________________________________________________________________________________________________________________    ______    ______    __________

    'daq.sdk.tests.workflow.tbackground/verifyAnalogInputSession'                                                                    true      false     false
    'daq.sdk.tests.workflow.tbackground/verifyAnalogOutputSession'                                                                   true      false     false
    'daq.sdk.tests.workflow.tbackground/verifyAnalogInputContinuous'                                                                 true      false     false
    'daq.sdk.tests.workflow.tbackground/verifyAnalogOutputContinuous'                                                                true      false     false
.
.
```

To run a suite of tests that includes all subpackages of a specific package, use the `'IncludeSubpackages'` option in the `runtests` function call. The following code runs all tests below the `tests` package:

```
results = runtests('daq.sdk.tests','IncludeSubpackages',true,'Verbosity','Concise');
table(results)
```

Tip Run your complete modified test suite when all your individual updates are implemented and built.

See Also

Functions

`enableDemoAdaptorDiscovery` | `run` | `runtests`

Related Examples

- “Demo Adaptor Description” on page 2-2
- “Enable the Demo Adaptor” on page 2-5
- “Create Your Adaptor from the Demo Adaptor” on page 3-2
- “Modify Demo Tests for Your Adaptor” on page 3-8

Custom Adaptor Creation

Create Your Adaptor from the Demo Adaptor

Use the demo adaptor as a template for creating a custom adaptor which you can build, test, and access from the toolbox. The following sections provide a sequence of steps for adaptor modification. The examples in this topic create a custom adaptor named `MyAdaptor` with a vendor ID of `my`.

Edit and Build Your Adaptor

This section describes the step to make a new custom adaptor based on the shipped demo adaptor. This example modifies only the names of the adaptor, vendor, and devices, without any functional changes. You build the custom adaptor in a local folder, then add the build folders to the MATLAB path. This section uses two folder locations throughout:

Location	Description
`matlabroot`	MATLAB installation location. This is the MATLAB used both for the building of the adaptor, and for accessing the adaptor through a data acquisition session.
`C:\adaptors\daqsdk`	Local file location where the new adaptor is modified and built.

1. Create the build area in a location of your choice. This example works with a new folder, `C:\adaptors\daqsdk`. Create a subfolder here called `src`, and within that a subfolder named `daqadaptor`.
2. Copy the folder `DemoAdaptor` from *`matlabroot`*`\toolbox\daq\daqsdk\src\daqadaptor` into `C:\adaptors\daqsdk\src\daqadaptor`.
3. Inside `C:\adaptors\daqsdk\src\daqadaptor`, rename the folder `DemoAdapter` to be `MyAdaptor`.

   Navigate into `MyAdaptor`, and rename three of its files according to the following table:

Original Name	New Name
`custom_demo.cpp`	`custom_my.cpp`
`demoadaptor.cpp`	`myadaptor.cpp`

Original Name	New Name
`demoadaptor.hpp`	`myadaptor.hpp`

4 With a text editor, modify each of the three new files in the previous table, replacing all occurrences of text `DemoAdaptor`, `demoadaptor`, `DemoDriver`, and `custom_demo.cpp` with `MyAdaptor`, `myadaptor`, `MyDriver`, and `custom_my.cpp`, respectively, keeping the letter capitalization style with each replacement.

5 Further edit the contents of `myadaptor.cpp` as shown in the following table:

Original Text	Updated Text

`{`
`  shortName = "MW";`
`  fullName = "MathWorks";`
`  driverName = "DemoAdaptor";`
 	
`  return DAQSuccess;`
`}`
	`{`
	`  shortName = "MY";`
	`  fullName = "MyAdaptor";`
	`  driverName = "MyDriver";`
	
	`  return DAQSuccess;`
	`}`

`prefix = "MWDev";`	`prefix = "MyDev";`

`DAQStatus MyDriver::inputSingleScanImpl(`
`{`
`  deviceManager_->inputSingleScan(groupIndex, data);`
`  return DAQSuccess;`
`}`
	`DAQStatus MyDriver::inputSingleScanImpl(`
	`{`
	`  // deviceManager_->inputSingleScan(groupInd`
	`  data.push_back(1.125);`
	`  data.push_back(2.250);`
	`  return DAQSuccess;`
	`}`

The last row of this table causes the `inputSingleScanImpl` function to return hard data, rather than calling the driver function to read data.

With these modifications saved and in place, you are ready to build the adaptor.

6 In MATLAB, run the following utility to build the executable MEX-file for `MyAdaptor`:

```
daq.sdk.utility.mex.buildAdaptor('MyAdaptor','custom_my', ...
  'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor','C:\adaptors\daqsdk\bin\win64')
```

The function input arguments specify the adaptor name, source code file, source file location, and where to put the built output.

Note The `buildAdaptor` function requires that your system be configured with Microsoft® Visual Studio® 2013 or later.

---

7. Create the folder `C:\adaptors\daqsdk\+daq`, and copy into it the folder `+demoadaptor` found at *`matlabroot`*`\toolbox\daq\daqsdk\+daq\+demoadaptor`.
8. Navigate into `C:\adaptors\daqsdk\+daq`, and rename `+demoadaptor` to `+myadaptor`.
9. Navigate into `C:\adaptors\daqsdk\+daq\+myadaptor`, and edit these two MATLAB files in that folder:

   ```
   Session.m
   VendorInfo.m
   ```

   - In both of these files, replace all occurrences of the texts `DemoAdaptor` and `demoadaptor` with `MyAdaptor` and `myadaptor`, respectively, keeping the letter capitalization style with each replacement.
   - In the `VendorInfo` file, use `%` characters to comment out the lines that hide the adaptor, between the begin and end remove indicators. The change looks like this:

     ```
     % BEGIN REMOVE
     %   if daq.internal.getOptions().HideDAQSDKAdaptor
     %      throw(MException(message('daqsdk:HardwareInfo:VendorIsHidden', mfilename('class'))));
     %   end
     % END REMOVE
     ```

   - Save and close the files.

   Your modified adaptor is now ready for use.

Use Your Adaptor in a Session

This example shows how to access the vendor and device represented by your modified adaptor. A data acquisition session with your adaptor allows you to add channels and get information from the device.

Start MATLAB, and use the following commands to make your adaptor available.

```
addpath 'C:\adaptors\daqsdk\bin\win64'
addpath 'C:\adaptors\daqsdk'
```

Then you can access your adaptor.

```
v = daq.getVendors

v =

Number of vendors: 2
```

```
index     ID      Operational         Comment
----- ----------- ----------- ------------------------
1     ni          false       Click here for more info
2     my          true        MyAdaptor
```

Use the index of the vendor ID `my` to get more information.

```
vendor = v(2)

vendor =

Data acquisition vendor 'MyAdaptor':

            ID: 'my'
      FullName: 'MyAdaptor'
AdaptorVersion: '3.13 (R2018a)'
DriverVersion: '1.0.0'
IsOperational: true
```

Create a session for your device.

```
s = daq.createSession('my')

s =

Data acquisition session using MyAdaptor hardware:
   Will run for 1 second (1000 scans) at 1000 scans/second.
   No channels have been added.
```

Add an analog input channel to the session, associated with the device `MyDev0`, channel `ai0`.

```
ch1 = addAnalogInputChannel(s,'MyDev0','ai0','Voltage')

ch1 =

Data acquisition analog output voltage channel 'ao0' on device 'MyDev0':

TerminalConfig: SingleEnded
          Range: -10 to +10 Volts
           Name: ''
             ID: 'ao0'
         Device: [1x1 daq.sdk.DeviceInfo]
MeasurementType: 'Voltage'
```

Add a second analog input channel.

```
ch2 = addAnalogInputChannel(s,'MyDev0','ai1','Voltage');
```

View the session to see the channel configurations.

```
s
```

```
s =

Data acquisition session using MyAdaptor hardware:
   Will run for 1 second (1000 scans) at 1000 scans/second.
   Number of channels: 2
      index Type Device Channel MeasurementType      Range       Name
      ----- ---- ------ ------- --------------- ---------------- ----
      1     ai   MyDev0 ai0     Voltage (Diff)  -10 to +10 Volts
      2     ai   MyDev0 ai1     Voltage (Diff)  -10 to +10 Volts
```

Examine the objects so far in the base workspace.

```
whos
```

```
   Name        Size            Bytes  Class                                Attributes

  ch1         1x1                 8  daq.sdk.AnalogInputVoltageChannel
  ch2         1x1                 8  daq.sdk.AnalogInputVoltageChannel
  s           1x1                 8  daq.myadaptor.Session
  v           1x2                16  daq.VendorInfo
  vendor      1x1                 8  daq.myadaptor.VendorInfo
```

With the example data hard coded into the adaptor `inputSingleScanImpl` function, you can execute a single scan measurement on the session channels.

```
data = inputSingleScan(s)
```

```
data =

    1.1250    2.2500
```

You can also read streaming input data, in this case provided by the demo adaptor `DAQstream` object. The session default configuration captures 1000 scans in 1 second.

```
stdata = startForeground(s);
whos stdata
```

```
  Name           Size             Bytes  Class     Attributes

  stdata      1000x2              16000  double
```

`stdata` contains a column of 1000 samples for each channel. View the first six rows.

```
stdata(1:6,:)
```

```
         0    0.2500
    0.2487    0.4987
    0.4818    0.7318
    0.6845    0.9345
```

```
    0.8443    1.0943
    0.9511    1.2011
```

When you are finished, delete the session and clear the objects.

```
delete(s)
clear v vendor s ch1 ch2
```

See Also

Functions
`buildAdaptor`

Related Examples

- “Modify Demo Tests for Your Adaptor” on page 3-8

More About

- “Adaptor Creation Summary” on page 1-5
- “Adaptor Functions for a Data Acquisition Session” on page 3-23

Modify Demo Tests for Your Adaptor

This topic describes how to copy demo adaptor tests and modify them for use with your own adaptor. The steps below assume you have an adaptor called `MyAdaptor`, as created in the example of “Create Your Adaptor from the Demo Adaptor” on page 3-2.

1. Copy *`matlabroot`*`\toolbox\daq\daqsdk\tests` to `C:\adaptors\daqsdk\tests`
2. In a file browser, navigate to the SDK tests package folder `C:\adaptors\daqsdk\tests\+daq\+sdk`.
3. Rename the folder `+tests` to `+mytests`.

   The next steps require you to edit and save your test files. You can use the MATLAB editor, or any editor of your choice. Because the tests are MATLAB files, using the MATLAB editor is recommended for debugging purposes.
4. Navigate to `C:\adaptors\daqsdk\tests\+daq\+sdk\+mytests`, and open the file `hardwareconfiguration.m`. In MATLAB you can navigate to its folder and open the editor:

   ```
   cd ('C:\adaptors\daqsdk\tests\+daq\+sdk\+mytests')
   edit hardwareconfiguration
   ```

   Change the vendor and device parameters in this manner, using your own names.

Original Text	Updated Text

`% HardwareInfo`
`  VendorName = 'mw';`
`  VendorFullName = 'MathWorks';`
 	
`  DeviceID1 = 'MWDev0';`
`  DeviceID2 = 'MWDev1';`
`  DeviceID3 = 'MWDev2';`
	`% HardwareInfo`
	`  VendorName = 'my';`
	`  VendorFullName = 'MyAdaptor';`
	
	`  DeviceID1 = 'MyDev0';`
	`  DeviceID2 = 'MyDev1';`
	`  DeviceID3 = 'MyDev2';`

5. Save and close the file.

   The updated vendor information now allows your tests to run on your adaptor.
6. Modify all files in `C:\adaptors\daqsdk\tests\+daq\+sdk\+mytests\+workflow\` so that all lines use `mytests` instead of `tests`. For example,

   ```
   classdef tbackground < daq.sdk.mytests.workflow.BaseDAQSessionWorkflowTester
   ```

7 Restart MATLAB. Use the following commands to add your adaptor and tests to the command path.

```
addpath 'C:\adaptors\daqsdk\bin\win64'
addpath 'C:\adaptors\daqsdk'
addpath 'C:\adaptors\daqsdk\tests'
```

Run the test for single scan inputs.

