United
Electronic
Industries

The High-Performance Alternative

UeiDaq Framework User Manual

January 2020 Edition

Table of contents

1. Introduction

This manual provides documentation for using the UeiDaq framework API.

The UeiDaq framework offers a simple yet powerful API to access your UEI data acquisition devices. The API is used to access devices from the PowerDAQ and PowerDNA product lines. It also implements a simulation device that allows the end-user to start working with the API without hardware. The capabilities of the simulation device are modeled after the capabilities of PowerDAQ multi-function devices.

The UeiDaq framework comes with bindings for various programming languages such as C, C++, C#, VB6, VB.NET and scientific software packages such as LabVIEW and Matlab.

The UeiDaq framework API is supported on Window 7 and greater.

This document gives further details on the features and functionalities of the API that a designer can employ to create an application.

2. UeiDaq framework architecture

2.1. Overview

The following diagram and component summary provide an overview of the architecture of the UeiDaq library.

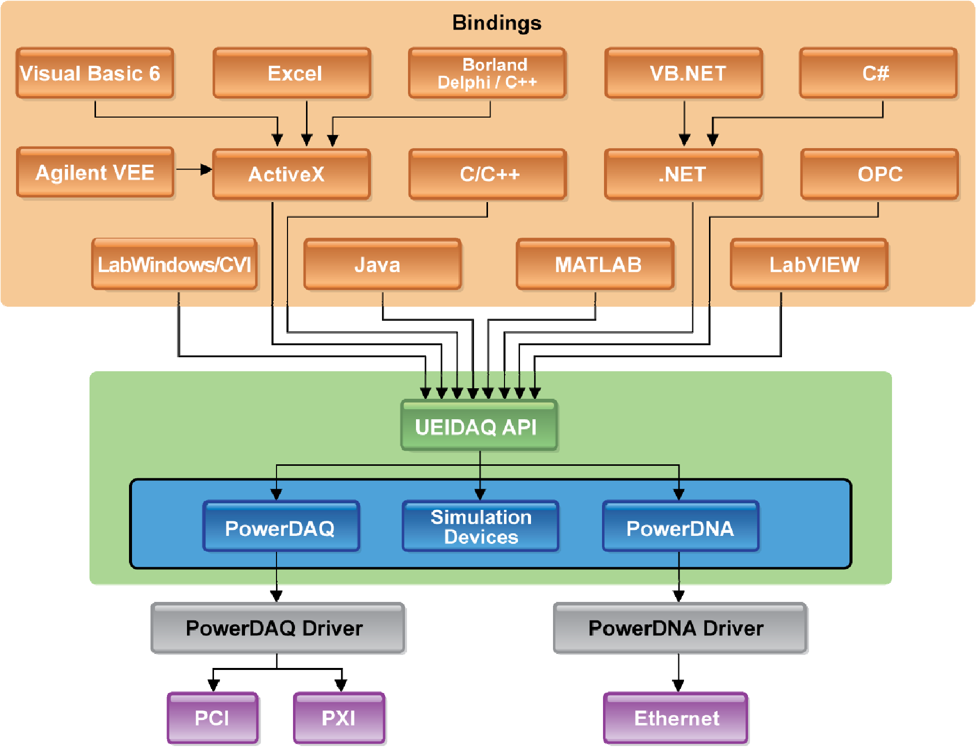


Figure 1 UeiDaq framework architecture

Hardware plugins	Used by the framework for communication with hardware and/or simulation devices. There is one plugin for each family of supported devices: PowerDAQ, PowerDNA, and simulation
Core	The heart of the framework. Detects available hardware plugins and implements a hierarchy of objects that abstracts communication with the plugins through a common interface
Bindings	Make the core's object oriented API available to various development environments and test and measurement software packages

2.2. Key Concepts

The UeiDaq framework can be programmed using various development environments:

- A set of classes for object-oriented environments such as the C++ language, .NET framework languages and ActiveX enabled environments.
- A set of functions for procedural languages such as ANSI C; the concept of classes is still present and objects are manipulated using handles and accessor functions.
- A set of VIs for the National Instruments LabVIEW package. A session refnum is used to specify and operate a session. Property nodes are used to access object parameters.
- An adapter for the MathWorks Matlab DAQ toolbox.
- An OPC server.
- An Excel add-in.

2.3. UeiDaq objects

2.3.1. Hierarchy

The UeiDaq framework API is object oriented and implements a hierarchy of classes to manage the communication with the data acquisition device.

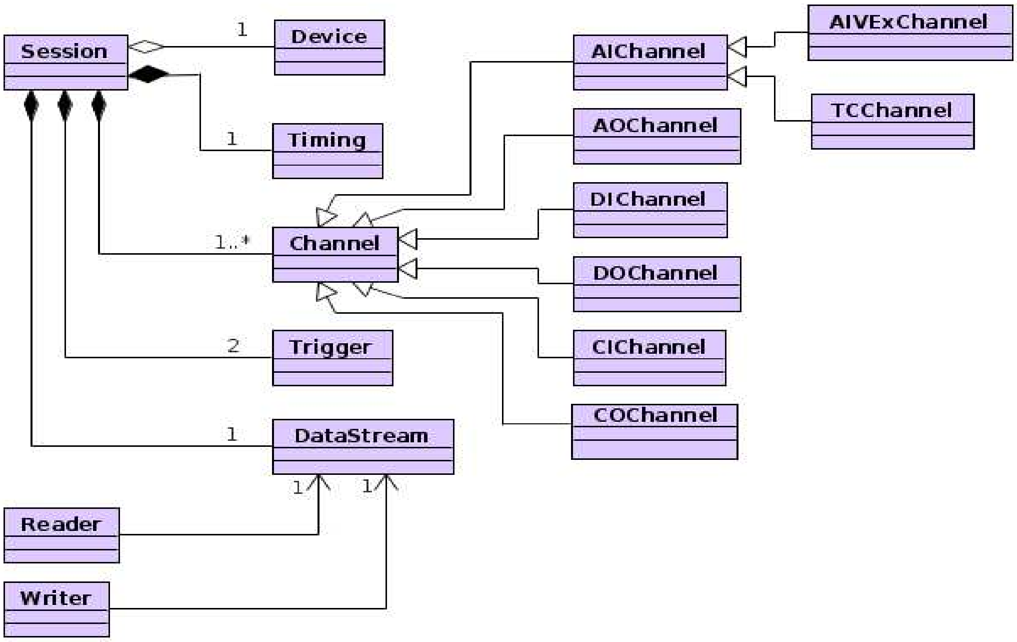


Figure 2 UeiDaq framework UML class diagram showing a subset of Channel classes

The UML class diagram in Figure 2 details the relationship between the various classes in the framework.

If you don't know UML don't panic. The most important thing to understand in the diagram is that the main object your application will interface with is the Session object.

The Session object acts as a container for other objects, such as Timing and Trigger. You only need to manage the lifetime of the Session object, which will in turn control the lifetime of its child objects.

The following sections provide a short description of each class. Please refer to the UeiDaq Framework Reference Manual for detailed information about each UeiDaq framework class.

2.3.2. Session

The Session manages communication with a data acquisition device subsystem. In order to do anything with your data acquisition device, you need to create a session first. You can then configure the session parameters and operate the session.

A session is tied to one subsystem at a time. If you want to use multiple subsystems (for example simultaneous analog input and output), you need to use multiple sessions.

2.3.3. Channels

A channel is part of the channel list associated with a Session. The channel object gives access to channel-specific parameters, such as gain for analog input channels.

Figure 2 shows that there is a derivation of the Channel class for each subsystem type: Analog Input, Analog Output, Digital Input, Digital Output, Counter Input, and Counter Output.

Note that Figure 2 also shows that the AIChannel class is also derived for specialized analog input measurement types: Voltage with excitation (AIVexChannel) and Thermocouple (TCChannel).

2.3.4. Devices

Each Session is tied to a subsystem that belongs to a specific device. The device object gives access to device properties such as its name, serial number, and calibration data.

2.3.5. DataStream

The DataStream object is used to transfer data between the device and the host.

2.3.6. Timing

The Timing object specifies how the subsystem clocks are configured. With it you can select whether you want to use a clock or not and whether the clock is internal or external.

2.3.7. Trigger

The Trigger object defines how the Session is started or stopped. You can configure the Session to start immediately or to wait for an external event to happen before starting.

2.3.8. Reader

The Reader object manages the transfer and formatting of data from a Session object. This object lifetime is independent of the session's lifetime; it needs to be linked to the session's DataStream object before it can start reading data.

The Reader class is designed to be derived. You can overload it to add functionalities to it; for example you could create an FFTReader object that would retrieve data from a session and compute an FFT before returning the result to the calling program.

2.3.9. Writer

The Writer object manages the transfer and formatting of data to a Session object. This object lifetime is independent of the Session's lifetime; it needs to be linked to the Session's DataStream object before it can start writing data.

3. Operating a Session

Figure 3 below is the typical flow of operations necessary to configure and execute a session:

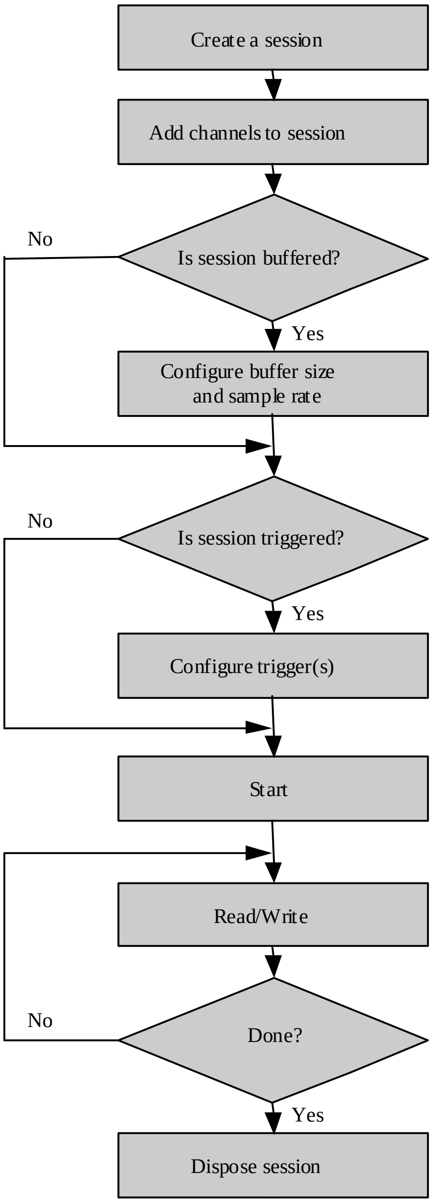


Figure 3 Flow of operations

The following sections provide examples to show how each operation is programmed with the UeiDaq framework. These examples are primarily coded using the C++ API; however, note that each development environment supported by the framework follows the same model.

3.1. Creating a session

To get started, you need to create a session object.

In C++, simply instantiate a CUeiSession object:

```
//Create a session on the stack
UeiDaq::CUeiSession mySession;

//Create a session on the heap
UeiDaq::CUeiSession* pMySession = new UeiDaq::CUeiSession();
```

In C, call the `UeiDaqCreateSession()` function:

```
SessionHandle mySession;
int error = UeiDaqCreateSession(&mySession);
```

From any managed .NET language, create a Session object:

```
UeiDaq.Session mySession = new UeiDaq.Session()
```

The next step is to create the channel list associated with the session.

3.2. Creating the channel list

The Session object has methods that select and configure channels that will be accessed during the session. All channels must belong to the same subsystem; for example, you cannot configure a single session with both Analog Input and Analog Output channels.

The framework uses resource strings to select which device, subsystem and channels to use within a session. The syntax for resource strings is similar to a web URL:
`<device class>://<IP address>/<Device Id>/<Subsystem><Channel list>`

The `device class` can be any of the following:

- “`pwrdaq`” for PowerDAQ PCI and PXI boards
- “`pdna`” for PowerDNA Ethernet I/O modules
- “`simu`” implements software simulation of various data acquisition devices, allowing you to start programming the framework without hardware.

The `device Id` must start with the string “dev” followed by the 0-based device number.

The `IP address` only needs to be specified if the device class requires it. PowerDNA devices need an IP address; PowerDAQ devices will ignore it, if specified.

The `subsystem` must be one of the following:

- AI: analog input session to measure voltage, current, temperature, strain, etc.
- AO: analog output session to generate voltage, current, etc.
- DI: digital input session to measure discrete signals.
- DO: digital output session to generate digital patterns.
- CI: counter input session to count discrete events, or measure pulse width and period.
- CO: counter output session to generate pulses and pulse trains.
- CAN: CAN bus session to send/receive data over a CAN bus.
- COM: serial port session to send/receive data over a serial port.
- CSDB: specialized serial session to send/receive data over a Commercial Serial Data Bus port.
- SSI: specialized serial session to send/receive data over a serial port using the Synchronous Serial Interface standard.
- HDLC: specialized serial session to send/receive data over a serial port that supports the high-level data link control protocol.
- ATX: ARINC-429 transmitter session to send data to an ARINC-429 bus.
- ARX: ARINC-429 receiver session to receive data from an ARINC-429 bus.
- MILB: MIL-1553B session to send/receive data over a MIL-1553 port.
- IRIG: Inter-range Instrumentation Group timing generation and synchronization session to capture IRIG data or to generate timing and synchronization signals for other device.
- VR: variable reluctance session to measure velocity and position and to count the teeth of a rotating gear-toothed wheel.
- SYNC: synchronization session to configure multi-layer, multi-chassis synchronization to a 1PPS pulse or the IEEE-1588 Precision Time Protocol standard.

The `channel list` is a comma-separated list of channels; you can also specify a range of channels using the lowest and highest channels separated by a colon. The channels in the channel list do not have to be sequential; you can even repeat channels multiple times.

Note that the resource string is not case sensitive. The following are examples of supported resource string formatting:

```
Pwrdaq://Dev0/Ai0:15
PDNA://192.168.100.2/Dev1/AO3
simu://Dev2/Di3
```

3.2.1. Analog input

There are several categories of Analog Input measurement: Voltage, Voltage with excitation, Current, Thermocouple, Resistance, Resistance temperature detectors (RTD), Linear variable differential transformer (LVDT), Synchro/Resolver, and Accelerometer.

NOTE: For each of the Analog Input sessions, timestamps may also be acquired by adding a *ts* channel as the last channel in the resource string.
For example: "*pdna://192.168.100.2/dev0/Ai0:3,ts*". The time unit for the acquired timestamp is seconds.

3.2.1.1. Voltage

Voltage measurements are configured using the Session object's method `CreateAIChannel().`

The gain to apply on each voltage channel is specified using low and high input range parameters. For example, if your device has an input range of –10/+10V with gains of 1,10,100,1000, specifying an input range of –0.1,+0.1 will turn on the gain of 100.

```
// Add 4 channels (0 to 3) to the channel list and configure
// them to measure a voltage between -10.0V and 10.0V in
// differential mode

MySession.CreateAIChannel("pwrdaq://dev0/Ai0:3",
                          -10.0, 10.0,
                          UeiAIChannelInputModeDifferential);
```

`CreateAIChannel` creates the specified number of channel child objects and initializes them with the specified input range and mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateAIChannel` several times in a row to add channels with different input ranges and modes to the list.

3.2.1.2. Voltage with excitation

Voltage measurements with excitation are configured using the Session object’s method `CreateAIVExChannel()`.

You use a voltage with excitation channel when measuring data from a sensor that requires excitation, such as a strain gauge or a load cell.

```
// Add 4 channels (0 to 3) to the channel list and configure
// them to measure a voltage between –0.05V and 0.05V from a
// full bridge sensor in differential mode. Also configure the
// excitation voltage to 10.0V and turn on ratiometric scaling.

MySession.CreateAIVExChannel(“pdna://192.168.100.2/dev0/Ai0:3”,
                             -0.05, 0.05,
                             UeiSensorBridgeFull,
                             10.0,
                             true,
                             UeiAIChannelInputModeDifferential);
```

`CreateAIVExChannel` creates the specified number of channel child objects and initializes them with the specified input range, sensor type, excitation voltage and input mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateAIVExChannel` several times in a row to add channels with different parameters to the list.