```
t = daq.sdk.mytests.workflow.tinputsinglescan;
results = run(t);
table(results)
```

```
Running daq.sdk.mytests.workflow.tinputsinglescan
..
Done daq.sdk.mytests.workflow.tinputsinglescan
__________


ans =

  2×6 table

                                    Name                                     Passed    Failed    Incomplete    Durati
    ____________________________________________________________________    ______    ______    __________    ______

    'daq.sdk.mytests.workflow.tinputsinglescan/verifyInputSingleScan'       true      false     false          1.691
    'daq.sdk.mytests.workflow.tinputsinglescan/verifyInputSingleScanLoop'   true      false     false         1.2947
```

8 For streaming tests, there are three files to modify in the folder `C:\adaptors\daqsdk\tests\+daq\+sdk\+mytests\+development\+streaming`.

Modify `tstreambasic.m` using your own vendor and device information, as follows:

Original Text	Updated Text

`properties(TestParameter)`
`VendorName = {'mw'}; % Add vendor ad`
`DeviceID = {'MWDev1'}; % Add device`
`end`
	`properties(TestParameter)`
	`VendorName = {'my'}; % Add vendor ad`
	`DeviceID = {'MyDev1'}; % Add device`
	`end`

Modify both `tstreamread.m` and `tstreamwrite.m` using your own adaptor information, as follows:

Original Text	Updated Text

`properties (ClassSetupParameter)`
`adaptorName = {'DemoAdaptor'}`
`end`
	`properties (ClassSetupParameter)`
	`adaptorName = {'MyAdaptor'}`
	`end`

Original Text	Updated Text
adaptorPath = fullfile(matlabroot, ... 'toolbox','daq','daqsdk','bin',computer('arch'));	adaptorPath = 'c:\adaptors\daqsdk\bin\win64'

9 You can now run any of the streaming tests on your adaptor. For example, restart MATLAB and enter the following code:

```
addpath 'C:\adaptors\daqsdk\bin\win64'
addpath 'C:\adaptors\daqsdk'
addpath 'C:\adaptors\daqsdk\tests'
t = daq.sdk.mytests.development.streaming.tstreamread;
results = run(t);
table(results)
```

```
Running daq.sdk.mytests.development.streaming.tstreamread
........
Done daq.sdk.mytests.development.streaming.tstreamread
__________


ans =

  8×6 table

                                                                           Name
    _____________________________________________________________________________________________________________________

    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIFiniteRead(scanRate=value1,num(
    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIFiniteRead(scanRate=value1,num(
    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIFiniteRead(scanRate=value2,num(
    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIFiniteRead(scanRate=value2,num(
    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIContinuousRead(scanRate=value1
    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIContinuousRead(scanRate=value1
    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIContinuousRead(scanRate=value2
    'daq.sdk.mytests.development.streaming.tstreamread[adaptorName=MyAdaptor]/VerifyAIContinuousRead(scanRate=value2
```

Further Suggestions

Run Test Suites

You can run the full suite of tests for your adaptor by specifying the package folder to use all the tests contained in it.

```
results = runtests('daq.sdk.mytests','IncludeSubpackages',true,'Verbosity','Concise');
table(results)
```

Modify Functionality Tests

- As you write your adaptor, you must modify the test files to correspond to the functionality implemented for your device. In deciding the sequence in which you

implement and test functionality, consider “Session Workflows with the Demo Adaptor” on page 2-6.

- In addition to adaptor name, you must modify where applicable the vendor name, device driver name, device names, vendor ID, etc.

See Also

Functions
`run` | `runtests`

Related Examples

- “Test the Demo Adaptor” on page 2-13
- “Create Your Adaptor from the Demo Adaptor” on page 3-2

Errors and Exceptions

Nonstreaming

To indicate that a standard SDK function has resulted in an error for an expected reason, return the appropriate error code (as opposed to `DAQSuccess`), as provided in `include/daqsdktypes.h`.

To indicate that a custom SDK function has resulted in an error, throw a `DAQDiagnostic` (see `daqinterfaces.hpp`) containing a custom error code and a diagnosis message string.

To indicate that a standard SDK function has resulted in an error for a reason that is specific to the function of the custom adaptor (vendor-specific error), throw a `DAQDiagnostic`.

You can define a custom error code as a negative value less than `daqsdk::DAQErr_ReservedRangeEnd` (see `daqsdktypes.h`). while a custom warning code can be defined as a positive value greater than `daqsdk::DAQWrn_ReservedRangeEnd`.

Streaming

To indicate that an error has occurred during the configuration of the stream (`configureStream`), return a custom error code. You should also implement the `DAQStreamAnalog::getDiagnosticFromStatus`, which when given a custom error code, returns a string describing the error condition.

To indicate that an error has occurred during streaming (that is, after the stream has started but before it is done or has stopped), return a custom error code or throw an exception.

See Also

Related Examples

- “Create Your Adaptor from the Demo Adaptor” on page 3-2

More About

- “Adaptor API Reference” on page 4-2
- “Streaming API Reference” on page 4-23
- “Sequence for Errors and Exceptions” on page 5-21

Channel Groups

In this section...

Channel Group Description

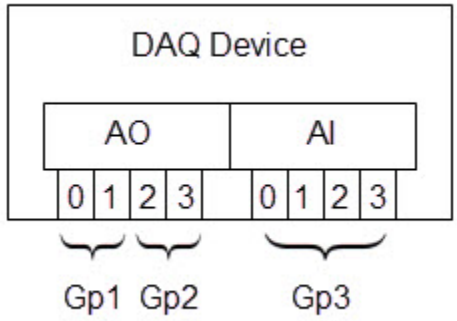
A simplified view of a DAQ device is that actual devices provide channels with common functions, logically grouped into subsystems. For example, all channels that provide analog input data may be thought of as belonging to an analog input subsystem.

Another view of this DAQ device is as a provider and consumer of data. The largest unit of data that can be acquired or generated simultaneously, by one or more channels, is a scan. The logical grouping that acquires or consumes one or more scans of data is a channel group. The definition of the channel group is usually constrained by the driver and hardware. For example, when all channels belonging to a single analog input subsystem also share a single clock.

A *channel group* is an aggregation of channels, usually of the same subsystem, which operate together. For example, all the analog output subsystem channels on a device must be configured, reserved, and act together to generate data as a single scan.

You should define channel groups in a way that reflects the driver constraints, and provide a means for identifying all channels belonging to the group for acquiring and generating scans. Typically, channel groups provide functions to stream data to or from a device buffer.

Each channel on the device has a unique address, defined by device, subsystem, and channel ID. Each channel must be assigned to one channel group. The following diagram illustrates one possible channel group arrangement. In this scenario, analog output (AO) channels 0 and 1 might serve a different purpose than AO channels 2 and 3; while analog input (AI) channels 0–3 are used all together.



For reference information on the functions used in configuring channel groups, see “Hardware Management” on page 4-7.

Channel Group Restrictions

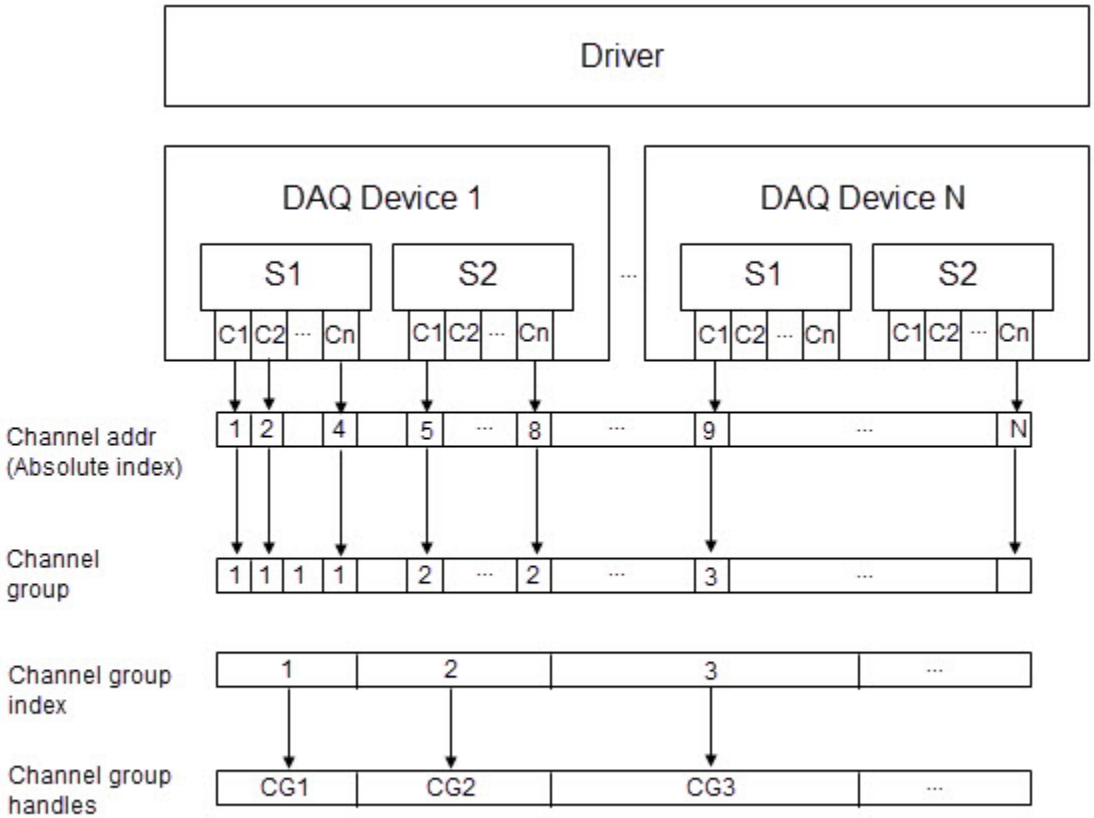
- All channels in a group operate together. This allows synchronized streaming to the extent supported by the hardware.
- A channel cannot belong to more than one group.
- All channels in a group are requested, reserved, and released together.
- By default, a channel group—and therefore all its channels—can be accessed by only one data acquisition session at a time. You cannot add channels to a session if any other channels in their groups are already added to a different session. If your driver allows a group to be accessed by different sessions, you can control this behavior using the `isRegistrationReservationImpl` function.

Device Discovery

Device discovery occurs by calling `daq.getDevices` in a MATLAB session. Part of discovery is enumeration, whereby all devices and channels are indexed. The result of enumeration is a set of channel group handles, which the adaptor uses to address channels on the numerous devices of the session.

A channel group usually includes all the channels of one subsystem of one device, as shown in the following diagram. But other configurations are possible. For example, a

channel group could include all channels of all subsystems in a single device, or all channels of the same type of subsystem across several devices.



This diagram illustrates the process of enumeration performed during `daq.getDevices`. Through the driver, the adaptor accesses the numerous supported devices, and determines their subsystems and channels. The adaptor then derives an absolute (unique) index for each channel, and assigns each to a channel group. Each channel group has an index, and a resulting unique channel group handle. Through these handles, the adaptor performs the operations of a data acquisition session.

You can create handles to any of the possible objects in your configuration, such as devices, channels, and subsystems, but the adaptor templates provided with Data Acquisition Toolbox use only channel group indices.

See Also

More About

- “Vendor Adaptor Templates” on page 3-20
- “Adaptor API Reference” on page 4-2
- “Streaming API Reference” on page 4-23
- “Streaming Input and Output” on page 2-10
- “Streaming Sequence Diagrams” on page 5-4

Custom Functions

Custom functionality provided by your adaptor that is not part of the standard session-based interface can be exposed to MATLAB via the DAQ SDK custom interface. For example, your device might provide an on-board power supply.

Note This topic assumes experience writing MEX-files.

To add a custom function, first review the custom functions available in the demo adaptor in the folder *`matlabroot`*`\matlab\toolbox\daq\daqsdk\+daq\+demoadaptor\+custom`. The installed files in this folder are:

```
testHasInputsHasOutputs.m
testHasInputsNoOutputs.m
testNoInputsNoOutputs.m
testThrowCustomExceptions.m
```

Use these steps to create your own custom function:

1. Add a function to the `MyDriver` class (`MyDriver::customFunction`).
2. Add a function to the `MyAdaptor` class (`MyAdaptor::customFunction`) that calls `MyDriver::customFunction` with the designated:
   - Inputs
   - Outputs
   - Custom error code
3. Update `custom_my.cpp` to:
   - Define a function to call (dispatch) the custom adaptor function `MyAdaptor::customFunction`.
   - Update the `customizeMap` function to add:

     ```
     functionMap["myCustomFunction"] = customFunction;
     ```

     where `customFunction` is the name of the MEX-function that calls `MyAdaptor::customFunction`, and `myCustomFunction` is the name of function in MATLAB.
4. Define your custom MATLAB function `myCustomFunction.m` in the `+daq\+myadaptor\+custom` subpackage for your adaptor.

- Choose the appropriate template from `\+daq\+demoadaptor\+custom\` to copy and rename.
  - Has Inputs, Has Outputs
  - Has Inputs, No Outputs
  - No Inputs, No Outputs
- Rename the file to perform the desired function, for example, `myCustomFunction.m`.
- Edit `myCustomFunction.m` to
  - Update: `functionName = 'myCustomFunction';`
  - Provide the inputs to the function as a structure.

For functionality that is not part of the standard session interface, contact MathWorks® technical support at `https://www.mathworks.com/support/contact_us` to let us know what functionality you need.

See Also

More About

- "C Matrix API" (MATLAB)
- "Vendor Adaptor Templates" on page 3-20
- "Errors and Exceptions" on page 3-12
- "Deliver Your Adaptor" on page 3-22

Vendor Adaptor Templates

The Data Acquisition Toolbox SDK is installed with a set of source file stubs for an adaptor called `VendorAdaptor`. The source files are installed in the folder:

matlabroot`\toolbox\daq\daqsdk\src\daqadaptor\VendorAdaptor`

The basic components and structure of the source files for this adaptor are the same as those in the `DemoAdaptor`. If you need to create an adaptor from scratch, it is recommended that you use a copy of the `VendorAdaptor` source files. The following table indicates the purpose of each source file.

File	Description
`vendoradaptor.hpp`, `vendoradaptor.cpp`	Wraps device driver code in methods that allow you to configure, discover, and enumerate the hardware in MATLAB.
`daqstream_analog.hpp`, `daqstream_analog.cpp`	Implements a DAQStream object for an analog subsystem, that allows you to stream data to and from the hardware.
`custom_vendor.cpp`	Dispatches custom calls from MATLAB to the adaptor. At a minimum this must contain a `customizeMap` function.

Typical Workflow to Create Adaptor

To create an adaptor from the set of template files in the folder `VendorAdaptor`, use the following steps. Assume that you want to name the adaptor `MyAdaptor`.

1. Create a copy of the entire folder, and name it `MyAdaptor`.
2. Working in the new folder called `MyAdaptor`, change the names of the files:

Original Name	New Name
`vendoradaptor.hpp`	`myadaptor.hpp`
`vendoradaptor.cpp`	`myadaptor.cpp`
`custom_vendor.cpp`	`custom_my.cpp`

3. Update the content of the files so that the new names are used for references to other files, the adaptor, devices, and vendor.

4 Update the functions to use your driver code. For more information, see “Adaptor API Reference” on page 4-2.

5 Build the adaptor with the `buildAdaptor` function.

See Also

Related Examples

- “Create Your Adaptor from the Demo Adaptor” on page 3-2

More About

- “Adaptor Creation Summary” on page 1-5
- “Custom Functions” on page 3-18

Deliver Your Adaptor

When you have a custom adaptor for delivery, you can create a toolbox to deliver the adaptor MEX-file along with your device driver and any other support files, such as documentation, data files, examples, and so on.

You should document any adaptor behavior that involves:

- Differences from standard data acquisition session behavior
- Custom functions

For information on creating and distributing a toolbox, see “Toolbox Distribution” (MATLAB).

For creating and delivering documentation to support your toolbox, see “Display Custom Documentation” (MATLAB).

For your custom examples, see “Display Custom Examples” (MATLAB).

Adaptor Functions for a Data Acquisition Session

In this section...

This topic lists the adaptor and streaming functions that need to be implemented for each stage and operation of a session lifetime.

Device Discovery

Device discovery is accomplished with the `daq.getDevices` function. Implement the following adaptor functions for this task.

Adaptor Functions	Notes

`initImpl`
`enumerateDevicesImpl`
`commitDevicesImpl`
`getOrderOfChannelAdditionImpl`
	Devices — Identify devices for driver.

`enumerateSubsystemsImpl` `commitSubsystemsImpl`	**Device Subsystems** — Repeated for each device.
`enumerateChannelsImpl` `commitChannelsImpl`	**Subsystem Channels** — Repeated for each subsystem of each device.
`getChannelGroupIndexImpl`	**Channel Index** — Repeated for each channel of each subsystem.
`getVendorInfoImpl` `getDriverVersionImpl`	**Vendor Information** — Get the vendor information and driver information.
`getDeviceInfoImpl` `getFirmwareVersionImpl`	**Device Information** — Get the device information. Repeated for each device.

Adaptor Functions	Notes

`getSubsystemsOfTypeImpl`
`getMeasurementTypesImpl`
`getDefaultMeasurementTypeImpl`
`getCouplingsImpl`
`getDefaultCouplingImpl`
`getSampleTypesImpl`
`getDefaultSamplingTypeImpl`
`getNativeDataTypeImpl`
`getRateLimitImpl`
`getResolutionImpl`
`getTerminalConfigsImpl`
`getRangesAvailableForTerminalConfigImpl`
`getRangesAvailableForTerminalConfigImpl`
`getDefaultTerminalConfigImpl`
`isOnDemandOperationSupportedImpl`
`getChannelNamesImpl`
	Analog Input — Repeated for each device with an analog input subsystem.

`getSubsystemsOfTypeImpl`
`getMeasurementTypesImpl`
`getDefaultMeasurementTypeImpl`
`getNativeDataTypeImpl`
`getRateLimitImpl`
`getResolutionImpl`
`getTerminalConfigsImpl`
`getRangesAvailableForTerminalConfigImpl`
`getDefaultTerminalConfigImpl`
`isOnDemandOperationSupportedImpl`
`getChannelNamesImpl`
	Analog Output — Repeated for each device with an analog output subsystem.

`getSubsystemsOfTypeImpl`
`getMeasurementTypesImpl`
`getDefaultMeasurementTypeImpl`
`getRateLimitImpl`
`isOnDemandOperationSupportedImpl`
`getDigitalChannelTypesImpl`
`getChannelNamesImpl`
	Digital Input/Output — Repeated for each device with a digital input/output subsystem.

Session Configuration and Single Scan Operation

The session configuration controls which devices and channel settings you use for data input and output. For each of the following data acquisition session functions, implement the corresponding adaptor functions.

Session Function	Adaptor Functions	Notes

`addAnalogInputChannel`
 	
`addAnalogOutputChannel`
 	
`addDigitalChannel`
	`addChannelImpl`
	`getGroupRateLimitsImpl`
	`setRateImpl`
	`getRateImpl`
		Repeated for each channel added to the session.

`removeChannel`	`removeChannelImpl` `getGroupRateLimitsImpl`	

`session.Rate`	`unreserveChannelGroupImpl`
	`setRateImpl`
	`getRateImpl`
		Set the session `Rate` property value.

`inputSingleScan`	`inputSingleScanImpl`	
`outputSingleScan`	`outputSingleScanImpl`	
`prepare`	`isDeviceAvailableImpl` `reserveChannelGroupImpl`	
`release`	`unreserveChannelGroupImpl`	

Streaming

Streaming uses DAQStream objects for transferring data between the session and the device driver. The session configuration is necessary to support streaming.

Implement the following functions for the adaptor or stream objects, as indicated in the notes.

Session Function	Source Functions	Notes
`queueOutputData`		No adaptor stream function.

Session Function	Source Functions	Notes

`startForeground`
 	
`startBackground`
	`makeStream`
	`initialize`
	`terminate`
	`configureStream`
	`unconfigureStream`
	`registerCallbacks`
	`unregisterCallbacks`
	`prestart`
	`start`
	`stop`
	`getNumInputScansAvailable`
	`getNumScansOutputByHardware`
	`getOutputBufferSize`
	`flushOutputBuffer`
	`isDeviceDone`
	`read`
	`write`
	`readWrite`
		Implemented in `daqstream*` code. See "Streaming API Reference" on page 4-23 and "Streaming Sequence Diagrams" on page 5-4.

`stop`	`stop`	Session function used for stopping background operation.

Session Reset

Session Function	Adaptor Functions	Notes
`daqreset`	`releaseChannelsImpl`	Repeated for all channels.
	`releaseSubsystemsImpl`	Repeated for all subsystems.
	`releaseDevicesImpl`	Repeated for all devices.
	`termImpl`	Terminate sessions.

See Also

More About

- "Create Your Adaptor from the Demo Adaptor" on page 3-2

- “Adaptor API Reference” on page 4-2
- “Streaming API Reference” on page 4-23

API Reference

Adaptor API Reference

This topic provides an overview of each function included in the demo adaptor source file, `demoadaptor.cpp`, grouped in the following categories. The `vendoradaptor.cpp` template includes similar functions.

In this section...

Lifetime

Lifetime functions include those that involve the loading and unloading of the driver interface.

initImpl

Syntax	`DAQStatus DemoDriver::initImpl()`
Purpose	Initialize and load implementation of `daqsdk::IDriver` interface
Inputs	None
Output	None
Return status	`DAQErr_Driver_Init` on failure. `DAQSuccess` on success.

termImpl

Syntax	`DAQStatus DemoDriver::termImpl()`

Purpose	Terminate and unload implementation of `daqsdk::IDriver` interface
Inputs	None
Output	None
Return status	`DAQErr_Driver_Term` on failure. `DAQSuccess` on success.

Enumeration

Enumeration functions involve the recognition of devices, subsystems, and channels.

enumerateDevicesImpl

Syntax	`DemoDriver::enumerateDevicesImpl(Index &deviceCount) const`
Purpose	Enumerate devices available via vendor driver
Inputs	None
Output	Number of the devices enumerated
Return status	`DAQErr_Driver_EnumerateDevices` on failure. `DAQSuccess` on success.

commitDevicesImpl

Syntax	`DemoDriver::commitDevicesImpl(Index deviceCount)`
Purpose	Inform the driver that the enumerated devices are to be committed, in enumerated order, for use by the adaptor
Inputs	Number of devices enumerated
Output	None
Return status	`DAQErr_Driver_CommitDevices` on failure. `DAQSuccess` on success.

enumerateSubsystemsImpl

Syntax	`DemoDriver::enumerateSubsystemsImpl(Index deviceIndex, Index &subsystemCount) const`
Purpose	Enumerate the subsystems available via for a given device
Inputs	Index of the given device
Output	Number of the subsystems enumerated
Return status	`DAQErr_Driver_EnumerateSubsystems` on failure. `DAQSuccess` on success.

commitSubsystemsImpl

Syntax	`DemoDriver::commitSubsystemsImpl(Index deviceIndex, Index subsystemCount)`
Purpose	Inform the driver that the enumerated subsystems, for a given device, are to be committed, in enumerated order, for use by the adaptor
Inputs	Index of the given device, number of the subsystems enumerated
Output	None
Return status	`DAQErr_Driver_CommitSubsystems` on failure. `DAQSuccess` on success.

enumerateChannelsImpl

Syntax	`DemoDriver::enumerateChannelsImpl(Index deviceIndex, Index subsystemIndex, Index &channelCount) const`
Purpose	Enumerate the channels available via for a given subsystem and device
Inputs	Index of the given device, the index of the given subsystem
Output	Number of the channels enumerated
Return status	`DAQErr_Driver_EnumerateChannels` on failure. `DAQSuccess` on success.

commitChannelsImpl

Syntax	`DemoDriver::commitChannelsImpl(Index deviceIndex, Index subsystemIndex, Index channelCount)`
Purpose	Inform the driver that the enumerated channels, for a given device and subsystem, are to be committed, in enumerated order, for use by the Adaptor
Inputs	Index of the given device, the index of the given subsystem, the number of channels enumerated
Output	None
Return status	`DAQErr_Driver_CommitChannels` on failure. `DAQSuccess` on success.

getChannelGroupIndexImpl

Syntax	`DemoDriver::getChannelGroupIndexImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, Index &channelGroupIndex) const`
Purpose	Return the channel group index corresponding to a specified channel
Inputs	Index of the specified device, index of the specified subsystem, index of the specified channel