The parameter “`scale with excitation`”, determines whether the measurements are returned in V or in mV/V (measured voltage divided by excitation voltage).
Getting the measurements in mV/V is very useful with ratiometric sensors such as load cells that return a voltage proportional to the physical value measured and come with calibration values in mV/V.

You can only use voltage with excitation channels with devices that can provide excitation voltage, such as the DNA-AI-208 or DNx-AI-224.

3.2.1.3. Current

Current measurements are configured using the Session object’s method `CreateAICurrentChannel()`.

```
  // Add 4 channels (0 to 3) to the channel list and configure
  // them to measure a current between –10.0mA and 10.0mA in
  // differential mode

  MySession.CreateAICurrentChannel(“pdna://192.168.100.2/dev0/Ai0:3”,
                            -10.0, 10.0, UeiFeatureDisabled, // for CB
                            UeiAIChannelInputModeDifferential);
```

`CreateAICurrentChannel` creates the specified number of channel child objects and initializes them with the specified input range, circuit breaker enable, and mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateAICurrentChannel` several times in a row to add channels with different input ranges and modes to the list.

3.2.1.4. Thermocouple

Thermocouple measurements are configured using the Session object’s method `CreateTCChannel()`.

```
  // Add 4 channels (0 to 3) to the channel list and configure
  // them to measure a temperature between 0.0 and 1000.0 degrees C
  // from type J thermocouples, scale temperatures in Celsius
  // degrees and use a constant value of 25 degrees C for the
  // cold junction temperature

  MySession.CreateTCChannel(“pdna://192.168.100.2/dev0/Ai0:3”,
                            0, 1000.0,
                            UeiThermocoupleTypeJ,
                            UeiTemperatureScaleCelsius,
                            UeiColdJunctionCompensationConstant,
                            25.0,
                            “”,
                            UeiAIChannelInputModeDifferential);
```

`CreateTCChannel` creates the specified number of channel child objects and initializes them with the specified input range, thermocouple type, temperature scale, cold-junction compensation and input mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateTCChannel` several times in a row to add channels with different parameters to the list.

The gain to apply on each thermocouple channel is specified using low and high input range parameters. The unit of the range values is in degree C/F/K (depending on the selected temperature scale).

The measurements will be scaled in the unit specified by the “temperature scale” parameter. Depending on your hardware you can specify whether the scaling calculation will use a constant cold-junction temperature or measure it from a built in sensor. (See the DNA-STP-AI-U datasheet for information about our STP panels with built in CJC sensors.)

3.2.1.5. Resistance

Resistance measurements are configured using the Session object’s method `CreateResistanceChannel()`.

In order to measure a resistance, we need to know the amount of current flowing through it. We can then calculate the resistance by dividing the measured voltage by the known excitation current.

To measure the excitation current, we measure the voltage from a high precision reference resistor whose resistance is known.

```
// Add 4 channels (0 to 3) to the channel list and configure
// them to measure a resistance between 0.0 and 1000.0 Ohms.
// The resistive sensor is connected to the DAQ device using
// two wires, the excitation voltage is 5V

MySession.CreateResistanceChannel(“pdna://192.168.100.2/dev0/Ai0:3”,
                                  0, 1000.0,
                                  UeiTwoWires,
                                  5.0,
                                  UeiAIChannelInputModeDifferential);
```

`CreateResistanceChannel` creates the specified number of channel child objects and initializes them with the specified input range, wiring scheme, excitation voltage, reference resistor and input mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateResistanceChannel` several times in a row to add channels with different parameters to the list.

The measurements will be scaled in the unit specified by the “temperature scale” parameter.

3.2.1.6. RTD

Resistance temperature detector (RTD) measurements are configured using the Session object’s method `CreateRTDChannel()`.

RTD sensors are resistive sensors whose resistance varies with temperature. Knowing an RTD’s resistance at a given time, we can calculate the temperature using the “Callendar, Van-Dusen” equations.

RTD sensors are specified using the "alpha" (α) constant. It is also known as the temperature coefficient of resistance and symbolizes the resistance change factor per degree of temperature change. The RTD type is used to select the proper A, B and C coefficients for the Callendar Van-Dusen equation used to convert resistance measurements to temperature.

The exact same procedure used to configure resistance measurements in “`CreateResistanceChannel`” is used to configure the measurement of RTD resistances. In addition you must configure the RTD type and its nominal resistance at 0 deg. Celsius.

```
// Add 4 channels (0 to 3) to the channel list and configure
// them to measure a temperature between 0.0 and 200.0 deg. C.
// The RTD sensor is connected to the DAQ device using
// two wires, the excitation voltage is 5V,the alpha coefficient is
// 0.00385, and the nominal resistance is 100 ohms.
MySession.CreateRTDChannel(“pdna://192.168.100.2/dev0/Ai0:3”,
                           0, 1000.0,
                           UeiTwoWires,
                           5.0,
                           UeiRTDType3850,
                           100.0,
                           UeiTemperatureScaleCelsius,
                           UeiAIChannelInputModeDifferential);
```

`CreateRTDChannel` creates the specified number of channel child objects and initializes them with the specified input range, wiring scheme, excitation voltage, RTD type, temperature scale and input mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateRTDChannel` several times in a row to add channels with different parameters to the list.

The measurements will be scaled in the unit specified by the “temperature scale” parameter.

3.2.1.7. LVDT/RVDT

Linear variable differential transformer (LVDT) or rotational variable differential transformer (RVDT) measurements are configured using the Session object's method `CreateLVDTChannel()`.

LVDT/RVDT sensors are electromechanical transducers that measure displacement relative to a core position. The LVDT/RVDT sensor consists of a primary winding energized by a sine wave reference and two secondary windings, with the moveable core between the primary and secondary windings. As the core moves from the center position, an output voltage across the secondary windings is generated and used to determine the positional displacement (linear or rotational).

The `CreateLVDTChannel` method is used to program the input channels and parameters associated with each channel. This method will only work with devices that can provide an excitation waveform to the LVDT or RVDT.

```
// Configure 2 channels (0 to 1) on device 1. Specify
// -2.5v and 2.5v as the minimum / maximum range of the LVDT sensor
// and the sensor sensitivity specification at 153.65 mV/V/mm
// The LVDT sensor is connected to the DAQ device using
// a five-wire wiring scheme, and the excitation sine wave is
// configured to have an amplitude at 3.0Vrms, a frequency at 5kHz,
// and to be generated internally.
MySession.CreateLVDTChannel("pdna://192.168.100.2/Dev1/Ai0:1",
                           -2.5, //Minimum range
                            2.5, //Maximum range
                            153.65, //Sensor sensitivity
                            UeiLVDTFiveWires, //Wiring scheme
                            3.0, //Excitation voltage
                            5000.0, //Excitation frequency
                            false); //External excitation;
```

`CreateLVDTChannel` creates the specified number of channel child objects and initializes them with the specified input range, sensor sensitivity, wiring scheme, excitation voltage, excitation frequency, and excitation mode. The measurements will be scaled in the unit specified by the "sensor sensitivity" parameter. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateLVDTChannel` several times in a row to add channels with different parameters to the list.

See `CreateSimulatedLVDTChannel` in the Analog Output Channel section for information about configuring the device as a simulated LVDT/RVDT sensor.

3.2.1.8. SynchroResolver

Synchro/Resolver measurements are configured using the Session object’s method `CreateSynchroResolverChannel()`.

Synchro and resolver sensors are electromechanical transducers that measure the angular displacement of a rotating shaft.

The `CreateSynchroResolverChannel` method is used to program the input channels and parameters associated with each channel. This method will only work with devices that can provide an excitation waveform to the Synchro or Resolver sensor.

```
// Configure channel 0 on device 1 to acquire position measured
// by a synchro powered by a 600Hz sine waveform with
// amplitude of 10.0 VRMS excitation that is generated internally.

mySession.CreateSynchroResolverChannel(
                      "pdna://192.168.100.2/Dev1/Ai0",
                      UeiSynchroMode,
                      10.0,
                      600.0,
                      false);
```

`CreateSynchroResolverChannel` creates the specified number of channel child objects and initializes them with the specified synchro/resolver type, excitation voltage, excitation frequency, and excitation mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateSynchroResolverChannel` several times in a row to add channels with different parameters to the list.

See `CreateSimulatedSynchroResolverChannel` in the Analog Output Channel section for information about configuring the device as a simulated synchro or resolver sensor.

3.2.1.9. Accelerometer

Accelerometer measurements are configured using the Session object’s method `CreateAccelChannel()`.

ICP and IEPE sensors measure dynamic pressure, force, strain, or acceleration. The sensing element converts mechanical strain into a voltage, and the sensor is powered by a constant excitation source.

The `CreateAccelChannel` method is used to program the input channels and parameters associated with each channel. This method will only work with devices that can provide excitation current for ICP and IEPE sensors.

```
// Configure channels 0 to 3 on device 1 to acquire acceleration
// measured by a sensor with a sensitivity of 10 mV/g, powered by a
// current of 5mA. The gain of the device is adjusted to measure
// accelerations between -100.0g and +100.0g. Uses AC coupling and
// enables the low-pass anti-aliasing filter.

mySession.CreateAccelChannel(
                      "pdna://192.168.100.2/Dev1/Ai0:3",
                      -100.0,
                      100.0,
                      24.0,
                      UeiCouplingAC,
                      true);
```

`CreateAccelChannel` creates the specified number of channel child objects and initializes them with the gain, sensor sensitivity, excitation current, coupling setting, and low-pass filter setting. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateAccelChannel` several times in a row to add channels with different parameters to the list.

3.2.2. Analog output

There are several categories of Analog Output measurement: Voltage, Current, Waveform Protected Voltage Channel, Protected Current Channel, Simulated LVDT/RVDT, Simulated Synchro/Resolver, Simulated Thermocouple, and Simulated Resistance Temperature Detector.

3.2.2.1. Voltage

Voltage generation channels are configured using the Session object's method `CreateAOChannel()`.

```
// Add 4 channels (0 to 3) to the channel list with an expected
// minimum and maximum output voltage of -10v / 10v
MySession.CreateAOChannel("pwrdaq://dev0/Ao0:3", -10, 10);
```

`CreateAOChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.2.2. Current

Current generation channels are configured using the Session object's method `CreateAOCurrentChannel()`.

```
// Add 2 channels (0 to 1) to the channel list with an expected
// minimum and maximum output current of 0mA / 10mA
MySession.CreateAOCurrentChannel
                    "pdna://192.168.100.2/Dev1/Ao0:1",
                     0, 10);
```

`CreateAOCurrentChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.2.3. Waveform

Waveform generation channels are configured using the Session object's method `CreateAOWaveformChannel()`.

```
// Add 3 channels (0 to 2) to the channel.
// Set the clock from the PLL as the source of the clock used to time
// the main DAC.
// Specify using software for a DC offset on the DAC
// Specify external triggers or clock sources will not be routed over
// the SYNC lines.

MySession.CreateAOWaveformChannel(
                   "pdna://192.168.100.2/Dev0/Ao0:2",
                    UeiAOWaveformClockSourcePLL,
                    UeiAOWaveformOffsetClockSourceSW,
                    UeiAOWaveformClockRouteNone);
```

`CreateAOWaveformChannel` creates the specified number of channel child object(s) and sets up clocking and triggers. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.2.4. Protected voltage

Protected voltage generation channels are configured using the Session object’s method `CreateAOProtectedChannel()`.

Protected AO channels are available on certain devices, such as the DNA-AO-318. The amount of current flowing through each output is monitored at the given rate and must stay within the specified range, otherwise the device circuit breaker will open. Users can specify whether the device should attempt to reset the circuit breaker and how often the reset attempt should be made.

`CreateAOProtectedChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.2.5. Protected current

Current generation channels with over/under range protection are configured using the Session object’s method `CreateAOProtectedCurrentChannel()`.

Protected AO current channels are available on certain devices, such as the DNA-AO-318-020. The amount of current flowing through each output is monitored at the given rate and must stay within the specified range, otherwise the device circuit breaker will open. Users can specify whether the device should attempt to reset the circuit breaker and how often the reset attempt should be made.

`CreateAOProtectedChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.2.6. Simulated LVDT/RVDT

Simulated LVDT/RVDT generated channels are configured using the Session object’s method `CreateSimulatedLVDTChannel()`.

The `CreateSimulatedLVDTChannel` method is used to program the channels and parameters associated with each channel. This method will only work with devices that can provide an excitation waveform.

```
// Configure channel 0 on device 1 to simulate position measurement
// given by a LVDT with a sensitivity of 24 mV/V/mm, powered by a
// 600Hz sine waveform with amplitude of 10.0V RMS.
MySession.CreateSimulatedLVDTChannel(
               "pdna://192.168.100.2/Dev1/Ao0",
               24, // sensor sensitivity
               UeiLVDTFiveWires, //Wiring scheme
               10.0, //Excitation voltage
               600.0); //Excitation frequency
```

`CreateSimulatedLVDTChannel` creates the specified number of channel child objects and initializes them with the specified sensor sensitivity, wiring scheme, and excitation voltage and frequency. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

See `CreateLVDTChannel` in the Analog Input Channel section for information about configuring the device as a LVDT/RVDT input device.

3.2.2.7. Simulated synchro/resolver

Simulated Synchro or Resolver generated channels are configured using the Session object's method `CreateSimulatedSynchroResolverChannel()`.

The `CreateSimulatedSynchroResolverChannel` method is used to program the channels and parameters associated with each channel. This method will only work with devices that can provide an excitation waveform.

```
// Configure channel 0 on device 1 to simulate position measurement
// returned by a synchro powered by a 600Hz sine waveform with
// amplitude of 10.0 VRMS.
mySession.CreateSimulatedSynchroResolverChannel(
                      "pdna://192.168.100.2/Dev1/Ai0",
                      UeiSynchroMode,
                      10.0,
                      600.0,
                      false);
```

`CreateSimulatedSynchroResolverChannel` creates the specified number of channel child objects and initializes them with the specified synchro/resolver type, excitation voltage, excitation frequency, and excitation mode. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateSimulatedSynchroResolverChannel` several times in a row to add channels with different parameters to the list.

See `CreateSynchroResolverChannel` in the Analog Input Channel section for information about configuring the device as a synchro or resolver input device.

3.2.2.8. Simulated thermocouple

Simulated Thermocouple generated channels are configured using the Session object's method `CreateSimulatedTCChannel()`.

The `CreateSimulatedTCChannel` method is used to program the channels and parameters associated with each channel. This method can only be used on devices that support simulated thermocouple functionality, such as the DNx-TC-378 devices.

```
// Configure channel 0 on device 1 to simulate a type K thermocouple
//    Temperatures to simulate are provided in Celsius and
//    CJC sensors are read and offsets compensated for.
mySession.CreateSimulatedTCChannel(
                      "pdna://192.168.100.2/Dev1/Ao0",
                      UeiThermocoupleTypeK,
                      UEITemperatureScaleCelsius,
                      true);
```

`CreateSimulatedTCChannel` creates the specified number of channel child objects and initializes them with the specified thermocouple type, temperature scale, and CJC enable configuration. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateSimulatedTCChannel` several times in a row to add channels with different parameters to the list.

3.2.2.9. Simulated RTD

Simulated Resistance Temperature Detector (RTD) generated channels are configured using the Session object’s method `CreateSimulatedRTDChannel()`.

The `CreateSimulatedRTDChannel` method is used to program the channels and parameters associated with each channel. This method can only be used on devices that support simulated RTD functionality, such as the DNx-RTD-388 devices.

```
// Configure channel 0 on device 1 to simulate an RTD sensor
//    with an α constant equal to .003850 Ω/Ω/°C and
//    a nominal resistance of 100.0
//    Temperatures to simulate are provided in Celsius.
mySession.CreateSimulatedRTDChannel(
                      "pdna://192.168.100.2/Dev1/Ao0",
                      UeiRTDType3850,
                      100.0,
                      UEITemperatureScaleCelsius);
```

`CreateSimulatedRTDChannel` creates the specified number of channel child objects and initializes them with the specified RTD type, nominal resistance, and temperature scale. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

You can call `CreateSimulatedRTDChannel` several times in a row to add channels with different parameters to the list.

3.2.3. Digital input

Digital input channels are configured using the session object's method `CreateDIChannel()`.

```
// Add 4 ports (0 to 3) to the input port list
MySession.CreateDIChannel("pwrdaq://dev0/Di0:3");
```

`CreateDIChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).
Each entry in the channel list correspond to a digital port (usually 16 to 32 input lines per port). ***Note: you cannot specify individual digital input lines directly***.

Note that for devices with bi-directional ports, this operation configures the specified ports as input.

3.2.3.1. *Digital industrial input*

Digital industrial input channels are configured using the session object's method `CreateDIIndustrialChannel()`.

Digital Industrial channels are only available on certain DIO devices, such as the DNA-DIO-449, which supports industrial input ranges of ±150 VDC and 0-150 VAC.

You can program the levels at which the input line changes state as well as configure a digital filter to eliminate glitches and spikes.

```
// Add 2 ports (0 and 1) to the input port list
// Set low threshold to 1.5V, high threshold to 3.5V
// Set minimum input width to 0.01ms
MySession.CreateDIIndustrialChannel(
                  "pdna://192.168.100.2/dev0/Di0,1", 1.5, 3.5, 0.01);
```

Parameters include the following:

- `lowThreshold`: the low hysteresis threshold
- `highThreshold`: the high hysteresis threshold
- `minPulseWidth`: the digital filter minimum pulse width in ms. Use 0.0 to disable digital input filter.

`CreateDIIndustrialChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).
Each entry in the channel list correspond to a digital port (usually 16 to 32 input lines per port). ***Note: you cannot specify individual digital input lines directly***.

3.2.4. Digital output

Digital output channels are configured using the Session object's method `CreateDOChannel()`.

```
// Add 4 ports (0 to 3) to the output port list
MySession.CreateDOChannel(“pwrdaq://dev0/Do0:3”);
```

`CreateDOChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).

Each entry in the channel list correspond to a digital port. ***Note**: **you cannot specify individual digital output lines directly***.

Note that for devices with bi-directional ports, this operation configures the specified ports as output.

3.2.4.1. Digital output protected

Protected digital output channels are configured using the Session object's method `CreateDOProtectedChannel()`.

Protected DO channels are available on certain devices, such as the DNx-DIO-432, DNx-DIO-433, and DNA-DIO-462.

The amount of current flowing through each digital line is monitored at the specified rate and must stay within the specified range, otherwise the device will automatically open the circuit acting as a breaker.

You can specify whether the device should attempt to reestablish the circuit and how often it should try to do so.

```
// Add 1 port to the output port list
// Set current limits to [0.0, 0.2A]
// Set current sample rate to 100Hz
// Enable auto-retry at 10Hz
MySession.CreateDOProtectedChannel(“pdna://192.168.100.2/dev0/Do0”,
0.0, 0.2, 100.0, true, 10.0);
```

Parameters include the following:

- `resource`: device and channel(s) to add to the list
- `underCurrentLimit`: minimum amount of current allowed in Amps
- `overCurrentLimit`: maximum amount of current allowed in Amps

- `currentSampleRate:` current sampling rate. Determines how fast the breaker react after an over or under-current condition
- `autoRetry:` specifies whether the device will attempt to reestablish the circuit after an over or under-current condition
- `retryRate:` number of retries per second

`CreateDOProtectedChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).

Each entry in the channel list corresponds to a digital port. ***Note: you cannot specify individual digital output lines directly***.

3.2.5. Counter input

Counter input channels are configured using the Session object’s method `CreateCIChannel()`.

```
// Add 1 counter (0) to the input counter list and configure
// it to count events coming on its input pin, the gate will
// be controlled by software, the signal coming on its input
// pin will be divided by 10 and inverted before being counted.

MySession.CreateCIChannel(“pwrdaq://dev0/Ci0”, UeiCounterSourceInput,
UeiCounterModeCountEvents, UeiCounterGateInternal, 10, true);
```

`CreateCIChannel` creates the specified number of channel child object(s) and configure the counter(s) mode, input, gate, and clock divider. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.5.1. Quadrature encoder input

Counter input channels are configured using the Session object's method `CreateQuadratureEncoderChannel()`.

```
// Add 1 counter (0) to the input counter list and configure
// it to start with an initial count of 102190.
// Set decoding type to 1x and enable reset of the position when
// the Z input goes high
MySession.CreateQuadratureEncoderChannel(
                    "pdna://192.168.100.2/dev0/Ci0", 102190,
                    UeiQuadratureDecodingType1x, true,
                    UeiQuadratureZeroIndexPhaseZHigh);
```

Parameters include the following:

- `resource:` device and channel(s) to add to the list
- `initialPosition:` initial number of pulses when the session starts
- `decodingType:` decoding type 1x, 2x or 4x
- `enableZeroIndexing:` enable or disable resetting the measurement when a zero index event is detected
- `zeroIndexPhase:` specifies the states of A, B and Z inputs that will generate a zero index event

`CreateQuadratureEncoderChannel` creates the specified number of channel child object(s). (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.6. Timer/Frequency (Counter) output

Timer/Frequency (Counter) output channels are configured using the Session object's method `CreateCOChannel()`.

```
// Add 1 counter (1) to the output counter list and configure
// it to generate a pulse train timed by its internal clock, pulses
// will stay low for 102 clock ticks and high for 508 ticks,
// the gate will be controlled by software, the timer's clock
// will be divided by 10 and the signal generated won't be inverted.

MySession.CreateCOChannel("pwrdaq://dev0/Co1", UeiCounterSourceInput,
UeiCounterModeGeneratePulseTrain, UeiCounterGateClock, 102, 508, 10,
false);
```

`CreateCOChannel` creates the specified number of channel child object(s) and configure the timer(s) mode, clock, gate, generated pulses shape and clock divider. (Refer to the UeiDaq Framework Reference Manual for more details about this method).

3.2.7. Variable reluctance

Variable Reluctance (VR) channels are configured using the Session object's method “ `CreateVRChannel()`.

A VR-session type measures velocity and position and counts the teeth of a geartoothed wheel rotating next to a variable reluctance sensor. Each VR channel can convert the output signal of the VR sensor to a pulse train that can be counted and whose frequency can be measured.

The following call configures channel 0 of a VR-608 set as device 1:

```
// Configure session to write to channel 0 on device 1
session.CreateVRChannel("pdna://192.168.100.2/Dev1/vr0", vrMode);
```

The VR mode parameter can be set to any of the following:

- `UeiVRModeCounterTimed`: Count number of teeth detected during a timed interval
- `UeiVRModeCounterNPulses`: Measure the time taken to detect N teeth. Number of teeth needs to be set separately.
- `UeiVRModeZPulse`: Measure the number of teeth and the time elapsed between two Z pulses. The Z tooth is usually a gap or a double tooth on the encoder wheel

In addition you can set additional parameters using the channel object methods
(or a property node under LabVIEW):

- `Zero Crossing mode`: Configures the method used to detect zero crossing.
  A Zero crossing identifies the point in time where the VR sensor output voltage transitions from positive to negative. This point is the center of the tooth is transitioning past the front of the VR sensor.

  ```
  // Set ZC mode to let chip automatically program zero crossing
  // (alternate mode is UeiZCModeFixed)
  pVrChan-> SetZCMode(UeiZCModeChip);
  ```

- `Zero Crossing level`: Configures the threshold to detect zero crossing -only when ZC mode is set to `UeiZCModeFixed`.

  ```
  // Set ZC level to 2.0V
  pVrChan-> SetZCLevel(2.0);
  ```

- `Adaptive Peak Threshold mode`: Configures the adaptive peak threshold (APT) mode on variable reluctance sessions. APT finds the point in time where the VR sensor output voltage falls below a certain threshold. This point marks the beginning of the gap between two teeth on the gear-wheel.

  ```
  // Set APT mode to let chip automatically program peak threshold
  // (alternate mode is UeiAPTModeFixed)
  pVrChan-> SetAPTMode(UeiAPTModeChip);
  ```

- `APT threshold`: Configures the APT fixed threshold. This parameter is only used when APT mode is set to UeiAPTModeFixed.

  ```
  // Set APT threshold to 4 volts
  pVrChan->SetAPTThreshold(4);
  ```

- `Number of teeth`: Configures the number of teeth on the encoder wheel. This parameter is required to measure the position when the mode is set to Timed or NPulses. It is ignored in ZPulse mode.

  ```
  // Set number of teeth to 60
  pVrChan->SetNumberOfTeeth(60);
  ```

- `Size of Z-tooth`: For gear-wheels with an index/z-tooth you can set the number of missing teeth (from -1 to -3) or long teeth (1 to 3) with this function to add them to the total number of teeth:

  ```
  // Set width of the z-tooth to 0 teeth
  // (gear wheel has no index tooth)
  pVrChan->SetZToothSize(0);
  ```

- `Timed Mode Rate`: When a channel is in `UeiVRModeCounterTimed` mode the channel’s counter-timer unit saves a new read point into the input buffer at a rate / time-base that you configure with `SetTimedModeRate()`.
  This is not the sampling rate (see `SetADCRate` call) which is the same for all channels, but rather the rate at which data is stored:

  ```
  // Set the read frequency to 10 Hz in UeiVRModeCounterTimed mode
  pVrChan->SetTimedModeRate(10);
  ```

`CreateVRChannel` creates the specified number of port child object(s) and configures the above communication parameters.
Note that you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.8. Serial port

Serial ports are configured using the Session object’s method `CreateSerialPort()`.

```
// Configure 4 serial ports on the serial device in device 1
// to use the same communication parameters: 57600 bps, 8 data
// bits, no parity and 1 stop bit.

mySession.CreateSerialPort(“pdna://192.168.100.2/Dev1/COM0:3”,
UeiSerialBitsPerSecond57600, UeiSerialDataBits8, UeiSerialParityNone,
UeiSerialStopBits1);
```

`CreateSerialPort` creates the specified number of port child object(s) and configures the communication parameters: bit per second, number of data bits, parity and number of stop bits.

Note that there is no multiplexing of data (contrary to what’s being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.9. CSDB port

Commercial Standard Digital Bus (CSDB) ports are configured using the Session object's method `CreateCSDBPort()`. This method can only be used on devices that support CSDB functionality, such as the DNx-CSDB-509.

```
// Configure 2 CSDB ports on the serial device in device 1
// to use the same communication parameters: 50000 bps, odd parity,
// 10 µs inter-byte delay, 200 µs inter-block delay,
// and 100000 µs frame period.

mySession.CreateCSDBPort("pdna://192.168.100.2/Dev1/CSDB0,1",
                         50000,
                         1,
                         blockSize,
                         numMessagesPerFrame,
                         10,
                         200,
                         100000);
```

Parameters include the following:

- Bit rate: bits per second
- Parity: odd or even
- CSDB message block size: the number of bytes in a message block including the address and status bytes
- Number of CSDB message blocks per frame
- Inter-byte delay within a message block in microseconds
- Inter-block delay within a frame in microseconds
- Period at which the frame is transmitted in microseconds

`CreateCSDBPort` creates the specified number of port child object(s) and configures the above communication parameters.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.10. SSI port

Synchronous Serial Interface (SSI) ports are configured using the Session object's method `CreateSSIMasterPort()` and/or `CreateSSISlavePort()`. These methods can only be used on devices that support SSI functionality, such as the DNx-SL-514.

```
// Configure 2 SSI master ports on the serial device in device 1
//      use 125000 bps, 8-bit word size, enable clock,
//      do not enable termination resistor,
//      set the pause time to 10000 µs, set the transfer timeout to
//      16.02 µs, and set the bit update time to 0.45 µs

mySession.CreateSSIMasterPort("pdna://192.168.100.2/Dev1/SSI0,1",
                               125000,
                               8,
                               TRUE,
                               FALSE,
                               10000.0,
                               16.03,
                               0.45);

// Configure 2 SSI slave ports on the serial device in device 2
//      use 125000 bps, 8-bit word size, enable clock,
//      do not enable termination resistor,
//      set the pause time to 10000 µs, set the transfer timeout to
//      16.09 µs, and set the bit update time to 0.0 µs

mySession.CreateSSISlavePort("pdna://192.168.100.2/Dev2/SSI2,3",
                               125000,
                               8,
                               TRUE,
                               FALSE,
                               10000.0,
                               16.00,
                               0.0);
```

`CreateSSIMasterPort` and `CreateSSISlavePort` creates the specified number of port child object(s) and configures the above communication parameters.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

Also note that data received over the master port can be optionally timestamped. Pass a TRUE to the `CUeiSSIMasterPort` member function to enable timestamps:

```
masterPort->EnableTimestamping(true);
```

3.2.11. HDLC port

High-level Data Link Control (HDLC) ports are configured using the Session object's method `CreateHDLCPort()`. This method can only be used on devices that support HDLC functionality, such as the DNx-SL-504.

```
// Configure HDLC ports 2&3 on the serial device in device 1

mySession.CreateHDLCPort("pdna://192.168.100.2/Dev1/hdlc2,3",
                         UeiHDLCPortRS232,
                         100000,
                         UeiHDLCPortEncodingNRZ,
                         UeiHDLCPortCRCNone,
                         UeiHDLCPortClockBRG,
                         UeiHDLCPortClockExternal);
```

Parameters include the following:

- Physical interface: (RS-232, RS-422, RS-485)
- Bits per second: the number of bits per second transmitted / received
- Encoding: the method used to encode bits, (e.g., NRZ, NRZI, NRZIMark, etc.)
- CRC: the method used to calculate the cyclic redundancy code (error checking)
- TX clock source: clock source used to synchronize transmitter
- RX clock source: clock source used to synchronize receiver

`CreateHDLCPort` creates the specified number of port child object(s) and configures the above communication parameters.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.12. CAN bus

CAN ports are configured using the session object’s method `CreateCANPort()`.

```
// Configure the four CAN ports available on the PowerDNA CAN-503
// layer to use the same parameters: 250000 bps, extended, frames,
// normal port operation and acceptance mask and code.

mySession.CreateCANPort(“pdna://192.168.100.2/Dev1/CAN0:3”,
UeiCANBitsPerSecond250K, UeiCANFrameExtended, UeiCANPortModeNormal,
0xffffffff, 0x0);
```

`CreateCANPort` creates the specified number of port child object(s) and the communication parameters: bit per second, frame format, port operation mode, etc.

Note that there is no multiplexing of data (contrary to what’s being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per CAN port to be able to read and write from/to each port in the port list.

3.2.13. ARINC-429 bus

There are two methods to configure ARINC-429 input (receiver) and output (transmitter) ports.

3.2.13.1. ARINC-429 receiver ports

ARINC input ports are configured using the session object’s method `CreateARINCInputPort()`.

```
// Configure 2 ARINC-429 input ports to receive at 100000 bps,
// using no parity and disabling the SDI filter
mySession.CreateARINCInputPort("pdna://192.168.100.2/Dev0/ARX0,1",
UeiARINCBitsPerSecond100000, UeiARINCParityNone, false, 0);
```

`CreateARINCInputPort` creates the specified number of port child object(s) and the communication parameters: bit per second, parity and SDI filter setting.

Note that there is no multiplexing of data (contrary to what’s being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader per ARINC-429 input port to be able to read from each port in the port list.

3.2.13.2. ARINC-429 transmitter ports

ARINC input ports are configured using the session object's method `CreateARINCOutputPort()`.

```
// Configure 2 ARINC-429 output ports to receive at 100000 bps,
// using no parity

mySession.CreateARINCOutputPort("pdna://192.168.100.2/Dev0/ATX0,1",
UeiARINCBitsPerSecond100000, UeiARINCParityNone);
```

`CreateARINCOutputPort` creates the specified number of port child object(s) and the communication parameters: bit per second and parity.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one writer per ARINC-429 output port to be able to write to each port in the port list.