Output	Channel group index
Return status	`DAQErr_Driver_GetChannelGroupIndex` on failure. `DAQSuccess` on success.

determineOrderOfChannelAdditionImpl

Syntax	`DemoDriver::getOrderOfChannelAdditionImpl(daqsdk::OrderOfChannelsInGroup &orderOfChannelsInGroup) const`
Purpose	Return an enumeration representing the order in which channels indices are sorted, by the driver, in channel groups
Inputs	None
Output	Order of channels within channel groups

Return status	`DAQErr_Driver_OrderOfChannelAddition` on failure. `DAQSuccess` on success.

Channel groups contain a list of channels ordered first by device, then by subsystem, and finally by channel. The group must acquire data from requested channels either in the listed order ("Sorted") or in the order requested ("InOrderOfAddition"). For example, if the group contains four channels and a user requests channels 4, 2, and 1, they should expect data from the channel group either in the order 4, 2, 1 (the order in which the channels were added) or in the order 1, 2, 4 (sorted). See “Channel Groups” on page 3-14.

releaseDevicesImpl

Syntax	`DemoDriver::releaseDevicesImpl(Index deviceIndex)`
Purpose	Release the resources committed by the driver for a specified device
Inputs	Index of the device resources to release
Output	None
Return status	`DAQErr_Driver_ReleaseDevices` on failure. `DAQSuccess` on success.

releaseSubsystemsImpl

Syntax	`DemoDriver::releaseSubsystemsImpl(Index deviceIndex, Index subsystemIndex)`
Purpose	Release the resources committed by the driver for a specified subsystem and device
Inputs	Index of the device resources to release, index of the subsystem resources to release
Output	None
Return status	`DAQErr_Driver_ReleaseSubsystems` on failure. `DAQSuccess` on success.

releaseChannelsImpl

Syntax	`DemoDriver::releaseChannelsImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex)`
Purpose	Release the resources committed by the driver for a specified channel of a subsystem of a device
Inputs	Index of the device resources to release, index of the subsystem resources to release, index of the channel resources to release
Output	None
Return status	`DAQErr_Driver_ReleaseChannels` on failure. `DAQSuccess` on success.

Hardware Management

Hardware management functions control the configuration of channel groups.

addChannelImpl

Syntax	`DemoDriver::addChannelImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex)`
Purpose	Register the specified channel with its channel group
Inputs	Index of the device, index of the subsystem for given device, index of the channel for the given subsystem
Output	None
Return status	`DAQErr_Driver_AddChannel` on failure. `DAQSuccess` on success.

removeChannelImpl

Syntax	`DemoDriver::removeChannelImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex)`
Purpose	Unregister the specified channel from its channel group
Inputs	Index of the device, index of the subsystem for given device, index of the channel for the given subsystem

Output	None
Return status	`DAQErr_Driver_RemoveChannel` on failure. `DAQSuccess` on success.

reserveChannelGroupImpl

Syntax	`DemoDriver::reserveChannelGroupImpl(ChannelGroupIndex groupIndex)`
Purpose	Reserve the specified channel group and all its resources
Inputs	Index of the channel group
Output	None
Return status	`DAQErr_Driver_ReserveChannelGroup` on failure. `DAQSuccess` on success.

unreserveChannelGroupImpl

Syntax	`DemoDriver::unreserveChannelGroupImpl(ChannelGroupIndex groupIndex)`
Purpose	Unreserve/release the specified channel group and all its resources
Inputs	Index of the channel group
Output	None
Return status	`DAQErr_Driver_UnreserveChannelGroup` on failure. `DAQSuccess` on success.

Vendor and Device Discovery

These functions retrieve information about vender and device.

getDriverVersionImpl

Syntax	`DemoDriver::getDriverVersionImpl(uint32_T &major, uint32_T &minor, uint32_T &patch) const`
Purpose	Return driver version number

Inputs	None
Output	Major version number, minor version number, patch version number
Return status	`DAQErr_Driver_GetDriverVersion` on failure. `DAQSuccess` on success.

getVendorInfoImpl

Syntax	`DemoDriver::getVendorInfoImpl(std::string &shortName, std::string &fullName, std::string &driverName) const`
Purpose	Return relevant vendor information (name and driver-name)
Inputs	None
Output	Vendor shortname (typically used as a vendor ID), vendor fullname, driver name (including full path)
Return status	`DAQErr_Driver_GetVendorInfo` on failure. `DAQSuccess` on success.

getDeviceInfoImpl

Syntax	`DemoDriver::getDeviceInfoImpl(Index deviceIndex, std::string &model, std::string &prefix, std::string &id, std::string &serialNumber, bool &isRecognizedDevice) const`
Purpose	Return relevant device information
Inputs	Index of the device
Output	Device model, device prefix (e.g., `'Dev'`, `'Audio'`, etc.), device ID, device serial number, indication of whether the driver recognizes and supports the device
Return status	`DAQErr_Driver_GetDeviceInfo` on failure. `DAQSuccess` on success.

getFirmwareVersionImpl

Syntax	`DemoDriver::getFirmwareVersionImpl(Index deviceIndex, uint32_T &major, uint32_T &minor, uint32_T &patch) const`
Purpose	Return firmware version number
Inputs	None
Output	Major version number, minor version number, patch version number
Return status	`DAQErr_Driver_GetFirmwareVersion` on failure. `DAQSuccess` on success.

Subsystem Discovery

These functions retrieve information about the subsystem.

getSubsystemsOfTypeImpl

Syntax	`DemoDriver::getSubsystemsOfTypeImpl(Index deviceIndex, IndexList &subsystemIndices, daqsdk::Subsystem subsystemType, daqsdk::TransferDirection transferDirection) const`
Purpose	Return subsystems of a given type (Analog, Digital, etc.) and direction (Input, Output)
Inputs	Index of the device, subsystem type, transfer direction
Output	List of subsystem indices with the given type/direction or empty if no matches are found
Return status	`DAQErr_Driver_GetSubsystemsOfType` on failure to execute the request (but not when no subsystems are found). `DAQSuccess` on success.

getMeasurementTypesImpl

Syntax	`DemoDriver::getMeasurementTypesImpl(Index deviceIndex, Index subsystemIndex, std::vector<daqdatatypes::MeasurementType> &measurementTypes) const`
Purpose	Return the measurement types supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	List of measurement types supported by the specified subsystem
Return status	`DAQErr_Driver_GetMeasurementTypes` on failure to execute the request (but not when no subsystems are found). `DAQSuccess` on success.

getDefaultMeasurementTypeImpl

Syntax	`DemoDriver::getDefaultMeasurementTypeImpl(Index deviceIndex, Index subsystemIndex, daqdatatypes::MeasurementType &defaultMeasurementType) const`
Purpose	Return the default measurement type supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Default measurement types supported by the specified subsystem
Return status	`DAQErr_Driver_GetDefaultMeasurementType` on failure. `DAQSuccess` on success.

getRateLimitImpl

Syntax	`DemoDriver::getRateLimitImpl(Index deviceIndex, Index subsystemIndex, daqdatatypes::RateLimit &rateLimit) const`
Purpose	Return the rate limits supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem

Output	Rate limits supported by the specified subsystem
Return status	`DAQErr_Driver_GetRateLimit` on failure. `DAQSuccess` on success.

getResolutionImpl

Syntax	`DemoDriver::getResolutionImpl(Index deviceIndex, Index subsystemIndex, uint8_T &resolution) const`
Purpose	Return the measurement resolution supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Measurement resolution supported by the specified subsystem
Return status	`DAQErr_Driver_GetResolution` on failure. `DAQSuccess` on success.

getTerminalConfigsImpl

Syntax	`DemoDriver::getTerminalConfigsImpl(Index deviceIndex, Index subsystemIndex, std::vector<daqdatatypes::TerminalConfiguration> &terminalConfigurations) const`
Purpose	Return the terminal configurations supported by a specified subsystem and device for each channel
Inputs	Index of the device, index of the subsystem
Output	Terminal configurations supported by the specified subsystem
Return status	`DAQErr_Driver_GetTerminalConfigs` on failure. `DAQSuccess` on success.

getRangesAvailableForTerminalConfigImpl

Syntax	`DemoDriver::getRangesAvailableForTerminalConfigImpl(Index deviceIndex, Index subsystemIndex, daqdatatypes::TerminalConfiguration terminalConfig, std::vector<daqdatatypes::Range> &ranges) const`

Purpose	Return ranges supported by specified terminal configuration for subsystem and device
Inputs	Index of the device, index of the subsystem, terminal configuration type
Output	Ranges supported by the specified terminal configuration for a given subsystem
Return status	`DAQErr_Driver_GetRangesAvailableForTerminalConfig` on failure. `DAQSuccess` on success.

getDefaultTerminalConfigsImpl

Syntax	`DemoDriver::getDefaultTerminalConfigsImpl(Index deviceIndex, Index subsystemIndex, std::vector<daqdatatypes::TerminalConfiguration> &defaultTerminalConfigs) const`
Purpose	Return the default terminal configuration types supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Default terminal configuration types supported by the specified subsystem
Return status	`DAQErr_Driver_GetDefaultTerminalConfigs` on failure. `DAQSuccess` on success.

isOnDemandOperationSupportedImpl

Syntax	`DemoDriver::isOnDemandOperationSupportedImpl(Index deviceIndex, Index subsystemIndex, bool &isSupported) const`
Purpose	Indicate whether on-demand operations are supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Whether on-demand operations are supported by the specified subsystem

Return status	`DAQErr_Driver_IsOnDemandOperationSupported` on failure. `DAQSuccess` on success.

getCouplingsImpl

Syntax	`DemoDriver::getCouplingsImpl(Index deviceIndex, Index subsystemIndex, std::vector<daqdatatypes::Coupling> &couplings) const`
Purpose	Return the couplings supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Couplings supported by the specified subsystem
Return status	`DAQErr_Driver_GetCouplings` on failure. `DAQSuccess` on success.

getDefaultCouplingImpl

Syntax	`DemoDriver::getDefaultCouplingImpl(Index deviceIndex, Index subsystemIndex, daqdatatypes::Coupling &defaultCoupling) const`
Purpose	Return the default coupling supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Default coupling supported by the specified subsystem
Return status	`DAQErr_Driver_GetDefaultCoupling` on failure. `DAQSuccess` on success.

getSampleTypesImpl

Syntax	`DemoDriver::getSampleTypesImpl(Index deviceIndex, Index subsystemIndex, std::vector<daqdatatypes::SampleType> &sampleTypes) const`
Purpose	Return the sample types supported by a specified subsystem and device

Inputs	Index of the device, index of the subsystem
Output	Sample types supported by the specified subsystem
Return status	`DAQErr_Driver_GetSampleTypes` on failure. `DAQSuccess` on success.

getDefaultSamplingTypeImpl

Syntax	`DemoDriver::getDefaultSamplingTypeImpl(Index deviceIndex, Index subsystemIndex, daqdatatypes::SampleType &defaultSampleType) const`
Purpose	Return the default sample type supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Default sample type supported by the specified subsystem
Return status	`DAQErr_Driver_GetDefaultSamplingType` on failure. `DAQSuccess` on success.

getDigitalChannelTypesImpl

Syntax	`DemoDriver::getDigitalChannelTypesImpl(Index deviceIndex, Index subsystemIndex, std::vector<daqdatatypes::MeasurementType> &channelMeasurementTypes) const`
Purpose	Return the digital channel type supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Vector of measurement types for the channels of the specified subsystem
Return status	`DAQErr_Driver_GetDigitalChannelTypes` on failure. `DAQSuccess` on success.

getChannelNamesImpl

Syntax	`DemoDriver::getChannelNamesImpl(Index deviceIndex, Index subsystemIndex, std::vector<std::string> &channelNames) const`
Purpose	Return the channel names supported by a specified subsystem and device
Inputs	Index of the device, index of the subsystem
Output	Channel names supported by the specified subsystem
Return status	`DAQErr_Driver_GetChannelNames` on failure. `DAQSuccess` on success.

Configuration

Configuration functions control rates, ranges, and coupling.

getRateImpl

Syntax	`DemoDriver::getRateImpl(ChannelGroupIndex groupIndex, daqsdk::float64 &rate) const`
Purpose	Return the rate supported by a specified channel group in its current configuration
Inputs	Index of the group
Output	Rate supported by the specified channel group
Return status	`DAQErr_Driver_GetRate` on failure. `DAQSuccess` on success.

setRateImpl

Syntax	`DemoDriver::setRateImpl(ChannelGroupIndex groupIndex, daqsdk::float64 rate)`
Purpose	Set the rate for a specified channel group in its current configuration
Inputs	Index of the group, rate

Output	None
Return status	`DAQErr_Driver_SetRate` on failure. `DAQSuccess` on success.

getChannelCouplingImpl

Syntax	`DemoDriver::getChannelCouplingImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, daqdatatypes::Coupling &coupling) const`
Purpose	Return the channel coupling of a specified channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel
Output	Coupling supported by the specified channel
Return status	`DAQErr_Driver_GetChannelCoupling` on failure. `DAQSuccess` on success.

setChannelCouplingImpl

Syntax	`DemoDriver::setChannelCouplingImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, std::string coupling)`
Purpose	Set the channel coupling of a specified channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel
Output	None
Return status	`DAQErr_Driver_SetChannelCoupling` on failure. `DAQSuccess` on success.

getChannelTerminalConfigImpl

Syntax	`DemoDriver::getChannelTerminalConfigImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, daqdatatypes::TerminalConfiguration &terminalConfig) const`

Purpose	Return the terminal configuration of a specified channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel
Output	Terminal configuration supported by the specified channel
Return status	`DAQErr_Driver_GetChannelTerminalConfig` on failure. `DAQSuccess` on success.

setChannelTerminalConfigImpl

Syntax	`DemoDriver::setChannelTerminalConfigImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, std::string terminalConfig)`
Purpose	Set the terminal configuration of a specified channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel
Output	None
Return status	`DAQErr_Driver_SetChannelTerminalConfig` on failure. `DAQSuccess` on success.

getChannelRangeImpl

Syntax	`DemoDriver::getChannelRangeImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, daqdatatypes::Range &range) const`
Purpose	Return the range of a specified channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel
Output	Range supported by the specified channel
Return status	`DAQErr_Driver_GetChannelRange` on failure. `DAQSuccess` on success.

setChannelRangeImpl

Syntax	`DemoDriver::setChannelRangeImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, daqdatatypes::Range range)`
Purpose	Set the range of a specified channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel
Output	None
Return status	DAQErr_Driver_SetChannelRange on failure. `DAQSuccess` on success.

getChannelDirectionImpl

Syntax	`DemoDriver::getChannelDirectionImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, daqdatatypes::ChannelDirection &direction) const`
Purpose	Return the direction of a specified digital channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel
Output	Channel direction for specified channel.
Return status	`DAQErr_Driver_GetChannelDirection` on failure. `DAQSuccess` on success.

setChannelDirectionImpl

Syntax	`DemoDriver::setChannelDirectionImpl(Index deviceIndex, Index subsystemIndex, Index channelIndex, std::string direction)`
Purpose	Set the direction of a specified digital channel for a given subsystem and device
Inputs	Index of the device, index of the subsystem, index of the channel, direction of the channel specified as `"Input"` or `"Output"`
Output	None

Return status	`DAQErr_Driver_SetChannelDirection` on failure. `DAQSuccess` on success.

Reservation

Reservation functions query device and channel availability.

isDeviceAvailableImpl

Syntax	`DemoDriver::isDeviceAvailableImpl(Index deviceIndex, bool &isDeviceAvailable) const`
Purpose	Return whether or not a specified device is still available, connected, committed, and enumerated
Inputs	Index of the device
Output	Return whether or not the device is available as previously committed and enumerated
Return status	`DAQErr_Driver_IsDeviceAvailable` on failure. `DAQSuccess` on success.

isRegistrationReservationImpl

Syntax	`DemoDriver::isRegistrationReservationImpl(Index deviceIndex, bool &isReservation) const`
Purpose	Return whether or not registering a channel in a channel group reserves the channel group
Inputs	Index of the channel group
Output	Return whether or not “registration is reservation.” (See “Channel Groups” on page 3-14.)
Return status	`DAQErr_Driver_IsRegistrationReservation` on failure. `DAQSuccess` on success.

Single Scans

Single scan functions acquire or generate a static scan of data.

inputSingleScan

Syntax	`DemoDriver::inputSingleScanImpl(ChannelGroupIndex groupIndex, DataScan &data) const`
Purpose	Acquire a scan of data for all channels registered with a channel group
Inputs	None
Output	Acquired scan of data
Return status	`DAQErr_Driver_InputSingleScan` on failure. `DAQSuccess` on success.

OutputSingleScan

Syntax	`DemoDriver::outputSingleScanImpl(ChannelGroupIndex groupIndex, DataScan &&outputData) const`
Purpose	Generate a scan of data for all channels registered with a channel group
Inputs	Data to generate output
Output	None
Return status	`DAQErr_Driver_OutputSingleScan` on failure. `DAQSuccess` on success.

See Also

Related Examples

- “Create Your Adaptor from the Demo Adaptor” on page 3-2

More About

- “Errors and Exceptions” on page 3-12
- “Channel Groups” on page 3-14
- “Streaming API Reference” on page 4-23

- “Custom Functions” on page 3-18

Streaming API Reference

In this section...

Streaming is used in the generation or acquisition of clocked data to allow asynchronous operation that does not block MATLAB. Stream channels accommodate the flow of data separately from the session-dispatcher route. This also allows data sets that might exceed the size of the memory on the device.

The following functions are defined in the analog streaming source file `daqstream_analog.cpp`. Corresponding functions for digital I/O streaming are defined in `daqstream_digital.cpp`.

Initialization and Configuration

makeStream

Syntax	`static DAQStream* makeStream(...)`
Purpose	Factory method to create channel group stream.
Inputs	Fixed signature
Output	Pointer to DAQStream
Usage notes	Use as shown in `daqstream_analog.cpp`.

initialize

Syntax	`int64_T DAQStreamAnalog::initialize()`
Purpose	Initialize channel group after constructor.
Inputs	None
Output	None

terminate

Syntax	`int64_T DAQStreamAnalog::terminate()`
Purpose	Terminate DAQ stream prior to its destruction.
Inputs	None
Output	None

configureStream

Syntax	`int64_T DAQStreamAnalog::configureStream()`
Purpose	Configure group of channels on the DAQ device driver for streaming operation.
Inputs	None
Output	None

unconfigureStream

Syntax	`int64_T DAQStreamAnalog::unconfigureStream()`
Purpose	Unconfigure groups of channels when DAQStream channel is closed from MATLAB.
Inputs	None
Output	None

registerCallbacks

Syntax	`int64_T DAQStreamAnalog::registerCallbacks()`
Purpose	Register any callback handlers required by DAQ device driver following `configureStream`.
Inputs	None
Output	None

unregisterCallbacks

Syntax	`int64_T DAQStreamAnalog::unregisterCallbacks()`

Purpose	Unregister immediately prior to `unconfigureStream` any callback handlers registered with DAQ device driver.
Inputs	None
Output	None

Start and Stop

prestart

Syntax	`int64_T DAQStreamAnalog::prestart()`
Purpose	Called on a per-run basis prior to the streaming operation `start`, typically to reset scan counters.
Inputs	None
Output	None

start

Syntax	`int64_T DAQStreamAnalog::start()`
Purpose	Start the streaming operation for the given channelGroupHandle.
Inputs	None
Output	None

stop

Syntax	`int64_T DAQStreamAnalog::stop()`
Purpose	Stop the streaming operation for the given channelGroupHandle.
Inputs	None
Output	None

Data Availability

getNumInputScansAvailable

Syntax	`int64_T DAQStreamAnalog::getNumInputScansAvailable(uint64_T& numScansAcquired)`
Purpose	Query the number of input scans available to be read by a `read` or `readWrite` call.
Inputs	None
Output	Number of scans.

getNumScansOutputByHardware

Syntax	`int64_T DAQStreamAnalog::getNumScansOutputByHardware(uint64_T & numScansGenerated)`
Purpose	Query the number of scans output by the hardware by a `write` or `readWrite` call.
Inputs	None
Output	Number of scans.

getOutputBufferSize

Syntax	`int64_T DAQStreamAnalog::getOutputBufferSize(uint64_T& outputBufferSize)`
Purpose	Query the DAQ device output buffer size in number of scans.
Inputs	None
Output	Buffer size in scans.

flushOutputBuffer

Syntax	`int64_T DAQStreamAnalog::flushOutputBuffer()`
Purpose	Empty the output buffer.

Inputs	None
Output	None

isDeviceDone

Syntax	`int64_T DAQStreamAnalog::isDeviceDone(bool& isDone)`
Purpose	Poll the vendor driver immediately following a call to `stop`.
Inputs	None
Output	True if device is done streaming.

Transfer Data

read

Syntax	`int64_T DAQStreamAnalog::read(float64 * const pReadBuffer, uint64_T numReadScans)`
Purpose	Read acquired data from the DAQ device into the read buffer.
Inputs	`pReadBuffer`: buffer used by stream to store input data acquired from the device. `numReadScans`: number of scans to copy into the provided buffer.
Output	None
Notes	The stream is responsible for the lifetime of the buffer.

write

Syntax	`int64_T DAQStreamAnalog::write(float64 const * const pWriteBuffer, uint64_T numWriteScans)`
Purpose	Write data from the buffer to the device for output generation.
Inputs	`pWriteBuffer`: buffer used by stream to store output data to be generated by the device. `numWriteScans`: number of valid scans to copy from the provided buffer.
Output	None

Notes	The stream is responsible for the lifetime of the buffer.

readWrite

Syntax	`int64_T DAQStreamAnalog::readWrite(float64* const pReadBuffer, uint64_T numReadScans, float64 const * const pWriteBuffer, uint64_T numWriteScans)`
Purpose	Simultaneously read and write data between the buffers and a DAQ device duplex channel.
Inputs	`pReadBuffer`: buffer used by stream to store input data acquired from the device. `numReadScans`: number of scans to copy into the provided buffer. `pWriteBuffer`: buffer used by stream to store output data to be generated by the device. `numWriteScans`: number of valid scans to copy from the provided buffer.
Output	None
Notes	The stream is responsible for the lifetime of the buffer.

See Also

Related Examples

- “Create Your Adaptor from the Demo Adaptor” on page 3-2

More About

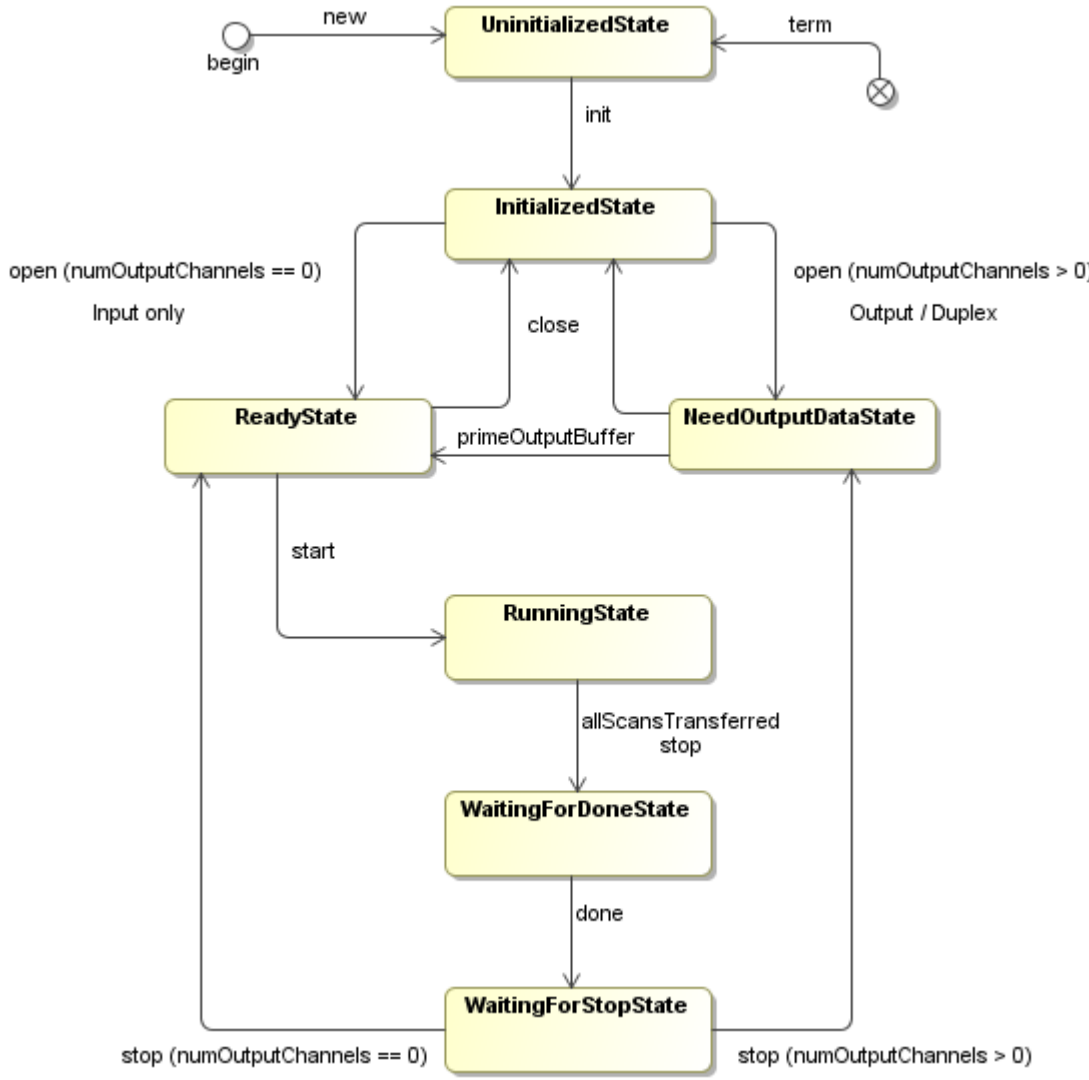
- “Errors and Exceptions” on page 3-12
- “Channel Groups” on page 3-14
- “Streaming Input and Output” on page 2-10
- “State Machine Diagram” on page 5-2
- “Streaming Sequence Diagrams” on page 5-4

- “Adaptor API Reference” on page 4-2
- “Custom Functions” on page 3-18

State and Sequence Diagrams

State Machine Diagram

This composite diagram shows the state machine for input, output, and duplex channels.



See Also

More About

- “Streaming API Reference” on page 4-23
- “Foreground Streaming Sequences” on page 5-5
- “Background Streaming Sequences” on page 5-12

Streaming Sequence Diagrams

These sequence diagrams provide details of timing for streaming functionality. They might be useful for debugging code during development of your adaptor.

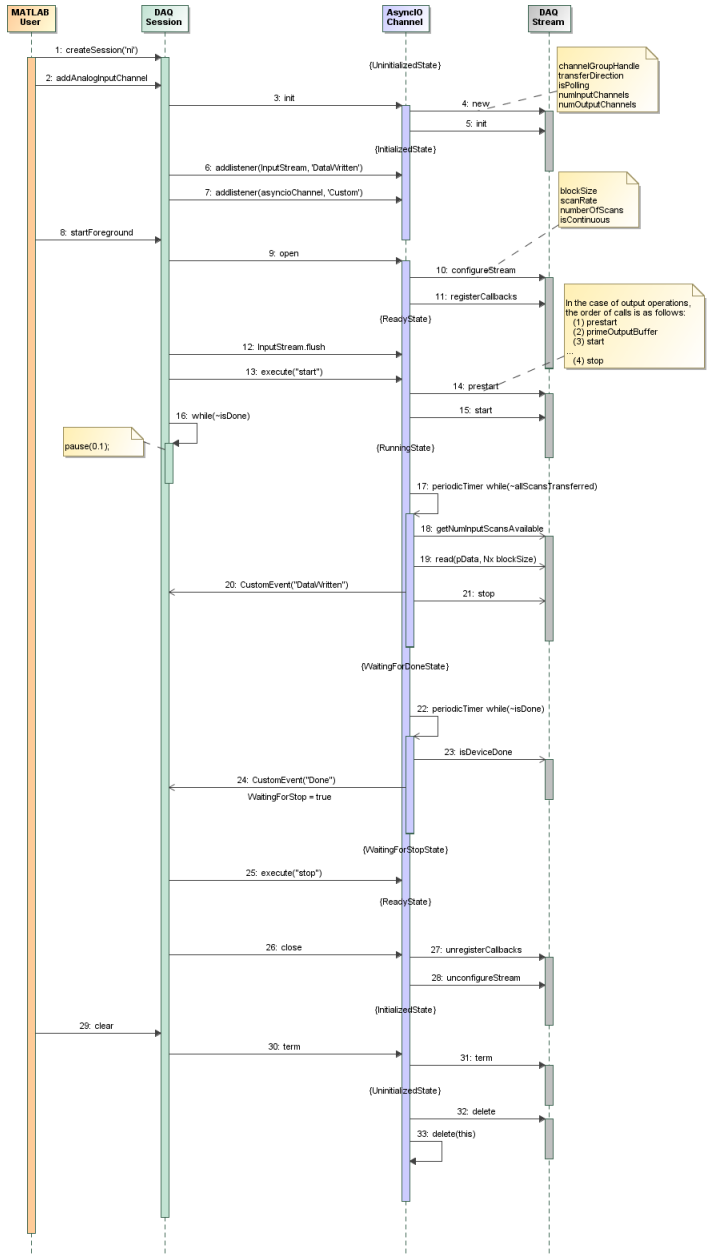
Foreground Streaming Sequences

This topic includes sequence diagrams for finite analog input and output in the foreground.

In this section...

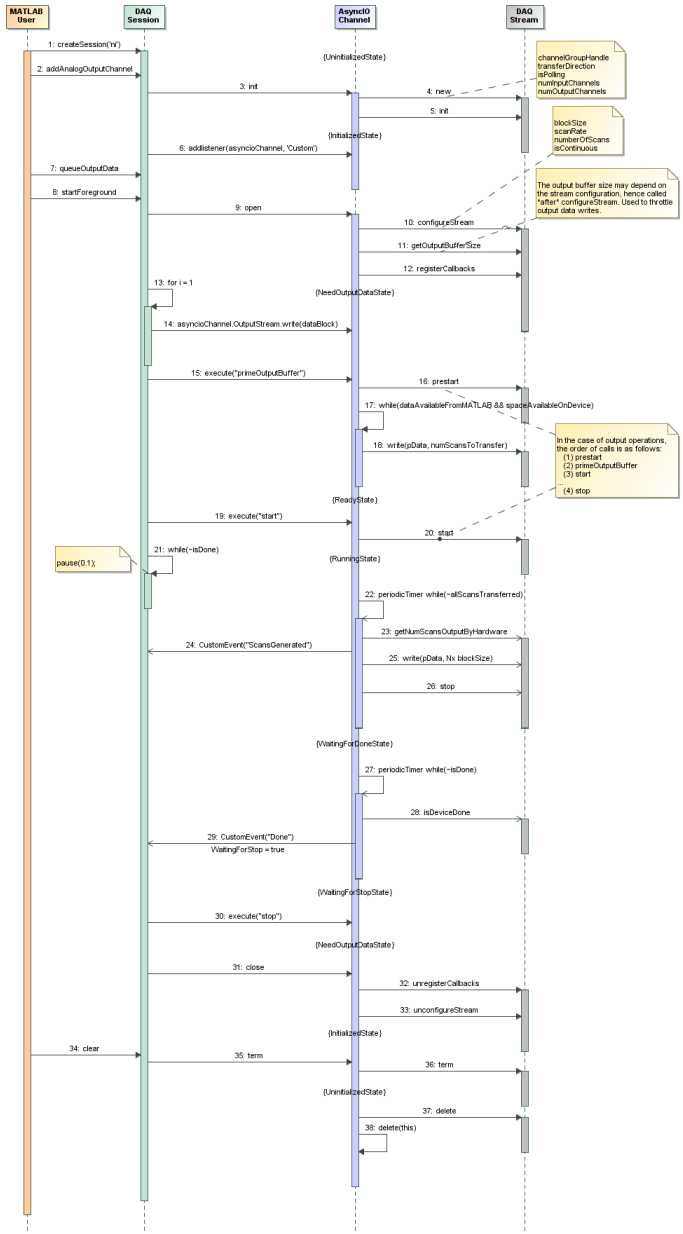
Sequence for Finite Foreground Input

This diagram shows the timing sequence for a finite (fixed-size) analog input in the foreground. It demonstrates the interfaces between a DAQ session, AsyncIO channel, and a DAQ stream when a user is performing finite foreground clocked acquisitions using the session interface.


MATLAB User
DAQ Session
AsyncIO Channel
DAQ Stream
1: createSession('ni')
{UninitializedState}
channelGroupHandle
transferDirection
isPolling
numInputChannels
numOutputChannels
2: addAnalogInputChannel
3: init