3.2.14. MIL-1553 bus

MIL-1553 ports are configured using the session object’s method `CreateMIL1553Port()`.

```
session.CreateMIL1553Port(“pdna://192.168.100.2/dev0/milb0”,
                          UeiMIL1553CouplingTransformer,
                          UeiMIL1553OpModeBusController);
```

The created MIL1553 port can be used in one of three modes:

- `UeiMIL1553OpModeBusMonitor` – port allows receiving ongoing activity on the bus using `CUeiMIL1553Reader` object. In this mode of operation `CUeiMIL1553Writer` object also allows to send unscheduled continuous data on the bus.
- `UeiMIL1553OpModeRemoteTerminal` – port allows to program remote terminals and send and receive data from the remote terminal data memory.
- `UeiMIL1553OpModeBusController` – port allows to program bus controller scheduler and send and receive data from and to bus controller data memory.

The port can be used in one of the four coupling modes:

- `UeiMIL1553CouplingDisconnected`– port is completely disconnected from the bus
- `UeiMIL1553CouplingTransformer` – normal mode of operations
- `UeiMIL1553CouplingLocalStub` – isolation coupler of the layer, requires special; version of the hardware
- `UeiMIL1553CouplingDirect -` direct connection without transformer

You will need to create one reader and one writer per port to access port data in any modes of operation.

A user can select which bus to transmit data using *`CUeiMIL1553Port::SetRxBus()`* method. Notice that for transmission bus A or bus B should be selected (default is bus A) while for bus monitor both buses are enabled by default.

3.2.15. IRIG

There are several methods to configure IRIG channels: IRIG timekeeper, IRIG input and output channels, and IRIG DO TTL channels.

3.2.15.1. IRIG timekeeper

IRIG timekeeper ports are configured using the Session object's method `CreateIRIGTimeKeeperChannel()`. This method can only be used on devices that support IRIG functionality, such as the DNx-IRIG-650.

The following sample code shows how to configure the time keeper channel of an IRIG-650 set as device 1:

```
// Configure the time keeper

CUeiIRIGTimeKeeperChannel* pTKChannel =
        irigSession.CreateIRIGTimeKeeperChannel(
                    "pdna://192.168.100.2/Dev1/Irig0",
                    UeiIRIG1PPSInternal,
                    autoFollow);
```

Parameters include the following:

- `1PPS source`: source of the 1PPS signal (internally generated 1PPS is selected in the example above)
- `Auto-follow`: When auto-Follow is enabled, if the external 1PPS source does not deliver pulses (because of a break in timecode transmission, for example), the Timekeeper can switch to internal timebase when externally derived one is not available.

In addition you can set additional parameters using the channel object methods (or a property node under LabVIEW):

- `Nominal Value enabled`: Select whether to use nominal period (i.e. 100E6 pulses of 100MHz base clock) or the period measured by timekeeper (it measures and averages number of base clock cycles between externally derived 1PPS pulses when they are valid).
  pTKChannel->EnableNominalValue(true);

- `Sub PPS enabled`: Select whether external timebase is slower than 1PPS or is not derived from the timecode.
  //disable sub pps
  pTKChannel->EnableSubPPS(false);

- `Initial time`: The initial time loaded in time keeper.

```
// initial time
tUeiANSITime now;
pTKChannel->SetInitialTime(now)
```

`CreateIRIGTimeKeeperChannel` creates the specified number of port child object(s) and configures the above communication parameters.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.15.2. IRIG output

IRIG time code output ports are configured using the Session object's method `CreateIRIGOutputChannel()`. This method can only be used on devices that support IRIG functionality, such as the DNx-IRIG-650.

The following sample code shows how to configure the time code output channel of a IRIG-650 set as device 1:

```
// configure the time code output

CUeiIRIGOutputChannel* pOutChannel =
        irigSession.CreateIRIGOutputChannel(
                    "pdna://192.168.100.2/Dev1/Irig0",
                    timeCodeformat);
```

Parameters include a Timecode Format parameter:

- `UeiIRIGTimeCodeFormatA`: IRIG-A
- `UeiIRIGTimeCodeFormatB`: IRIG-B
- `UeiIRIGTimeCodeFormatE_100Hz`: IRIG-E 100Hz
- `UeiIRIGTimeCodeFormatE_1000Hz`: IRIG-E 1000Hz
- `UeiIRIGTimeCodeFormatG`: IRIG-G

In addition you can set the following parameter using the channel object methods (or a property node under LabVIEW):

- `Start when input is valid`: If selected, the output time coder waits for the input time decoder to receive a valid time code before starting.

```
// start when input is valid
pOutChan->EnableStartWhenInputValid(true);
```

`CreateIRIGOutputChannel` creates the specified number of port child object(s) and configures the above communication parameters.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.15.3. IRIG input

IRIG time code input ports are configured using the Session object's method `CreateIRIGInputChannel()`. This method can only be used on devices that support IRIG functionality, such as the DNx-IRIG-650.

The following sample code shows how to configure the time code input channel of a IRIG-650 set as device 1:

```
// Configure the time code input

CUeiIRIGInputChannel* pInChannel =
        irigSession.CreateIRIGInputChannel(
                    "pdna://192.168.100.2/Dev1/Irig0",
                    decoderInput,
                    timeCodeformat);
```

Parameters include the following:

- `Decoder Input Type`: time code format used by the input signal, (e.g., `UeiIRIGDecoderInputAM`, `UeiIRIGDecoderInputManchesterRF0`)
- `Timecode Format`: input where the timecode signal is connected, (e.g., `UeiIRIGTimeCodeFormatB`: IRIG-B)

In addition you can set additional parameters using the channel object methods (or a property node under LabVIEW):

- `Idle character`: Determines whether idle character in the timing byte stream are accepted.
  ```
  // disable idle character
  pInChan->EnableIdleCharacter(false);
  ```

- `Single P0 Marker: Determines whether to use only one marker P0 in the timing byte stream.`
  ```
  // Enable single P0 marker
  pInChan->EnableSingleP0Marker(true);
  ```

`CreateIRIGInputChannel` creates the specified number of port child object(s) and configures the above communication parameters.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.15.4. IRIG TTL outputs

IRIG TTL output channels are configured using the Session object's method `CreateIRIGDOTTLChannel()`. All four TTL outputs are represented using only one channel object.This method can only be used on devices that support IRIG functionality, such as the DNx-IRIG-650.

The following sample code shows how to configure the TTL output channels of an IRIG-650 set as device 1:

```
// configure the TTL output

CUeiIRIGDOTTLChannel* pTTLChan =
        irigSession.CreateIRIGDOTTLChannel(
                    "pdna://192.168.100.2/Dev1/Irig0",
                    Line0Source,
                    Line1Source,
                    Line2Source,
                    Line3Source);
```

Each of the TTL line sources (`Line[0:3]Source`) is configured with a specific TTL pattern, such as AM-> NRZ output (`UeiIRIGDOTTLAMtoNRZ`), custom frequency output `(UeiIRIGDOTTL1PPS),` the 1PPS pulse `(UeiIRIGDOTTL1PPS),` and more.

In addition you can set the following parameter using the channel object methods (or a property node under LabVIEW):

- `40 ns pulse`: Set pulse width to 40ns instead of the default 60µs.
  ```
  // enable 40 ns pulses on TTL line 1
  pTTLChan->Enable40nsPulse(1, true);
  ```
- `Use one or two TTL drivers`: Enables the second TTL driver (provides stronger driving capabilities and sharper edges).
  ```
  // enable dual TTL driver on all outputs
  pTTLChan->EnableTwoTTLBuffers(true);
  ```
- `Drive Sync line`: Drive sync line instead of TTL output line.
  ```
  // configure line 3 to drive sync line 3
  pTTLChan->DriveSyncLine(3, true);
  ```

`CreateIRIGDOTTLChannel` creates the specified number of port child object(s) and configures the above communication parameters.

Note that there is no multiplexing of data (contrary to what's being done with AI, AO, DI, DO, CI and CO sessions): you need to create one reader and one writer object per port to be able to read and write from/to each port in the port list.

3.2.16. SYNC

A SYNC session is a specialized session specifically for synchronization. The sync session configures hardware on the CPU board of any cube(s) and/or rack(s) that require synchronization, enabling multi-layer, multi-chassis synchronization to a one pulse per second (1PPS) reference signal or to the IEEE-1588 Precision Time Protocol (PTP) standard.

Refer to Chapter 5 for detailed information about using a SYNC session, as well as more information regarding all synchronization-specific capabilities.

3.3. Configuring the timing

3.3.1. Point by point timing mode

Point by point timing mode uses a software clock to time the data acquisition/generation and is well suited for slow speed operations (less than 500Hz).

The session object's method `ConfigureTimingforSimpleIO()` is called to perform a software timed acquisition or generation.

```
mySession.ConfigureTimingForSimpleIO();
```

In point by point mode, data is read or written from/to the device, one scan at a time, each channel in the channel list is acquired or updated once per software request.

The acquired data is returned as an array containing one sample per channel.
The data to be generated must be passed as an array containing one value for each channel.

3.3.2. Buffered timing mode

Buffered timing mode uses a hardware clock to time the data acquisition/generation. This mode is required to perform high-speed data acquisition or generation.

The session object’s method `ConfigureTimingForBufferedIO()` is called to perform a hardware timed acquisition or generation.

```
// Configure the timing object to acquire or generate 1000
// samples per channel at 5000 Hz, use the device’s internal clock
// rising edges in continuous mode.
mySession.ConfigureTimingForBufferedIO(1000,
UeiTimingClockSourceInternal, 5000.0, UeiDigitalEdgeRising,
UeiTimingDurationContinuous);
```

In buffered mode, samples are read or written from/to the device in buffers. The samples from each channel of the channel list are interleaved in the buffer.
With LabVIEW and the ActiveX interface, the samples are stored in a 2D array with all samples from the same channel in the same row.

3.3.2.1. Advanced Circular Buffer

Advanced circular buffer (ACB) is a UEI developed buffered timing mode that uses multiple internal buffers (called frames) for data transfers.

In the case of an input session, the driver continuously fills buffers with data from the device and sends them to the user application, which can process them at its own pace. For an output session, the user application can fill multiple buffers and send them to the driver without waiting for the device to be ready to accept them.

Having multiple buffers helps to avoid gaps in the acquired or generated data, especially under non real-time operating systems such as Windows. Note that general purpose operating systems sometimes takes away CPU cycles from your application to run various background tasks, such as services, servers, network requests, virus scanning, etc. During that time, the driver keeps filling-up buffers with data that your application will receive as soon as the operating system allows it to run.
Another advantage of circular buffers is the situation when your application takes more time to process a buffer than the time it took to acquire it. It allows your application to later catch up and avoid losing any data. Of course this will only work if this behavior is not recurrent.

The size of each buffer is equal to the buffer size parameter specified when calling `ConfigureTimingForBufferedIO()`. The default number of buffers is 4. You can change the number of buffers by calling the DataStream object's method "`SetNumberOfFrames`".

The circular buffer uses read and write pointers to keep track of the state of both reader and writers. The write pointer position moves along as the writer is writing new data into the circular buffer, and the read pointer does the same as the reader is reading.

- In input sessions, the reader is the user program and the writer is the Data Acquisition device.
- In output sessions, the reader is the Data Acquisition device and the writer is the user program.

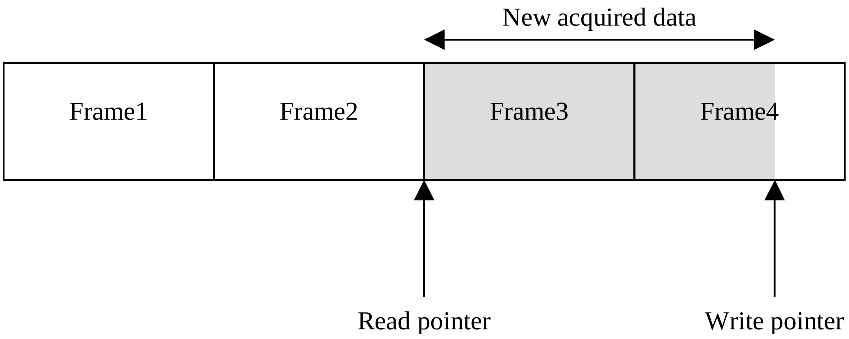


Figure 4 Diagram of read/write pointers in a typical circular buffer

Figure 4 shows a typical case where the reader is following the writer closely.

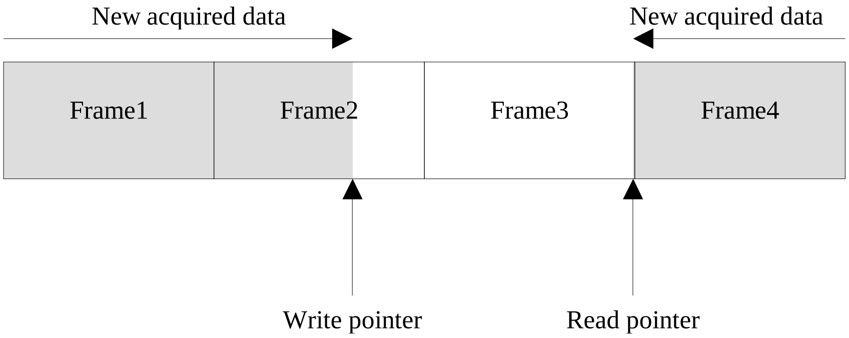


Figure 5 Diagram of read/write points when reads are lagging

Figure 5 shows a case where the reader is late and the writer is about to catch up with it.

This can happen if your application takes too long to process incoming buffers or doesn't send outgoing buffers fast enough.
A buffer over-run or under-run error will occur when the write pointer meets the reader pointer and the session will abort.

For input applications where losing data doesn't matter, you can tell the session to ignore the over-run error and the writer will over-write buffers whose data hasn't been processed yet with new data. The write pointer will just "pass" the read pointer without sending any error to the application.
The same thing is possible for output applications; the reader will recycle already generated data and send it again to the device.

You can disable the over/under run error notification by calling the DataStream object's method `"SetOverUnderRun"`.

The following sample shows how to set the number of frames and disable the buffer over-run error.

```
// Get a point to the data stream object
CueiDataStream* pDataStream = mySession.GetDataStream();

// Configure the number of frames
pDataStream->SetNumberOfFrames(8);

// Disable buffer over-run error
pDataStream->SetOverUnderRun(1);
```

By default, the reader reads the oldest data from the circular buffer and updates the read pointer accordingly. However, you can arbitrarily set the position of the read pointer by calling the DataStream object's methods `"SetRelativeTo"` and `"SetOffset"`.
This can be useful to skip unread data when you know your application is getting late or to always read the most recently acquired data and discard any "older" data.

You can set the `"RelativeTo"` property to `"CurrentPosition"` (its default value) or "`MostRecent`" which correspond to the position of the write pointer.
The "`Offset`" property specifies the new position of the read pointer relative to the "`RelativeTo`" property. Its value can be negative to move the read pointer backward or positive to move it forward.

For example to immediately read the most recently acquired 100 samples you would set "`RelativeTo`" to "`MostRecent`" and "`Offset`" to –100. This will move the read pointer 100 scans before the write pointer.

The following sample shows how to change the read pointer position.

```
// Get a point to the data stream object
CueiDataStream* pDataStream = mySession.GetDataStream();

// Configure the new RelativeTo property
pDataStream->SetRelativeTo(UeiDataStreamRelativeToMostRecentSample);

// Move read pointer 100 scans backward
pDataStream->SetOffset(-100);
```

You can access the number of scans that are available to be read from (or written to) the circular buffer using the DataStream object's method "`GetAvailableScans()`".
You can also get the total number of scans read from (or written to) the circular buffer using the DataStream object's method "`GetTotalScans()`".

3.3.3. Data map timing mode (legacy)

The "data map" timing mode (also called DMAP mode) is only available with PowerDNA devices. It allows for transfer of single scans at a given rate timed by a hardware clock and is the equivalent of legacy DMAP mode in the low-level programming environment.

This mode is very useful for real-time applications that need to acquire and process scans one by one but at a fixed rate.