4: new
5: init
{InitializedState}
6: addlistener(inputStream, 'DataWritten')
7: addlistener(asyncioChannel, 'Custom')
blockSize
scanRate
numberOfScans
isContinuous
8: startForeground
9: open
10: configureStream
11: registerCallbacks
{ReadyState}
In the case of output operations, the order of calls is as follows:
(1) prestart
(2) primeOutputBuffer
(3) start
(4) stop
12: inputStream.flush
13: execute("start")
14: prestart
15: start
16: while(~isDone)
pause(0.1);
{RunningState}
17: periodicTimer while(~allScansTransferred)
18: getNumInputScansAvailable
19: read(pData, Nx blockSize)
20: CustomEvent("DataWritten")
21: stop
{WaitingForDoneState}
22: periodicTimer while(~isDone)
23: isDeviceDone
24: CustomEvent("Done")
WaitingForStop = true
{WaitingForStopState}
25: execute("stop")
{ReadyState}
26: close
27: unregisterCallbacks
28: unconfigureStream
{InitializedState}
29: clear
30: term
31: term
{UninitializedState}
32: delete
33: delete(this)

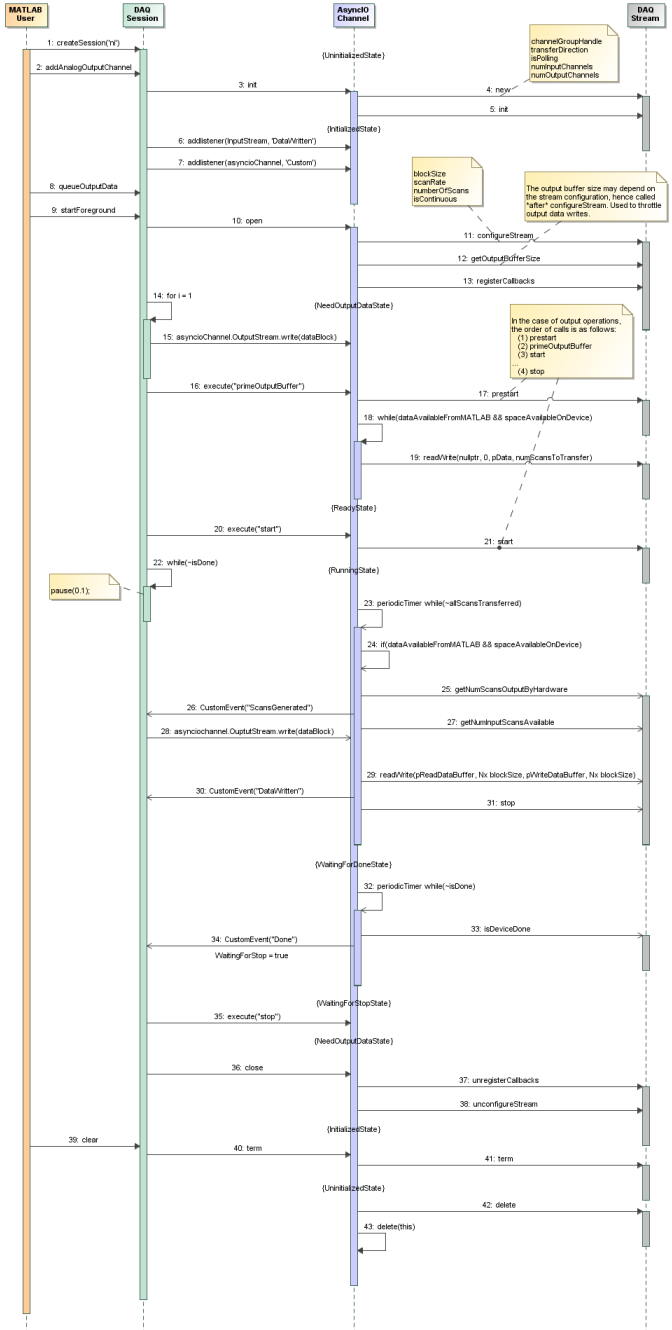
Sequence for Finite Foreground Output

This diagram shows the timing sequence for a finite (fixed-size) analog output in the foreground. It demonstrates the interfaces between a DAQ session, AsyncIO channel, and a DAQ stream when a user is performing finite foreground clocked signal generation using the session interface.


MATLAB User
DAQ Session
AsyncIO Channel
DAQ Stream
1: createSession('ni')
{UninitializedState}
channelGroupHandle
transferDirection
isPolling
numInputChannels
numOutputChannels
2: addAnalogOutputChannel
3: init
4: new
5: init
blockSize
scanRate
numberOfScans
isContinuous
{InitializedState}
6: addlistener(asyncIoChannel, 'Custom')
7: queueOutputData
8: startForeground
The output buffer size may depend on the stream configuration, hence called *after* configureStream. Used to throttle output data writes.
9: open
10: configureStream
11: getOutputBufferSize
12: registerCallbacks
13: for i = 1
{NeedOutputDataState}
14: asyncIoChannel.OutputStream.write(dataBlock)
15: execute("primeOutputBuffer")
16: prestart
17: while(dataAvailableFromMATLAB && spaceAvailableOnDevice)
In the case of output operations, the order of calls is as follows:
(1) prestart
(2) primeOutputBuffer
(3) start
...
(4) stop
18: write(pData, numScansToTransfer)
{ReadyState}
19: execute("start")
20: start
21: while(~isDone)
pause(0.1);
{RunningState}
22: periodicTimer while(~allScansTransferred)
23: getNumScansOutputByHardware
24: CustomEvent("ScansGenerated")
25: write(pData, Nx blockSize)
26: stop
{WaitingForDoneState}
27: periodicTimer while(~isDone)
28: isDeviceDone
29: CustomEvent("Done")
WaitingForStop = true
{WaitingForStopState}
30: execute("stop")
{NeedOutputDataState}
31: close
32: unregisterCallbacks
33: unconfigureStream
{InitializedState}
34: clear
35: term
36: term
{UninitializedState}
37: delete
38: delete(this)

Sequence for Finite Foreground Duplex Channel

This diagram shows the timing sequence for a finite (fixed-size) simultaneous analog input and output in the foreground. It demonstrates the interface between a DAQ session and the AsyncIO channel when a user is performing a finite input/output operation using the session interface


MATLAB User
DAQ Session
AsyncIO Channel
DAQ Stream
1: createSession('ni')
{UninitializedState}
channelGroupHandle
transferDirection
isPolling
numInputChannels
numOutputChannels
2: addAnalogOutputChannel
3: init
4: new
5: init
{InitializeState}
6: addlistener(inputStream, 'DataWritten')
7: addlistener(asyncioChannel, 'Custom')
blockSize
scanRate
numberOfScans
isContinuous
The output buffer size may depend on the stream configuration, hence called *after* configureStream. Used to throttle output data writes.
8: queueOutputData
9: startForeground
10: open
11: configureStream
12: getOutputBufferSize
13: registerCallbacks
14: for i = 1
{NeedOutputDataState}
15: asyncioChannel.OutputStream.write(dataBlock)
In the case of output operations, the order of calls is as follows:
(1) prestart
(2) primeOutputBuffer
(3) start
(4) stop
16: execute("primeOutputBuffer")
17: prestart
18: while(dataAvailableFromMATLAB && spaceAvailableOnDevice)
19: readWrite(nullptr, 0, pData, numScansToTransfer)
{ReadyState}
20: execute("start")
21: start
22: while(~isDone)
{RunningState}
pause(0.1);
23: periodicTimer while(~allScansTransferred)
24: if(dataAvailableFromMATLAB && spaceAvailableOnDevice)
25: getNumScansOutputByHardware
26: CustomEvent("ScansGenerated")
27: getNumInputScansAvailable
28: asyncioChannel.OutputStream.write(dataBlock)
29: readWrite(pReadDataBuffer, Nx blockSize, pWriteDataBuffer, Nx blockSize)
30: CustomEvent("DataWritten")
31: stop
{WaitingForDoneState}
32: periodicTimer while(~isDone)
33: isDeviceDone
34: CustomEvent("Done")
WaitingForStop = true
{WaitingForStopState}
35: execute("stop")
{NeedOutputDataState}
36: close
37: unregisterCallbacks
38: unconfigureStream
{InitializedState}
39: clear
40: term
41: term
{UninitializedState}
42: delete
43: delete(this)

See Also

More About

- “Streaming API Reference” on page 4-23
- “State Machine Diagram” on page 5-2
- “Background Streaming Sequences” on page 5-12

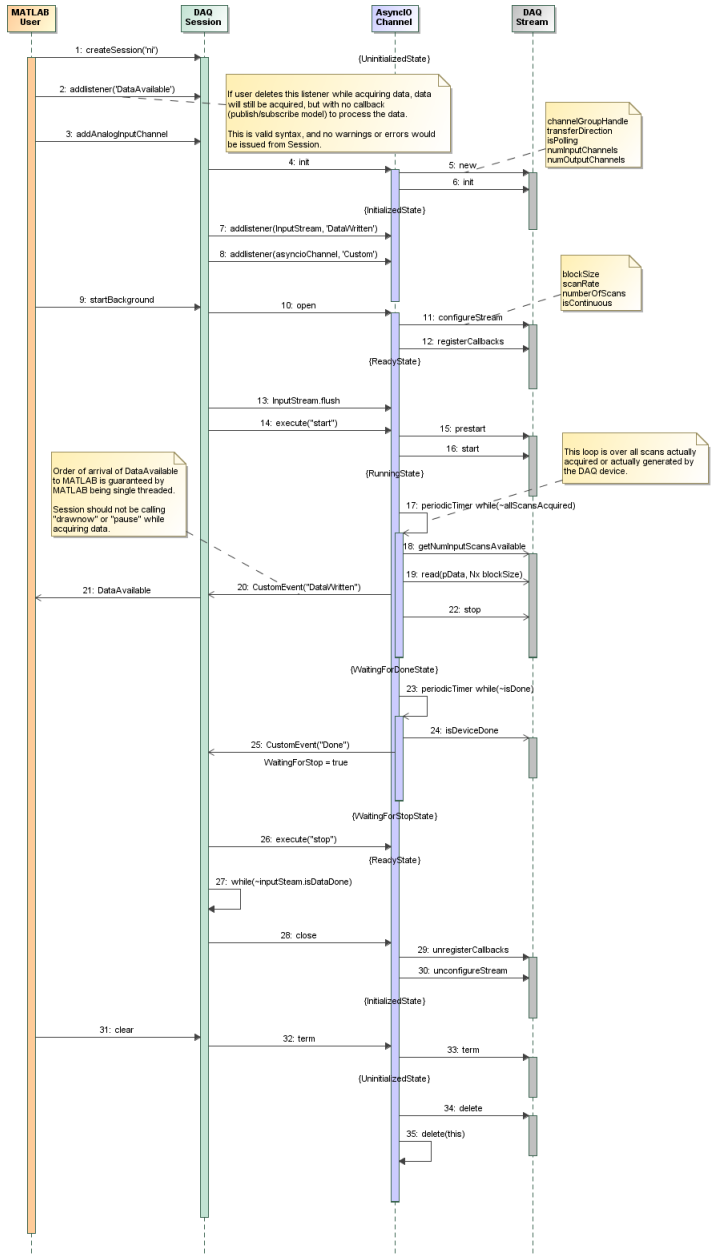
Background Streaming Sequences

This topic includes sequence diagrams for analog input and output in the background.

In this section...

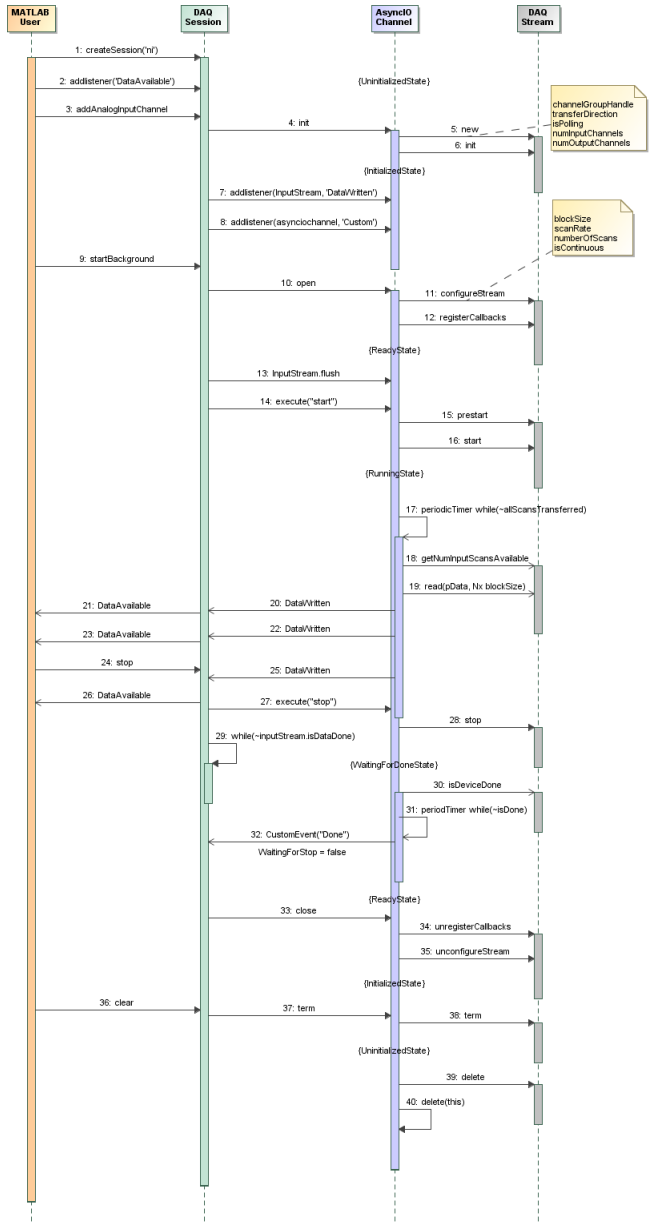
Sequence for Finite Background Input

This diagram shows the timing sequence for a finite (fixed-size) analog input in the background. It illustrates the interface between a DAQ session and the AsyncIO channel when a user is performing a finite background clocked acquisition using a the session interface.


MATLAB User
DAQ Session
AsyncIO Channel
DAQ Stream
1: createSession('ni')
{UninitializedState}
2: addlistener('DataAvailable')
If user deletes this listener while acquiring data, data will still be acquired, but with no callback (publish/subscribe model) to process the data.
This is valid syntax, and no warnings or errors would be issued from Session.
3: addAnalogInputChannel
channelGroupHandle
transferDirection
isPolling
numInputChannels
numOutputChannels
4: init
5: new
6: init
{InitializedState}
7: addlistener(InputStream, 'DataWritten')
8: addlistener(asyncioChannel, 'Custom')
blockSize
scanRate
numberOfScans
isContinuous
9: startBackground
10: open
11: configureStream
12: registerCallbacks
{ReadyState}
13: inputStream.flush
14: execute("start")
15: prestart
16: start
{RunningState}
This loop is over all scans actually acquired or actually generated by the DAQ device.
Order of arrival of DataAvailable to MATLAB is guaranteed by MATLAB being single threaded.
Session should not be calling "drawnow" or "pause" while acquiring data.
17: periodicTimer while(~allScansAcquired)
18: getNumInputScansAvailable
19: read(pData, Nx blockSize)
21: DataAvailable
20: CustomEvent("DataWritten")
22: stop
{WaitingForDoneState}
23: periodicTimer while(~isDone)
24: isDeviceDone
25: CustomEvent("Done")
WaitingForStop = true
{WaitingForStopState}
26: execute("stop")
{ReadyState}
27: while(~inputSteam.isDataDone)
28: close
29: unregisterCallbacks
30: unconfigureStream
{InitializedState}
31: clear
32: term
33: term
{UninitializedState}
34: delete
35: delete(this)

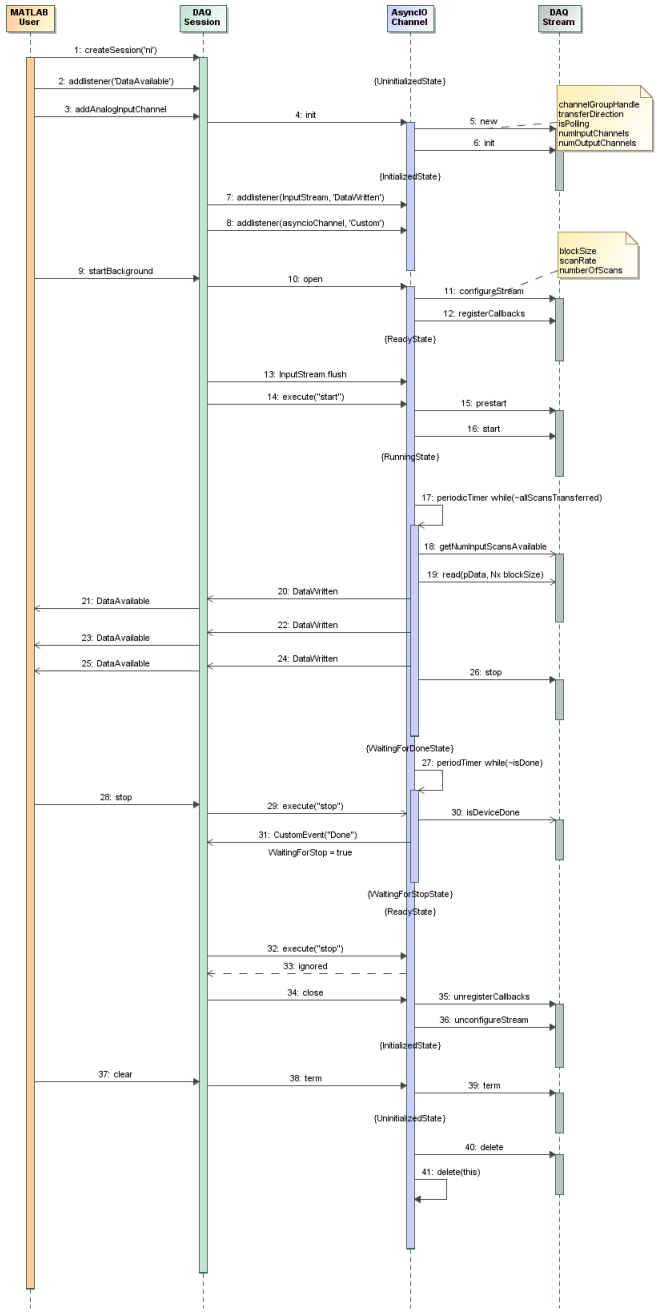
Sequence for Continuous Background Input with Stop

This diagram shows the timing sequence for a continuous analog input in the background, with a stop request while the device is acquiring data. It illustrates the interface between a DAQ session and the AsyncIO channel when a user is performing a continuous background clocked acquisition, then calls stop while data is being acquired.


MATLAB User
DAQ Session
AsyncIO Channel
DAQ Stream
1: createSession('ni')
2: addlistener('DataAvailable')
{UninitializedState}
3: addAnalogInputChannel
4: init
5: new
6: init
channelGroupHandle
transferDirection
isPolling
numInputChannels
numOutputChannels
{InitializedState}
7: addlistener(InputStream, 'DataWritten')
8: addlistener(asynciochannel, 'Custom')
blockSize
scanRate
numberOfScans
isContinuous
9: startBackground
10: open
11: configureStream
12: registerCallbacks
{ReadyState}
13: InputStream.flush
14: execute("start")
15: prestart
16: start
{RunningState}
17: periodicTimer while(~allScansTransferred)
18: getNumInputScansAvailable
19: read(pData, Nx blockSize)
21: DataAvailable
20: DataWritten
23: DataAvailable
22: DataWritten
24: stop
25: DataWritten
26: DataAvailable
27: execute("stop")
28: stop
29: while(~inputStream.isDataDone)
{WaitingForDoneState}
30: isDeviceDone
31: periodTimer while(~isDone)
32: CustomEvent("Done")
WaitingForStop = false
{ReadyState}
33: close
34: unregisterCallbacks
35: unconfigureStream
{InitializedState}
36: clear
37: term
38: term
{UninitializedState}
39: delete
40: delete(this)

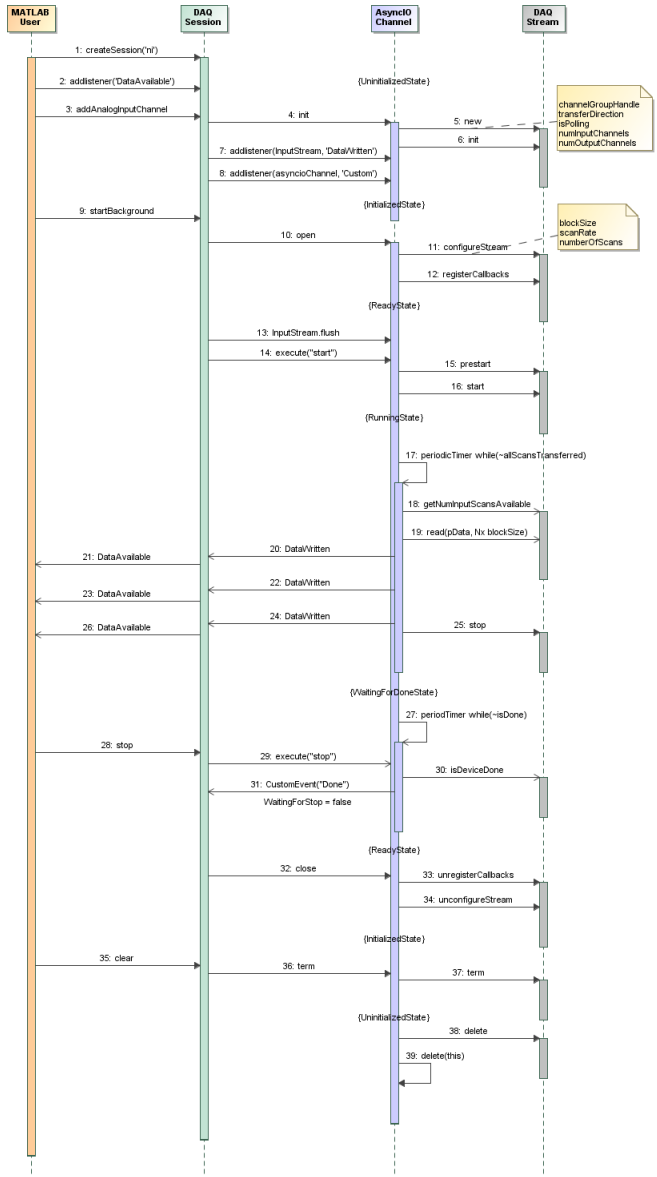
Sequence for Finite Background Input with Wait

This use case revisits the finite background acquisition, when all scans have been acquired and the background operation is naturally stopping, and at the same time, the user issues a stop command while the DAQ AsyncIO plugin is in the “WaitingForDoneState.”


MATLAB User
DAQ Session
AsyncIO Channel
DAQ Stream
1: createSession('ni')
2: addlistener('DataAvailable')
{UninitializedState}
3: addAnalogInputChannel
4: init
5: new
channelGroupHandle
transferDirection
isPolling
numInputChannels
numOutputChannels
6: init
{InitializedState}
7: addlistener(InputStream, 'DataWritten')
8: addlistener(asyncioChannel, 'Custom')
blockSize
scanRate
numberOfScans
9: startBackground
10: open
11: configureStream
12: registerCallbacks
{ReadyState}
13: InputStream.flush
14: execute("start")
15: prestart
16: start
{RunningState}
17: periodicTimer while(~allScansTransferred)
18: getNumInputScansAvailable
19: read(pData, Nx blockSize)
20: DataWritten
21: DataAvailable
22: DataWritten
23: DataAvailable
24: DataWritten
25: DataAvailable
26: stop
{WaitingForDoneState}
27: periodTimer while(~isDone)
28: stop
29: execute("stop")
30: isDeviceDone
31: CustomEvent("Done")
WaitingForStop = true
{WaitingForStopState}
{ReadyState}
32: execute("stop")
33: ignored
34: close
35: unregisterCallbacks
36: unconfigureStream
{InitializedState}
37: clear
38: term
39: term
{UninitializedState}
40: delete
41: delete(this)

Sequence for Finite Background Input with Stop Race

This situation revisits the finite background acquisition, when all scans have been acquired and the background operation is naturally stopping, and at the same time, the user issues a stop command while the DAQ AsyncIO plugin is in the last iteration of the “RunningState.”


MATLAB User
DAQ Session
AsyncIO Channel
DAQ Stream
1: createSession('ni')
2: addlistener('DataAvailable')
{UninitializedState}
3: addAnalogInputChannel
4: init
5: new
6: init
channelGroupHandle
transferDirection
isPolling
numInputChannels
numOutputChannels
7: addlistener(inputStream, 'DataWritten')
8: addlistener(asyncioChannel, 'Custom')
{InitializedState}
9: startBackground
10: open
11: configureStream
12: registerCallbacks
blockSize
scanRate
numberOfScans
{ReadyState}
13: inputStream.flush
14: execute("start")
15: prestart
16: start
{RunningState}
17: periodicTimer while(~allScansTransferred)
18: getNumInputScansAvailable
19: read(pData, Nx blockSize)
20: DataWritten
21: DataAvailable
22: DataWritten
23: DataAvailable
24: DataWritten
25: stop
26: DataAvailable
{WaitingForDoneState}
27: periodTimer while(~isDone)
28: stop
29: execute("stop")
30: isDeviceDone
31: CustomEvent("Done")
WaitingForStop = false
{ReadyState}
32: close
33: unregisterCallbacks
34: unconfigureStream
{InitializedState}
35: clear
36: term
37: term
{UninitializedState}
38: delete
39: delete(this)

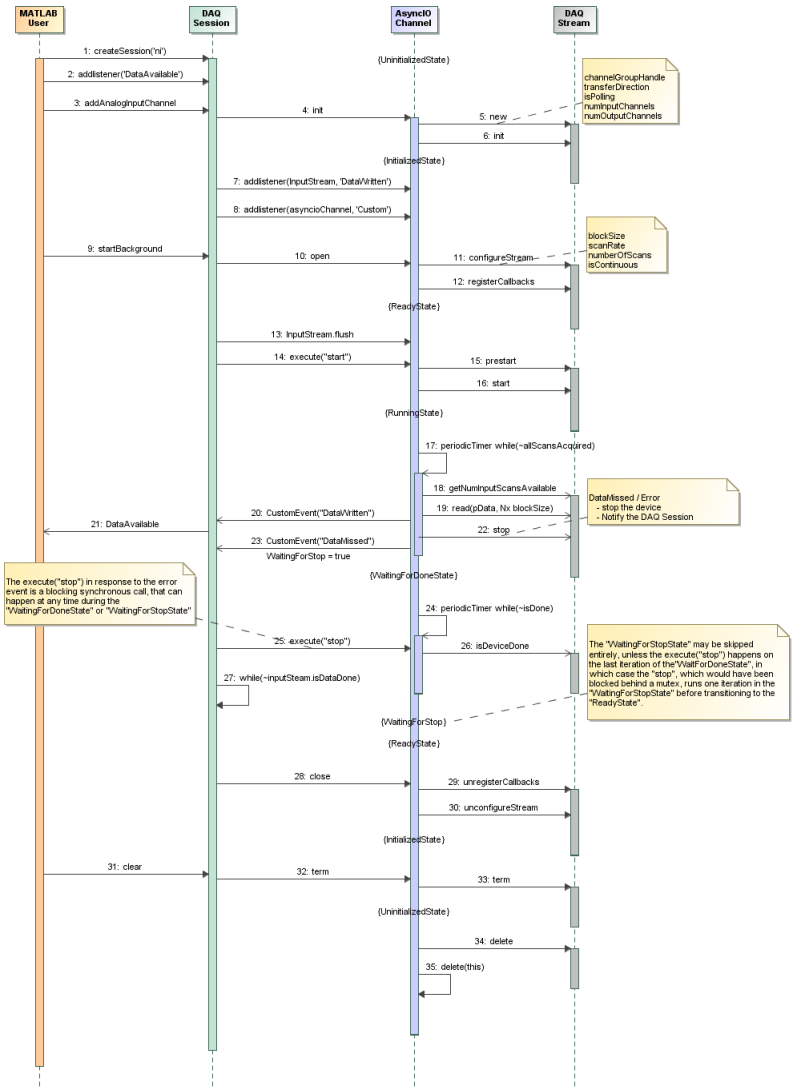
See Also

More About

- “Streaming API Reference” on page 4-23
- “State Machine Diagram” on page 5-2
- “Foreground Streaming Sequences” on page 5-5

Sequence for Errors and Exceptions

This sequence diagram summarizes the DataMissed events and error handling.



See Also

More About

- “Channel Groups” on page 3-14
- “Errors and Exceptions” on page 3-12
- “State Machine Diagram” on page 5-2
- “Foreground Streaming Sequences” on page 5-5
- “Background Streaming Sequences” on page 5-12

Functions — Alphabetical List

buildAdaptor

Build adaptor for third-party data acquisition interface

Syntax

```
daq.sdk.utility.mex.buildAdaptor(adaptorName,customFunc,srcPath,
outputPath)
daq.sdk.utility.mex.buildAdaptor(adaptorName,customFunc,srcPath,
outputPath,vendorLib)
script = daq.sdk.utility.mex.buildAdaptor( ___ )
```

Description

`daq.sdk.utility.mex.buildAdaptor(adaptorName,customFunc,srcPath, outputPath)` builds an adaptor for enumerating, configuring, and streaming data to and from a data acquisition device driver.

Note This function requires that your system is configured with Microsoft Visual Studio 2013 or later.

`daq.sdk.utility.mex.buildAdaptor(adaptorName,customFunc,srcPath, outputPath,vendorLib)` allows you to specify a custom library for the build.

`script = daq.sdk.utility.mex.buildAdaptor( ___ )` returns the script used for the build. This can be useful for diagnostic purposes.

Examples

Build Custom Adaptor

Build the custom adaptor named `MyAdaptor`.

```
daq.sdk.utility.mex.buildAdaptor('MyAdaptor','custom_my', ...
    'c:\adaptors\sdk\daqadaptor','c:\adaptors\sdk\bin\win64');
```

View Adaptor Build Script

Build the custom adaptor and return the build script.

```
scr = daq.sdk.utility.mex.buildAdaptor('MyAdaptor','custom_my', ...
'c:\adaptors\sdk\daqadaptor','c:\adaptors\sdk\bin\win64');
scr
```