The session object's method `ConfigureTimingForDataMappingIO()` is called to perform a hardware timed acquisition or generation.

```
// Configure the timing object to acquire or generate 1
// sample per channel at 100 Hz, using the device's internal clock
mySession.ConfigureTimingForDataMappingIO(
                    UeiTimingClockSourceInternal, 100.0);
```

This mode offers better performance than the "point by point" timing mode where each device is polled one by one in a software timed loop. In contrast, using the DMAP timing mode, all PowerDNA devices within one IO module are read simultaneously and the resulting data is transferred from the PowerDNA IO module to the host in one operation.

The acquired data is returned as an array containing one sample per channel.
The data to be generated must be passed as an array containing one value for each channel.

3.3.4. Edge detection timing mode

Edge detection timing mode configures the hardware to monitor the input lines of one or more digital port(s) and notify the user application when the specified event occurs. The edge detection mode is only available for digital input sessions.

The session object's method `ConfigureTimingForEdgeDetection()` is called to allow notification to the user application when an input line changes state.

The lines to monitor are set using a bit mask for each port configured in the session. The following sample code shows how to do this.

```
// Configure a digital input session with two input ports 0 and 1
mySession.CreateDIChannel("pwrdaq://Dev0/DI0,1");

// Configures the session to detect rising edges
mySession.ConfigureTimingForEdgeDetection(UeiDigitalEdgeRising);

// Select lines 2 and 3 for port 0 and line 0 for port 1
(CUeiDIChannel*)(mySession.GetChannel(0))->SetEdgeMask(0x6);
(CUeiDIChannel*)(mySession.GetChannel(1))->SetEdgeMask(0x1);
```

3.3.5. Messaging timing mode

The messaging timing mode is used with message communication devices such as serial ports CAN bus and ARINC-429 interfaces.

How a message is defined depends on the communication port type:

- On serial ports, messages are simply bytes.
- For CAN, ARINC, and MIL-1553 ports, see subsections below.

The session object's method `ConfigureTimingForMessagingIO()`is called to perform message communication provided that the device allows it.

```
// Configure the timing object to read and write messages
// from/to the communication ports.
// This session will get notified of new messages 10 times
// per second or as soon as 100 messages are received.
mySession.ConfigureTimingForMessagingIO(100, 10.0);
```

Once the session is configured for messaging IO, you can use reader and writer objects to simultaneously send and receive messages to and from the communication port.

You will need separate reader and writer objects for each communication port configured in the port list.

3.3.5.1. CAN bus data representation

On CAN ports, messages are CAN frames including frame arbitration identifier and payload data.

CAN frames are represented in C and C++ with the following data type:

```
typedef struct _tUeiCanFrame
{
   ///< CAN Frame arbitration id
   uInt32   Id;
   ///< Specifies whether this is a data, remote, or error frame
   tUeiCANFrameType   Type;
   ///< The number of significant bytes in the payload
   uInt32   DataSize;
   ///< The frame's payload. It can contain up to 8 bytes
   uInt8    Data[8];
} tUeiCANFrame;
```

3.3.5.2. ARINC 429 bus data representation

On ARINC-429 ports, messages are ARINC words including label, SDI, SSM bits and payload data.

ARINC words are represented in C/C++ with the following data type:

```
typedef struct _tUeiARINCWord
{
   /// The label of the word. It is used to determine the
   /// data type of the Data field, therefore, the method of
   /// data translation to use.
   uInt32   Label;
   /// Sign/Status Matrix or SSM. This field contains
   /// hardware equipment condition, operational mode,
   /// or validity of data content.
   uInt32   Ssm;
   /// Source/Destination Identifier or SDI.
   /// This is used for multiple receivers to identify
   /// the receiver for which the data is destined.
   uInt32   Sdi;
   /// The parity bit.
   uInt32   Parity;
   /// The payload of the word. Its format depends on the label.
   /// Most common formats are BCD (binary-coded-decimal) encoding,
   /// BNR (binary) encoding or discrete format where each
   /// bit represents a Pass/Fail, True/False or
   /// Activated/Non-Activated condition.
   uInt32   Data;
} tUeiARINCWord;
```

3.3.5.3. MIL-1553 bus data representation

On MIL-1553 ports, messages are represented in `tUeiMIL1553Frame` structure. Please notice that the same structure is used for different modes of operation – bus monitor, bus controller and remote terminal depends on the frame type.
MIL-1553 data is represented in C/C++ with the following data type:

```
typedef struct _tUeiMIL1553Frame
{
   ///< Specifies the type of 1553 frame
   tUeiMIL1553FrameType   Type;
   ///< Remote terminal frame belongs to
   uInt16   Rt;
   ///< Sub-address frame belongs to
   uInt16   Sa;
   ///< I/O block terminal frame belongs to
   uInt16   Block;
   ///< Command is required for BC functionality (for
UeiMIL1553FrameTypeTxFifo)
   tUeiMIL1553Command   Command;
   ///< The number of bytes in the payload (also size of data from TxFifo
frame)
   uInt32   DataSize;
   ///< The frame, bus monitor and TxFIFO frame payload. It can contain up to
36 uInt32s
   uInt32   RxTxData[36];
} tUeiMIL1553Frame;
```

The <*`tUeiMIL1553FrameType Type`* > frame type defines what kind of data is in the frame. Please remember that when you are passing allocated frames either to *`CUeiMIL1553Writer`* or to *`CUeiMIL1553Reader`* you must specify frame type.

Following types are defined:

- **`UeiMIL1553FrameTypeTxFifo`**: Data to transmit to output FIFO (used by *`CUeiMIL1553Writer`* only)
- **`UeiMIL1553FrameTypeBusMon`**: Data received from the bus monitor in bus monitor format FIFO (used by *`CUeiMIL1553Reader`* only)
- **`UeiMIL1553FrameTypeRtData`**: Data to/from remote terminal data area (used by both *`CUeiMIL1553Reader`* and *`CUeiMIL1553Writer`*)
- **`UeiMIL1553FrameTypeRtStatusData`**: Remote terminal status data (bit mask for subaddresses that processed Rx and Tx commands from the bus controller) (used by *`CUeiMIL1553Reader`* only)
- **`UeiMIL1553FrameTypeRtStatusLast`**: Bus status, last command and last mode command (data contains three elements in *`RxTxData`* array). Used by *`CUeiMIL1553Reader`* only)

- **`UeiMIL1553FrameTypeRtControlBlock`**: Used by *`CUeiMIL1553Writer`* only and allows to select active data block for remote terminals for both Tx and Rx data memory
- **`UeiMIL1553FrameTypeRtControlVector`**: Used by *`CUeiMIL1553Writer`* only for remote terminal control data - set vector, BIT words and bus for BC (data contains three elements in *`RxTxData`* array)**.**
- **`UeiMIL1553FrameTypeError`**: Frame contains error conditions encountered during operation

The `UeiMIL1553FrameTypeTxFifo` frame type is used to transmit data on the bus. Following fields need to be assigned before using it in *`CUeiMIL1553Writer`* *`.Write():`*
<Rt>: remote terminal to send command to
<Sa>: subaddress to send command to
<*`Command.Command`*> :one of the following command types:
UeiMIL1553CmdRx - Remote terminal to receive data from bus controller*.*
<*`Command.WordCount`*> specifies how many data words remote terminal should receive. <*`DataSize`*> contains number of data words in 16-bit format stored in <*`RxTxData[]`*>*.* Please notice that according to MIL-1553 command specification to send 32 data words <*`Command.WordCount`*> should be zero, while <*`DataSize`*> should be equal to 32.
UeiMIL1553CmdTx - Remote terminal to transmit data to bus controller.
<*`Command.WordCount`*> specifies how many data words remote terminal should transmit.
UeiMIL1553CmdRxTx- One remote terminal to transmit data to another remote terminal. <*`Command.WordCount`*> specifies how many data words remote terminal should transmit. Fields <*Rt2*> and <*Sa2*> must contain remote terminal and subaddress for the transmit command of Rt-Rt pair
UeiMIL1553CmdModeTx - Remote terminal to transmit data and/or status word to bus controller. See UeiMIL1553CmdTx for parameters
UeiMIL1553CmdModeRx - Remote terminal to receive data for mode command from bus controller. See UeiMIL1553CmdRx for parameters
UeiMIL1553CmdRxBroadcast - Remote terminal to receive broadcast data from bus controller
UeiMIL1553CmdRxTxBroadcast - One remote terminal to broadcast data to other remote terminal
UeiMIL1553CmdModeTxBroadcast - Mode command without data, remote terminals should not reply
UeiMIL1553CmdModeRxBroadcast - Mode command with data, remote terminals should receive data

Frame type UeiMIL1553FrameTypeBusMon is used to receive data from the bus monitor. Each command and status word on the bus is stored in the separate frame. The following fields represent data:

<*Rt*> - remote terminal in the command (unused if status)

<*Sa*> - subaddress in the command (unused if status)

<*Command.Command*> - one of the following commands (see description in *`tUeiMIL1553CommandType`*):

```
UeiMIL1553CmdTx
UeiMIL1553CmdRx
UeiMIL1553CmdModeTx
UeiMIL1553CmdModeRx
UeiMIL1553CmdModeTxBroadcast
UeiMIL1553CmdModeRxBroadcast
```

<*Command.WC*> - word count extracted in the command

UeiMIL1553FrameTypeBusMon frame always contain at least one element of data in <*`RxTxData[]`*> array. The size of data is stored in the <*`DataSize`*> field.

Bus monitor data is represented as follows:

First 32-bit word:

bit 31: parity error on the bus, if any
bit 30: set to 1 for command or status
bits 29 thru 16: time in 15.15ns interval since previous command or status.
bits 15 thru 0: command or status as received from the bus.

If there is a data following the command it represented in the following format:

bit 31: parity error on the bus, if any
bit 30: set to 0 for data word
bits 29 thru 16: time in 15.15ns interval since previous command or status.
bits 15 thru 0: data as received from the bus.

If timestamps are enabled using `CUeiMIL1553Port::EnableTimestamping()` method (timestamps are enabled by default) the last two words contain 32 bit “absolute” timestamp of the message in 10us resolution (timestamps are reset when the session starts) and the various flags defining current bus status and what bus (A or B) the command was received on. See “PowerDNA API Reference Manual” for further details.

3.4. Configuring the trigger(s)

The session automatically configures triggers by default to immediately start and stop operation on software request. You can override this and configure the session to start and/or stop when an external event occurs.

Digital, Analog, and Internal triggers are described in the subsections below.

Refer to Chapter 5 for more information about synchronizing triggers on multiple I/O devices and I/O modules.

3.4.1. Digital trigger

Digital triggers are used when you need to start or stop one device upon an external digital event. This is a per-device trigger; however, you can also use it to synchronize multiple devices by configuring each device to use its external trigger and connect the same external signal to every device.

The session object's method `ConfigureStartDigitalTrigger()` is called to start the session when an external digital event occurs.

```
// Configure the session to start operation when the rising
// edge of a TTL signal is applied on the external trigger input
// of the device.
mySession.ConfigureStartDigitalTrigger(UeiTriggerSourceExternal,
UeiDigitalEdgeRising);
```

Call the session object's method `ConfigureStopDigitalTrigger()` to stop the session when an external digital event occurs.

```
// Configure the session to stop operation when the falling
// edge of a TTL signal is applied on the external trigger input
// of the device.
mySession.ConfigureStopDigitalTrigger(UeiTriggerSourceExternal,
UeiDigitalEdgeFalling);
```

3.4.2. Analog software trigger

The analog software trigger looks at every sample acquired on the specified channel until the trigger condition is met. Once the trigger condition is met, the application can read the acquired scans in a buffer. The trigger level and hysteresis are specified in the same unit as the measurements.

Note that the acquisition starts immediately and runs in the background until the trigger condition is met or a timeout expires.

The session object's method `ConfigureAnalogSoftwareTrigger()` is called to start the session when the specified condition is met.

```
// Configure the session to only read data when the acquired signal
// meets the following condition: signal connected to the first
// channel in the channel list must rise over 1.0V within an
// hysteresis window of 0.5V. Keep 100 scans before the trigger
// condition
mySession.ConfigureAnalogSoftwareTrigger(UeiTriggerActionStartSession
, UeiTriggerConditionRising, 0, 1.0, 0.5, 100);
```

3.4.3. Internal signal trigger

Signal triggers are used to synchronize multiple devices connected to each other via a bus or backplane with synchronization capabilities.

- PDXI PowerDAQ boards can use any of the 8 PXI trigger lines as a trigger signal.
- PowerDNA layers can use any of the 4 Synchronization lines available on the PowerDNA bus as a trigger signal.
- PowerDNA IO modules can use their "`SyncIn/SyncOut`" external signal as a trigger signal.

The session object's method `ConfigureSignalTrigger()` is called to start the session when the specified signal event occurs.

```
// Configure the session to start operation when an event is received
// at the SyncIn connector.
mySession.ConfigureSignalTrigger(UeiTriggerActionStartSession,
“SyncIn”)
```

The trigger signal is specified using a string. Available trigger signals are:

- "`SyncIn`": event occurs when a rising edge is detected on the SynIn input pin.
- "`Sync0`" to "`Sync3`": event occurs when a rising edge is detected on one of the PowerDNA backplane synchronization lines.
- "`PXI0`" to"`PXI7`": event occurs when a rising edge is detected on one of the 8 PXI backplane trigger lines.

Refer to Chapter 5 for more information about synchronizing internal triggers.

3.5. Starting the session

You can start the session by calling the session object’s method `Start()`:

```
// Start the session
mySession.Start();
```

Note that if you don’t explicitly start the session, it will be automatically started the first time you try to transfer data.

3.6. Reading/Writing data from/to the device

The UeiDaq framework uses helper objects called readers and writers to transfer data to and from the device.

You can use various implementations of the reader/writer objects depending on the format of data you want to read/write.

You can even implement your own reader object by deriving one of the existing one to implement some post-processing.

Readers and writers for each type of session are listed below:

- Analog input sessions can use either of the following:
    - an “AnalogRawReader” object to retrieve raw binary codes (straight from the A/D converter)
    - an “AnalogScaledReader” object to retrieve data scaled to a physical unit (V, degrees, mV/V).
- Analog Output sessions can use “AnalogRawWriter” or “AnalogScaledWriter” objects.
- Digital input and output sessions use “DigitalReader” and “DigitalWriter” objects.
- Counter input and output sessions use “CounterReader” and “CounterWriter” objects.
- Variable reluctance sessions use “VRReader” objects. Each reader returns a structure (a cluster in LabVIEW) containing the measured velocity, position and total teeth count since the session started.
- Serial port sessions use “SerialReader” and “SerialWriter” objects.
- HDLC serial port sessions use “HDLCReader” and “HDLCWriter” objects.
- CSDB serial port sessions use “CSDBReader” and “CSDBWriter” objects; CSDB data is represented by the `tUeiCSDBMessage` structure, which contains an address byte, a status byte, payload datasize, and payload data bytes.

- SSI serial port sessions use “SSIReader” and “SSIWriter” objects; users can program how many bytes to receive or transmit and whether or not to enable gray encoding.
- Serial port sessions use “SerialReader” and “SerialWriter” objects.
- CAN bus sessions use “CANReader” and “CANWriter” objects.
- ARINC-429 input sessions use “ARINCReader” and “ARINCRawReader” objects.
- ARINC-429 output session use “ARINCWriter” and “ARINCRawReader” objects.
- MIL-1553 port sessions use “MIL1553Reader” and “MIL1553Writer” objects.
- IRIG channel sessions use “IRIGReader” objects.

All reader or writer objects are programmed in the same manner: you first create the object and then connect it to the session:

```
// Create the read object
UeiDaq::CUeiAnalogScaledReader reader;

// Connect it to the session
reader.SetDataStream(mySession.GetDataStream())
```

You can now read data through the reader object:

```
// Read 1000 scans in the specified buffer
reader.ReadMultipleScans(1000, buffer);
```

The default behavior of reader and writer objects is to block until the specified number of scans is ready to be transferred. You can also configure those objects to work asynchronously. The method used to program readers and writers asynchronously is highly dependent on the programming language; you can find more information on how to do this in the Reference manual for each development environment.

Note that when the session is configured for messaging IO, each channel configured in the channel list must be accessed separately. Therefore you need to create reader and writer objects dedicated to each configured communication port.

3.7. Stopping the session

You can stop the session by calling the session object's `Stop()` method:

```
// Stop the session
mySession.Stop();
```

Note that if you don't explicitly stop the session, it will be automatically stopped when the session object is destroyed or goes out of scope.

3.8. Destroying the session

In C++, if you created the session object on stack, it will automatically free its resources when it goes out of scope. Alternatively, you can force it to free its resources by calling the `CleanUp()` method:

```
// Clean-up session
mySession.CleanUp();
```

If you dynamically created the session object, you need to destroy it to free all resources:

```
// Destroy session
delete(pMySession);
```

In C, you need to call `UeiDaqCloseSession()` to free all resources:

```
UeiDaqCloseSession(mySession);
```

With .NET managed languages, the garbage collector will take care of freeing resources once the session object is not referenced anymore. You can also force the session to release its resources by calling the "Dispose" method.

4. Programming the UeiDaq framework

4.1. UeiDaq framework C++ API

The UeiDaq framework ships with header files and import libraries for Visual Studio 6.0, Visual Studio.NET 2003, Visual Studio.NET 2005, Visual Studio.NET 2008 and Visual Studio.NET 2010.

Header files are located in the include directory:
<Program Files>\UEI\Framework\CPP\include
The import library is located in the lib directory:
<Program Files>\UEI\Framework\CPP\lib
Examples of MFC and console applications are located in the examples directory:
<Program Files>\UEI\Framework\CPP\examples

The UeiDaq framework installer creates an environment variable “`UEIDAQROOT`”, which is set to the root directory of the framework.

You only need to include one header file, “UeiDaq.h”, in your program to get access to the UeiDaq framework classes. It is recommended to include “UeiDaq.h” last, after any C and C++ standard headers you include. To help the compiler find this file, add the directory `$(UEIDAQROOT)\CPP\include` to the list of additional include directories in your project settings.

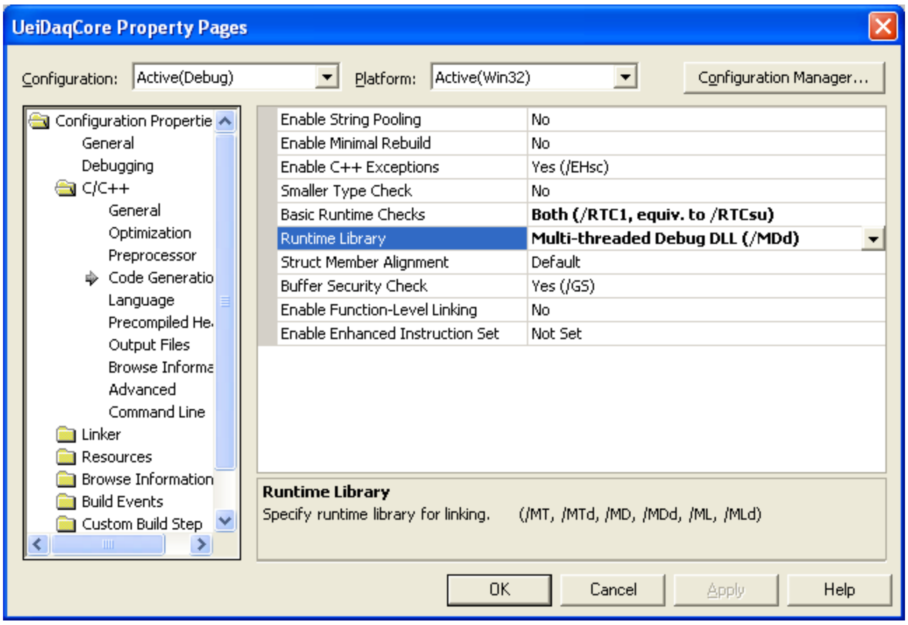
The header file contains directives to automatically link with the import library matching your compiler version so you only need to modify your linker settings to add the UeiDaq framework lib directory path.

You can link your program with the UeiDaq framework libraries statically or dynamically (see below).

All the UeiDaq framework classes are defined within their own “UeiDaq” namespace. You either need to insert the “`using namespace UeiDaq;`” directive in your program or use the prefix “`UeiDaq::`” when you reference the UeiDaq framework classes and types.

4.1.1. Shared library

When creating a new project, make sure that the **Runtime Library** setting in your project's "Code Generation" property page is set to "Multi-threaded DLL" for the **Release** configuration and "Multi-threaded Debug DLL" for the **Debug** configuration.



Configuration: Active(Debug)
Platform: Active(Win32)

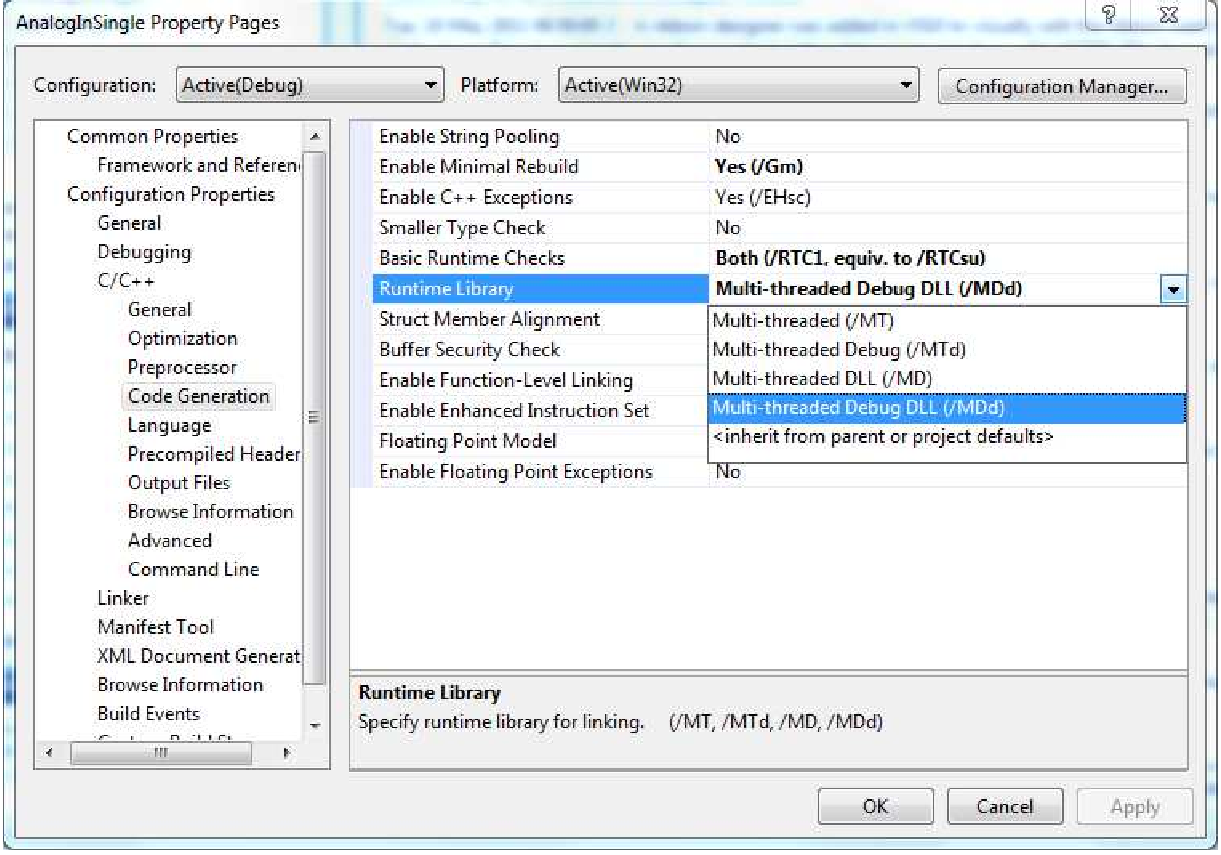
Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	**Both (/RTC1, equiv. to /RTCsu)**
Runtime Library	**Multi-threaded Debug DLL (/MDd)**
Struct Member Alignment	Default
Buffer Security Check	Yes (/GS)
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set

Runtime Library
Specify runtime library for linking. (/MT, /MTd, /MD, /MDd, /ML, /MLd)

4.1.2. Static library

The static libraries in the UeiDaq framework are built to use the static version of Microsoft’s C++ runtime.
Using the static library requires a few extra steps detailed below:

- Make sure **Runtime Library** setting of your project is set to Multi-threaded for **Release** configuration or Multi-threaded debug for **Debug** configuration



AnalogInSingle Property Pages

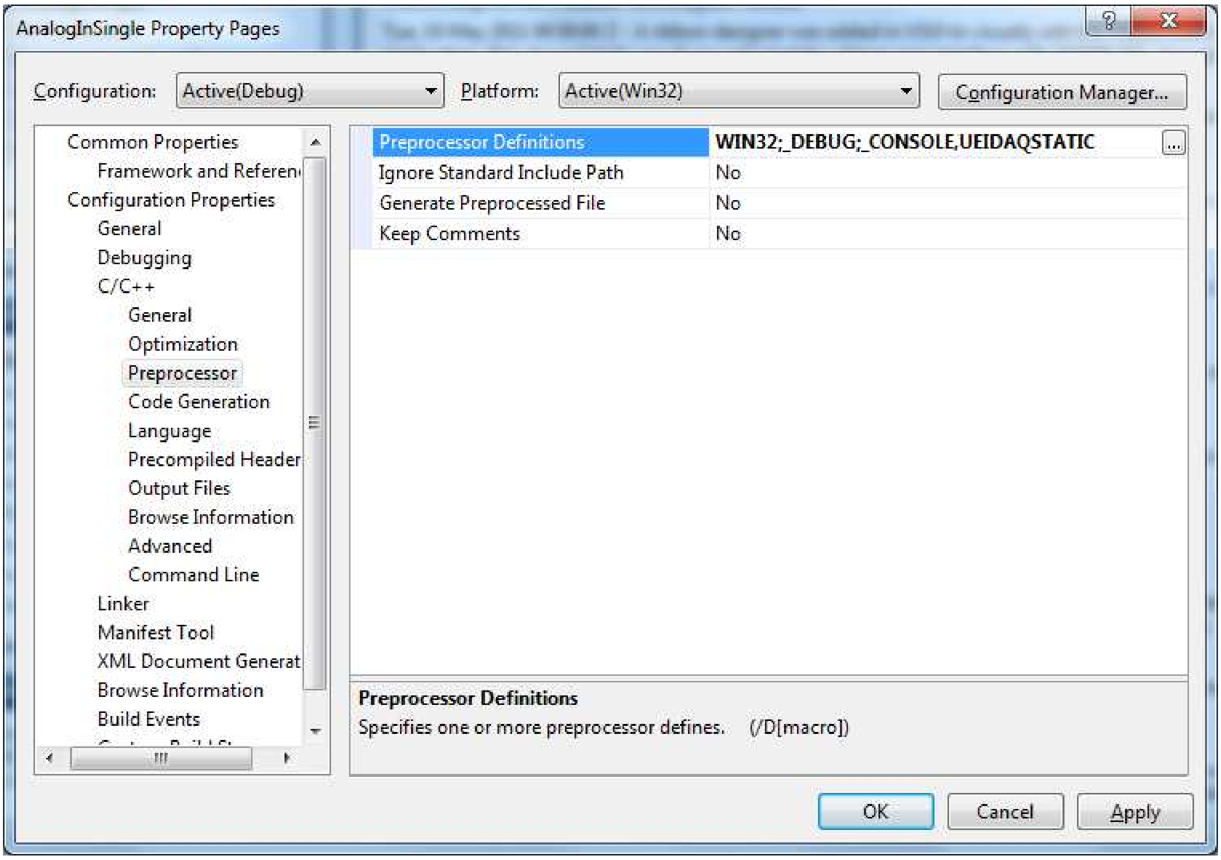
Configuration: Active(Debug)
Platform: Active(Win32)

Enable String Pooling	No
Enable Minimal Rebuild	**Yes (/Gm)**
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	**Both (/RTC1, equiv. to /RTCsu)**
Runtime Library	**Multi-threaded Debug DLL (/MDd)**
Struct Member Alignment	
Buffer Security Check	
Enable Function-Level Linking	
Enable Enhanced Instruction Set	
Floating Point Model	
Enable Floating Point Exceptions	No

Multi-threaded (/MT)
Multi-threaded Debug (/MTd)
Multi-threaded DLL (/MD)
Multi-threaded Debug DLL (/MDd)
<inherit from parent or project defaults>

Runtime Library
Specify runtime library for linking. (/MT, /MTd, /MD, /MDd)

- Add UEIDAQSTATIC to the preprocessor definitions.



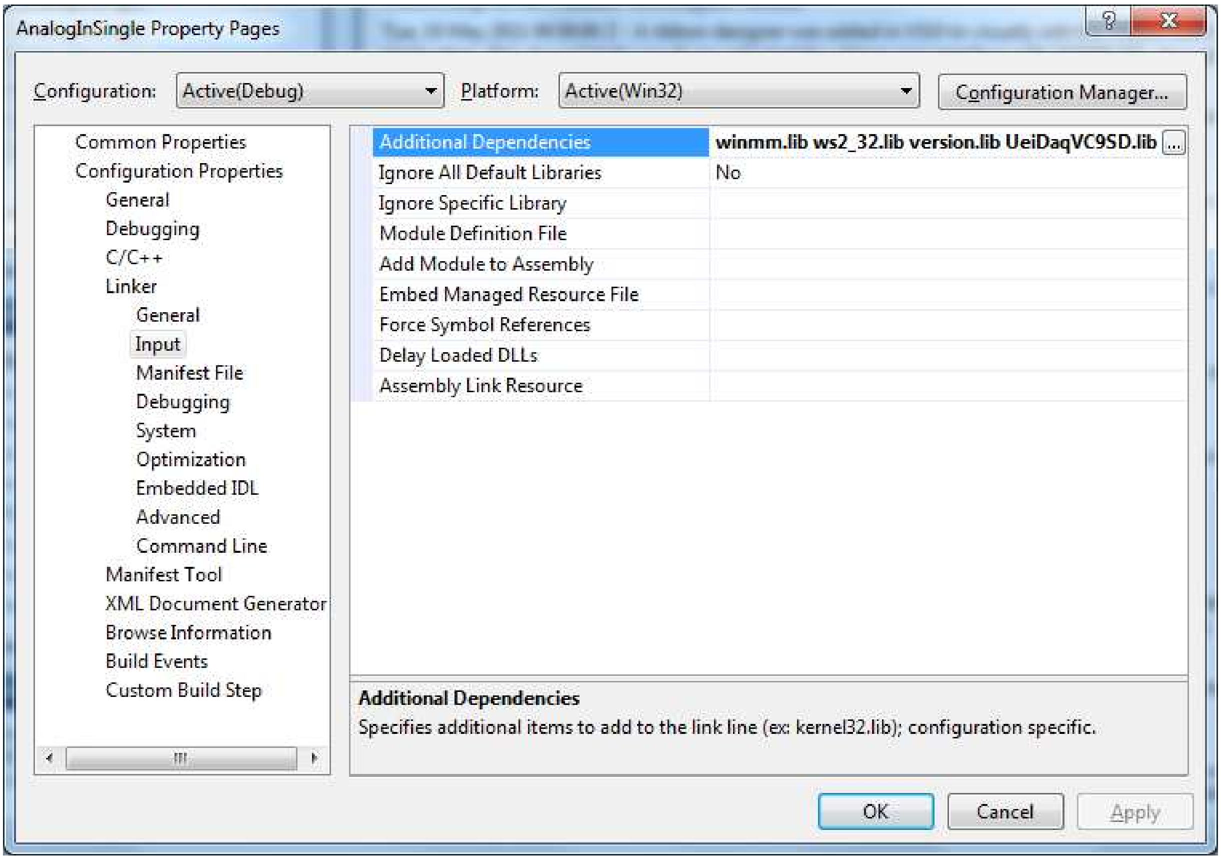
Configuration: Active(Debug)
Platform: Active(Win32)

Preprocessor Definitions	WIN32;_DEBUG;_CONSOLE,UEIDAQSTATIC
Ignore Standard Include Path	No
Generate Preprocessed File	No
Keep Comments	No

Preprocessor Definitions
Specifies one or more preprocessor defines. (/D[macro])

- Add the following libraries to **Additional dependencies**: winmm.lib, version.lib, ws2_32.lib,ueidaqvc*SD.lib

Visual Studio 2005: use UeiDaqVC8S.lib for release and UeiDaqVC8SD.lib for debug
Visual Studio 2008: use UeiDaqVC9S.lib for release and UeiDaqVC9SD.lib for debug
Visual Studio 2010: use UeiDaqVC10S.lib for release and UeiDaqVC10SD.lib for debug



Configuration: Active(Debug)
Platform: Active(Win32)

Additional Dependencies	winmm.lib ws2_32.lib version.lib UeiDaqVC9SD.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

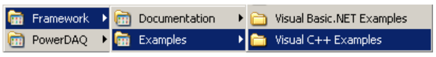
Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

- Finally, add a call the API `UeiDaqInitLib()` at the beginning of your program (before calling any other UeiDaq API).

```
UeiDaqInitLib();
```

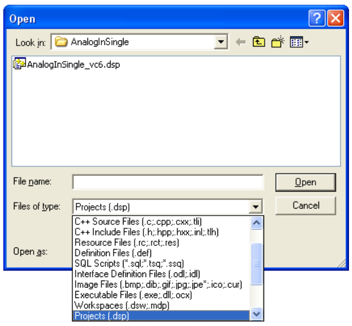
4.1.3. Step-by-step tutorial – Ms Visual C++ 6.0

The fastest path to creating new working code is to use existing example code as a base. Open up the existing examples from the Start Menu:



All examples in the example directory are "ready out of the box," and will compile in the example directory. They already contain the relevant directory references to the UeiDaq framework.