```
scr =

    'mex 'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor\Shared\dispatcher.cpp'
'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor\Shared\daqadaptor.cpp'
'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor\Shared\daqstream.cpp'
'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor\Shared\adaptorfactory.cpp'
'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor\MyAdaptor.cpp'
'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor\daqstream_analog.cpp'
'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor\custom_my.cpp'
-I'C:\Program Files\MATLAB\R2017\toolbox\daq\daqsdk\src\daqadaptor\Shared'
-I'C:\Program Files\MATLAB\R2017\toolbox\daq\daqsdk\src\include'
-I'C:\adaptors\daqsdk\src\daqadaptor\MyAdaptor' -DADAPTOR=MyAdaptor
-DDAQADAPTOR_EXPORT -DINT16_MIN=-32768 -DINT16_MAX=32767 -output MyAdaptor
-outdir 'C:\adaptors\daqsdk\bin\win64' -v -g COMPFLAGS='$COMPFLAGS -W3'
CXXFLAGS='$CXXFLAGS -std=c++11''
```

You can save this script to a file and further modify it. You can run your modified script with `eval` or `daq.sdk.utility.mex.runBuildScript`. For syntax options, type

```
help daq.sdk.utility.mex.runBuildScript
```

Build with Custom Library

Use the custom library `MyLibrary` for building an adaptor.

```
pathToHeaderAndLib = 'C:\libraries\MyLibrary'
myLibrary.HeaderPath = fullfile(pathToHeaderAndLib,'include');
myLibrary.LibPath = fullfile(pathToHeaderAndLib,'lib');
myLibrary.LibName = 'MyLibrary';
```

```
buildAdaptor('DemoAdaptor','custom_demo',adaptorPath,outputPath,myLibrary);
```

Input Arguments

adaptorName — Name of adaptor

char vector | string

Name of the adaptor, specified as a character vector or string.

Example: `'DemoAdaptor'`

Data Types: `char` | `string`

customFunc — File containing custom functions

char vector or string

Name of the file containing the source code for custom functions, specified as a character vector or string. The file must be in the folder identified by `srcPath`.

Example: `'custom_demo.cpp'`

Data Types: `char` | `string`

srcPath — Path to adaptor source folder

char vector or string

Path to adaptor source folder, specified as a character vector or string.

Example: `'c:\temp\sdk\daqadaptor'`

Data Types: `char` | `string`

outputPath — Path to adaptor MEX-file

char vector or string

Path to generated adaptor MEX-file location, specified as a character vector or string.

Example: `'c:\temp\sdk\bin\win64'`

Data Types: `char` | `string`

vendorLib — Vendor library locations

struct

Vendor library locations, specified as a structure containing these three fields:

- `HeaderPath` — a character vector specifying the path to the vendor header.
- `LibPat` — a character vector specifying the path to the vendor static library.
- `LibName` — a character vector specifying the name of the static library, without file extension.

Data Types: `struct`

Output Arguments

`script` — Build script
character vector

Build script returned as a character vector. This script indicates what the function ran to build the adaptor.

See Also

Topics

“Create Your Adaptor from the Demo Adaptor” on page 3-2

Introduced in R2017a

enableDemoAdaptorDiscovery

Allow SDK demo adaptor to be enabled for device discovery and usage

Syntax

```
daq.sdk.utility.enableAdaptorDiscovery
```

Description

`daq.sdk.utility.enableAdaptorDiscovery` allows the SDK demo adaptor to be found and used in a data acquisition session. The adaptor is enabled until the end of the MATLAB session or until execution of `daqreset`.

Examples

Enable the Demo Adaptor

Enable the demo adaptor and view its devices.

```
daqreset;
daq.sdk.utility.enableDemoAdaptorDiscovery;
devices = daq.getDevices
```

```
devices =

Data acquisition devices:

index Vendor Device ID    Description
----- ------ --------- ------------------
1     mw     MWDev0    MathWorks MW314159
```

```
2     mw     MWDev1    MathWorks MW314159
3     mw     MWDev2    MathWorks MW628318
```

See Also

Functions
`daqreset`

Topics

“Demo Adaptor Description” on page 2-2
“Enable the Demo Adaptor” on page 2-5
“Session Workflows with the Demo Adaptor” on page 2-6
“Create Your Adaptor from the Demo Adaptor” on page 3-2

Introduced in R2017a