We will want to work outside the example directory. Copy one example to a project directory or desktop to work on it. Start Visual C++ 6.0 and open the .dsp project file in the copied directory.


Open
Open
Cancel
Open as:
C++ Source Files (.c;.cpp;.cxx;.tli)
C++ Include Files (.h;.hpp;.hxx;.inl;.tlh)
Resource Files (.rc;.rct;.res)
Definition Files (.def)
SQL Scripts (*.sql;*.tsq;*.ssq)
Interface Definition Files (.odl;.idl)
Image Files (.bmp;.dib;.gif;.jpg;.jpe*;.ico;.cur)
Executable Files (.exe;.dll;.ocx)
Workspaces (.dsw;.mdp)
Projects (.dsp)

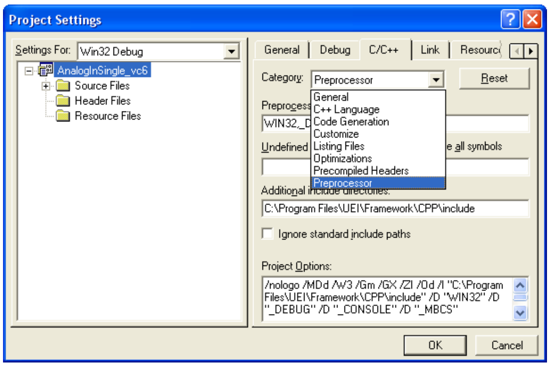
Look in: AnalogInSingle

AnalogInSingle_vc6.dsp

File name:

Files of type: Projects (.dsp)

The header and library files will need to be re-referenced under *Project > Settings*.
For the include files: *C/C++ > Preprocessor > Additional include directories:*
For the linker files: *Link > Input >Additional library path:*



Settings For: Win32 Debug

Category: Preprocessor

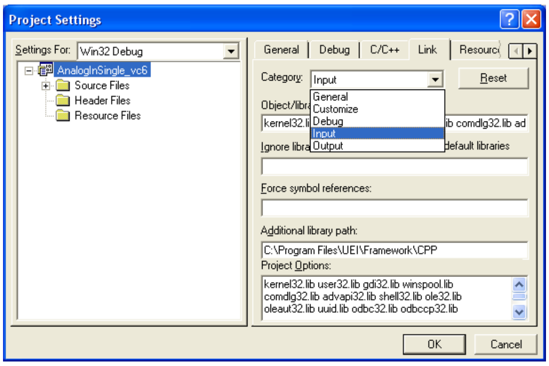
WIN32,_D

Additional include directories: C:\Program Files\UEI\Framework\CPP\include

☐ Ignore standard include paths

Project Options:

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /I "C:\Program
Files\UEI\Framework\CPP\include" /D "WIN32" /D
"_DEBUG" /D "_CONSOLE" /D "_MBCS"
```



Settings For: Win32 Debug

Category: Input

kernel32.li ib comdlg32.lib ad

Force symbol references:

Additional library path: C:\Program Files\UEI\Framework\CPP

Project Options:

```
kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib ole32.lib
oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

You may choose to make the header and library files available to all projects by going to: *Tools > Options > Directories > Show Directories for > Include files* or *Library files*.

Compile the project to test that it works. If you receive a compiler error message stating that you are missing files, check the include directory; if you are missing .lib files, check the library directory. Otherwise, you are now ready to enhance or rewrite the existing code to suit your needs.

4.2. UeiDaq framework C API

In addition to the C++ class library, the UeiDaq framework headers and import library also export a C API. This API can be programmed from Visual Studio, Borland C++, LabWindows/CVI and any other environment that can call external C code.

Header files are located in the include directory:
<Program Files>\UEI\Framework\CPP\include
The import library is located in the lib directory:
<Program Files>\UEI\Framework\CPP\lib
Examples of console application are located in the examples directory:
<Program Files>\UEI\Framework\CPP\examples

You only need to include one header file, “UeiDaq.h”, in your program to get access to the UeiDaq framework classes. It is recommended to include “UeiDaq.h” last, after any C and C++ standard headers you include. To help the compiler find this file, add the directory `$(UEIDAQROOT)\CPP\include` to the list of additional include directories in your project settings.

The header file contains directives to automatically link with the import library matching your compiler version so you only need to modify your linker settings to add the UeiDaq framework lib directory path.

4.3. UeiDaq framework .Net API

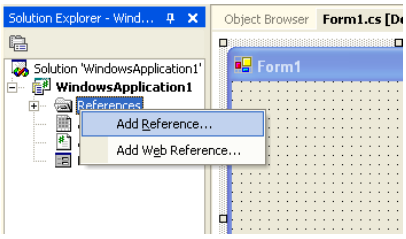
The UeiDaq framework implements an assembly for the .NET 2.0 framework that can be programmed from Visual Studio.NET 2005/2008/2010/2012/2013/2015 or any development environment that can interface with .NET assemblies.
The Ueidaq .Net framework API is now the recommended interface to program UEI hardware with Matlab.

The UeiDaq framework .NET assembly is called UeiDaqDNet.dll.

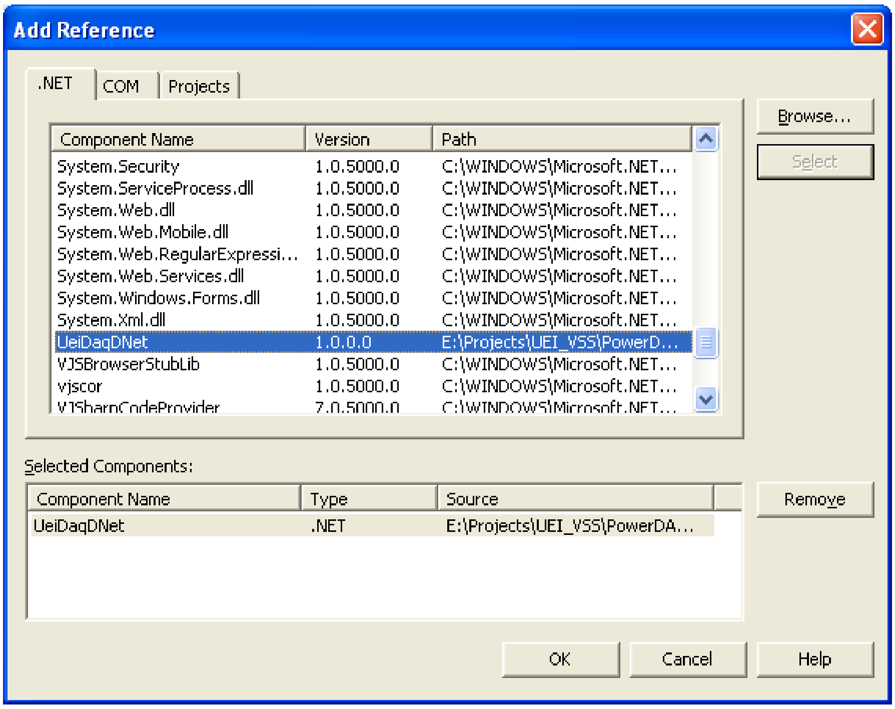
The 32-bit version is located in the following directory:
<Program Files>\UEI\Framework\DotNet
The 64-bit version is located in the x64 directory:
<Program Files>\UEI\Framework\DotNet\x64
Examples for VB.NET and C# are located in the examples directory:
<Program Files>\UEI\Framework\DotNet\examples

4.3.1. Using the UeiDaq .Net assembly in Visual studio

In order to use the UeiDaq .Net assembly, you first need to reference it in your project. Right-click on the references item in your project's view and select "Add References…"



The "Add Reference" dialog box will pop-up:

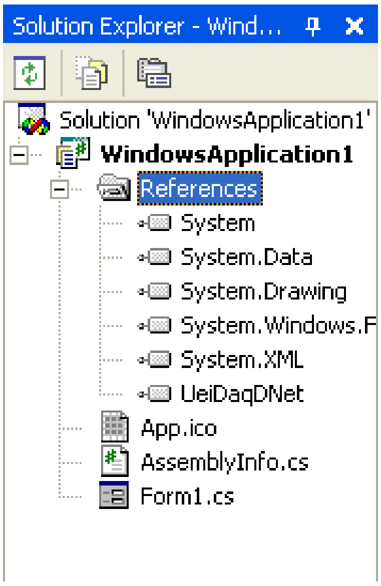


Component Name	Version	Path
System.Security	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
System.ServiceProcess.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
System.Web.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
System.Web.Mobile.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
System.Web.RegularExpressi…	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
System.Web.Services.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
System.Windows.Forms.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
System.Xml.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
UeiDaqDNet	1.0.0.0	E:\Projects\UEI_VSS\PowerD…
VJSBrowserStubLib	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
vjscor	1.0.5000.0	C:\WINDOWS\Microsoft.NET…
VJSharpCodeProvider	7.0.5000.0	C:\WINDOWS\Microsoft.NET…

Component Name	Type	Source
UeiDaqDNet	.NET	E:\Projects\UEI_VSS\PowerDA…

Scroll down and click the UeiDaqDNet assembly, click "Select" and then click "OK".

The UeiDaqDNet assembly should now be in your project references:



You are now ready to start programming with the UeiDaq framework!

All of the UeiDaq framework classes are defined within their own namespace “UeiDaq”. You either need to insert the “`using UeiDaq;`” directive in your program or use the prefix “`UeiDaq.`” when you reference the UeiDaq framework classes and types.

4.3.2. Using the UeiDaq .Net assembly in Matlab

4.3.2.1. Loading the assembly

The UeiDaq assembly path depends on the bitness of the operating system.

The code below queries the registry to detect where the assembly is located:

```
% Get location of UEIDAQ .NET assembly from registry
ueidaqPath = '';
try
    % this will only work on 64-bit PC
    ueidaqPath = winqueryreg('HKEY_LOCAL_MACHINE','Software\Wow6432Node\UEI\OOP',
'InstallDir');
    ueidaqPath = [ueidaqPath '\DotNet\x64\UeiDaqDNet.dll'];
catch e
    try
        % if not, maybe it is a 32-bit PC
        ueidaqPath = winqueryreg('HKEY_LOCAL_MACHINE','Software\UEI\OOP', 'InstallDir');
        ueidaqPath = [ueidaqPath '\DotNet\UeiDaqDNet.dll'];
    catch e
        % no registry, ueidaq software is not installed
        error('UeiDaq software is not installed');
    end
end
```

Loading the assembly and importing ueidaq namespace:

```
try
    NET.addAssembly(ueidaqPath);
    import UeiDaq.*;
```

Setting-up a session and reading/writing some data is done similarly to other object-oriented languages:

```
    % Create and configure UeiDaq framework session
    aiss = UeiDaq.Session();
    aiss.CreateAIChannel('simu://Dev0/Ai0:3', -10.0, 10.0,
                         UeiDaq.AIChannelInputMode.Differential);
    aiss.ConfigureTimingForBufferedIO(numScans, UeiDaq.TimingClockSource.Internal,
                                      scanRate, UeiDaq.DigitalEdge.Rising,
                                      UeiDaq.TimingDuration.Continuous);

    % Create a reader object to read data synchronously.
    reader = UeiDaq.AnalogScaledReader(aiss.GetDataStream());

    % Start session and read first buffer
    aiss.Start();

    netData = reader.ReadMultipleScans(numScans);
```

The data is returned as a .NET array. It needs to be converted to a matlab vector before doing anything useful with it:

```
     % Convert .NET array to matlab array and plot
    mlData = double(netData);
    plot(mlData);
```

Finally, close the session. The API handles errors by throwing exceptions. The exception handler catches them and display the error message.

```
aiss.Dispose();
catch e
  error(e.message);
end
```

4.4. UeiDaq framework ActiveX interface

The UeiDaq framework ships with a COM server that implements a set of COM interfaces. They can be programmed from Visual Basic 6, Borland Delphi or any development environment that can interface with COM/ActiveX interfaces.

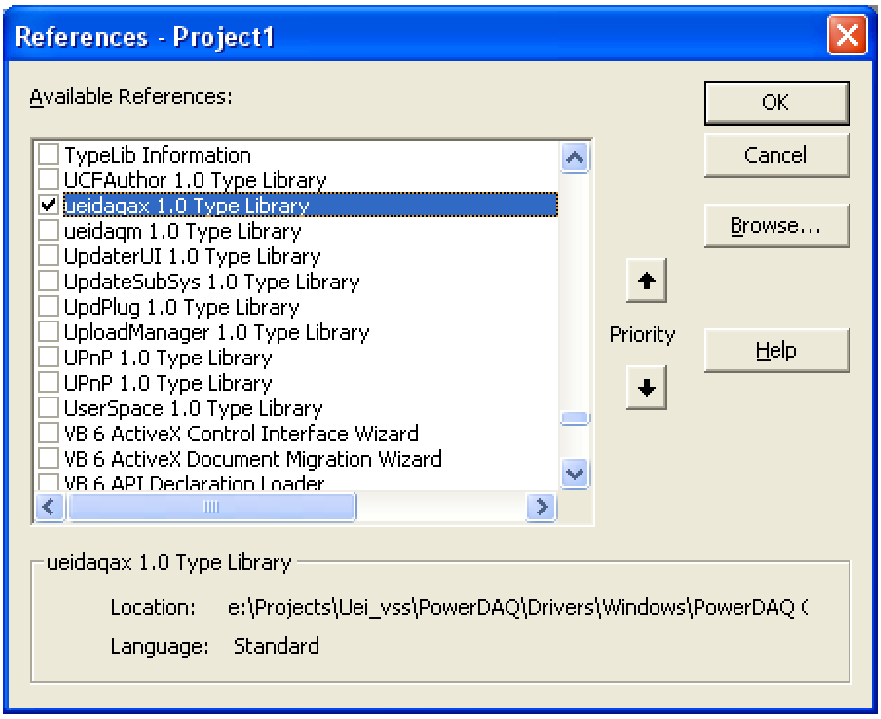
The UeiDaq framework .COM server is called UeiDaqAx.dll and is located in the following directory:

<Program Files>\UEI\Framework\ActiveX

Examples for VB 6 are located in the examples directory:

<Program Files>\UEI\Framework\ActiveX\examples\VB6

To use the UeiDaq ActiveX interface from VB6 you first need to reference it in your project. Select the “references…” item in the “Project” menu; the following dialog box will pop-up:



References - Project1

Available References:

- ☐ TypeLib Information
- ☐ UCFAuthor 1.0 Type Library
- ☑ ueidaqax 1.0 Type Library
- ☐ ueidaqm 1.0 Type Library
- ☐ UpdaterUI 1.0 Type Library
- ☐ UpdateSubSys 1.0 Type Library
- ☐ UpdPlug 1.0 Type Library
- ☐ UploadManager 1.0 Type Library
- ☐ UPnP 1.0 Type Library
- ☐ UPnP 1.0 Type Library
- ☐ UserSpace 1.0 Type Library
- ☐ VB 6 ActiveX Control Interface Wizard
- ☐ VB 6 ActiveX Document Migration Wizard
- ☐ VB 6 API Declaration Loader

ueidaqax 1.0 Type Library

Location: e:\Projects\Uei_vss\PowerDAQ\Drivers\Windows\PowerDAQ (

Language: Standard

Scroll down and select the ueidaqax type library. You can now start programming the UeiDaq framework from VB6.

4.4.1. Step by step tutorial - Ms Visual Basic 6.0

Create a new project: Standard EXE.

Add the UeiDaq Framework ActiveX control: *Project* > *References...* > *ueidaqax.dll*

Before starting with an example, familiarize yourself with the framework by using the *View* menu’s *Object Browser*, and selecting **ueidaqaxLib** instead of **<All Libraries>**. Many of the components of the UeiDaqAxLib you have already seen in Section 2.3. Section 3 (above) outlines how to use those objects to create meaningful applications.

The following is an example of how to read and display data in using the simulator device. The example creates a session, configures it for analog input, and then configures the timing, no triggers.

```
Dim session As UeiSession
Set session = New UeiSession            ' 3.1 Create session

' Configure session – sim, device 0, -10 to 10 V, analog in ch. 0
session.CreateAIChannel “simu://Dev0/Ai0”, -10#, 10#,
UeiAIChannelInputModeDifferential

session.ConfigureTimingForSimpleIO'     ‘ 3.3 Configure timing
```

The device is configured to look for data on the analog line. When the data acquisition process begins, gather it with a reader object and print it to a message box:

```
' Create reader
Dim reader As UeiAnalogScaledReader
Set reader = New UeiAnalogScaledReader
' Configure Reader
reader.SetDataStream session.GetDataStream

' Create an array with one element to store the scaled voltage from ch 0
Dim acquired_data() As Double

session.Start                           ' 3.5 Start acquiring data
acquired_data = reader.ReadSingleScan   ' 3.6 Read data into the array
session.Stop                            ' 3.7 Stop acquiring data

Msgbox aquired_data(0) & “ Volts”       ' Print acquired data to screen
Unload Form1                            ' and close the window on OK.
```

To perform the test on real hardware, change the `simu` device to a `pwrdaq` or `pdna`. This is a most trivial example of how to read one single value; the Visual Basic 6 Examples packaged with the framework contain more elaborate samples; with ten more lines, the examples show how to add a simple Ms-Graph.

4.4.2. Step by Step tutorial - Borland Delphi

The fastest path to creating new working code is to use existing example code as a base. Existing examples can be opened from the Start Menu:

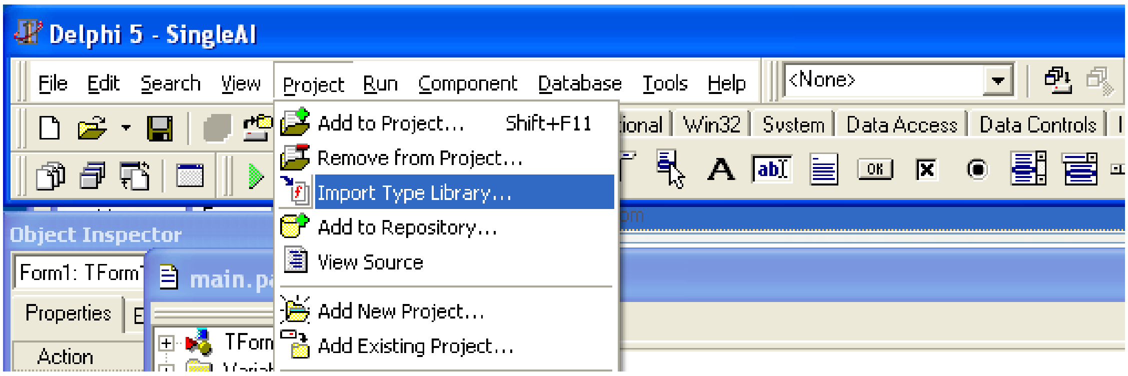
Programs/UEI/Framework/Examples/Borland Delphi examples

The support for programming the UeiDaq framework with Delphi is done through our ActiveX interface.

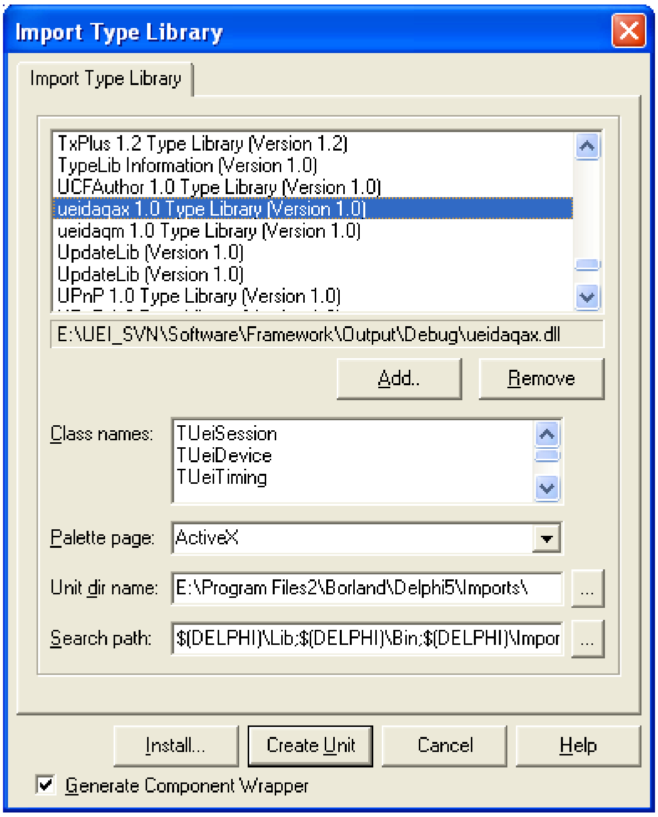
You first need to import the ueidaq activex library before running the sample program or you will get an error message “File not found: ‘ueidaqaxLib_TLb.pas’”.

The file ueidaqaxLib_TLB.pas is automatically generated by Delphi during the import process.

Under Delphi select the following menu: “Project/Import Type Library”.



In the dialog box, select the item “ueidaqax 1.0 Type Library” and click the “Create Unit” button.



TxPlus 1.2 Type Library (Version 1.2)
TypeLib Information (Version 1.0)
UCFAuthor 1.0 Type Library (Version 1.0)
ueidaqax 1.0 Type Library (Version 1.0)
ueidaqm 1.0 Type Library (Version 1.0)
UpdateLib (Version 1.0)
UpdateLib (Version 1.0)
UPnP 1.0 Type Library (Version 1.0)

E:\UEI_SVN\Software\Framework\Output\Debug\ueidaqax.dll

Class names: TUeiSession
TUeiDevice
TUeiTiming

Palette page: ActiveX

Unit dir name: E:\Program Files2\Borland\Delphi5\Imports\

Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Impor

☑ Generate Component Wrapper

Version 5 of Delphi has a bug when importing type libraries containing asynchronous event interfaces. If you get the error “*’}’ unexpected but identifier ‘OleVariant’ found*”, you need to modify the file ueidaqaxLib_TLb.pas. For each line where the error occurs remove the part between brackets.
For example:

```
FOnDataReady16(Self, Params[0] {out {??PSafeArray} OleVariant});
```

becomes:

```
FOnDataReady16(Self, Params[0]);
```

You can now run any of the examples.

4.4.3. Step by Step tutorial - Borland C++ Builder

The fastest path to creating new working code is to use existing example code as a base. Existing examples can be opened from the Start Menu:

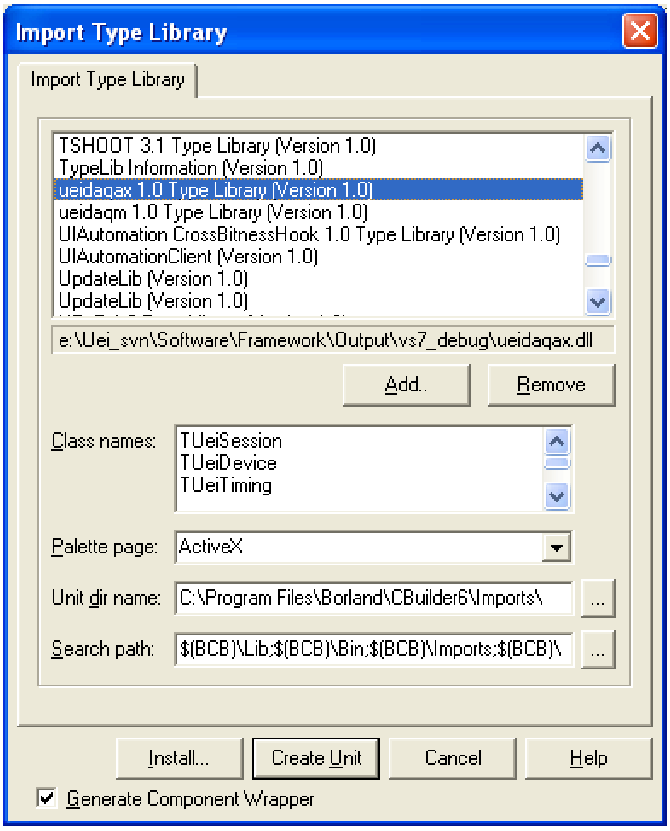
Programs/UEI/Framework/Examples/Borland C++ Builder examples

The support for programming the UeiDaq framework with C++ builder is done through our ActiveX interface.

You first need to import the ueidaq activex library before running the sample program or you will get an error message "File not found: 'ueidaqaxLib_TLb.cpp'".
The files ueidaqaxLib_TLB.cpp and ueidaqax_OCX.cpp are automatically generated by C++ builder during the import process.

Under C++ builder select the following menu: "Project/Import Type Library".

In the dialog box, select the item "ueidaqax 1.0 Type Library" and click the "Create Unit" button.



TSHOOT 3.1 Type Library (Version 1.0)
TypeLib Information (Version 1.0)
ueidaqax 1.0 Type Library (Version 1.0)
ueidaqm 1.0 Type Library (Version 1.0)
UIAutomation CrossBitnessHook 1.0 Type Library (Version 1.0)
UIAutomationClient (Version 1.0)
UpdateLib (Version 1.0)
UpdateLib (Version 1.0)

e:\Uei_svn\Software\Framework\Output\vs7_debug\ueidaqax.dll

Class names: TUeiSession
TUeiDevice
TUeiTiming

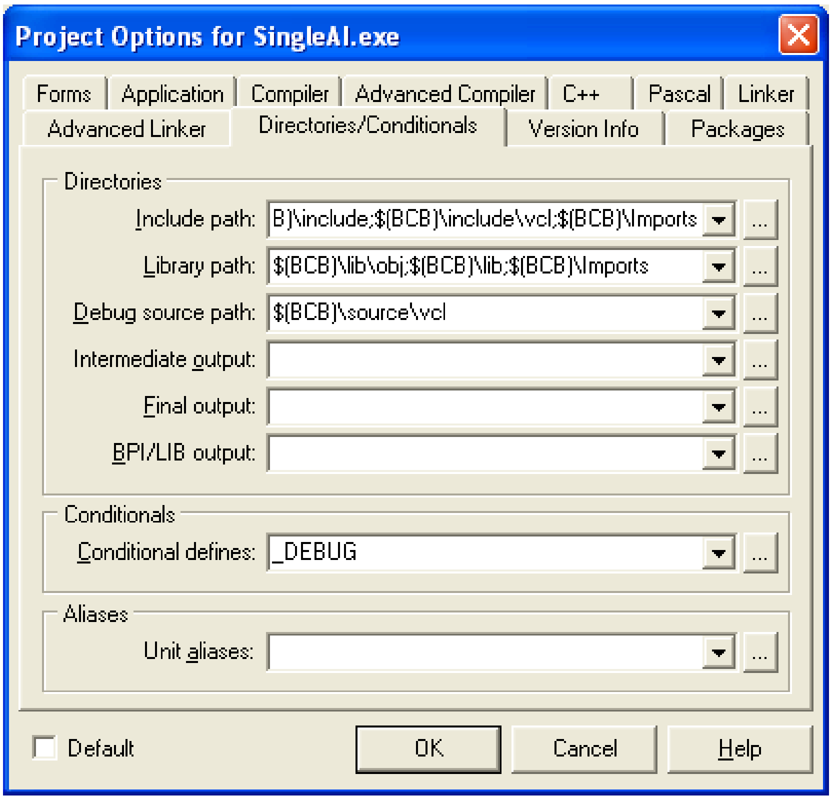
Palette page: ActiveX

Unit dir name: C:\Program Files\Borland\CBuilder6\Imports\

Search path: $(BCB)\Lib;$(BCB)\Bin;$(BCB)\Imports;$(BCB)\

☑ Generate Component Wrapper

Open the project options. In the "Directories/Conditionals" make sure that $(BCB)\Imports is listed in "Include Path" and "Library Path"



Include path: B)\include;$(BCB)\include\vcl;$(BCB)\Imports

Library path: $(BCB)\lib\obj;$(BCB)\lib;$(BCB)\Imports

Debug source path: $(BCB)\source\vcl

Intermediate output:

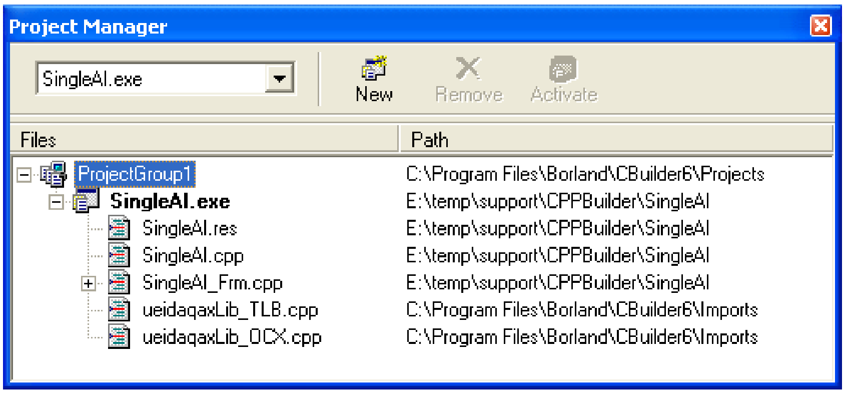
Final output:

BPI/LIB output:

Conditional defines: _DEBUG

Unit aliases:

Open the project manager and add the following files to the project:
$(BCB)\Imports\ueidaqax_OCX.cpp, $(BCB)\Imports\ueidaqax_TLB.cpp



SingleAI.exe

Files	Path
ProjectGroup1	C:\Program Files\Borland\CBuilder6\Projects
SingleAI.exe	E:\temp\support\CPPBuilder\SingleAI
SingleAI.res	E:\temp\support\CPPBuilder\SingleAI
SingleAI.cpp	E:\temp\support\CPPBuilder\SingleAI
SingleAI_Frm.cpp	E:\temp\support\CPPBuilder\SingleAI
ueidaqaxLib_TLB.cpp	C:\Program Files\Borland\CBuilder6\Imports
ueidaqaxLib_OCX.cpp	C:\Program Files\Borland\CBuilder6\Imports

Version 6 of C++ builder has a bug when importing type libraries containing asynchronous event interfaces. If you get the error: “*[C++ Error] ueidaqaxLib_OCX.cpp(187): E2034 Cannot convert 'TVariant' to 'tagSAFEARRAY * *'*”,

You need to edit the file $(BCB)\Imports\ueidaqaxLib_OCX.cpp and comment out the content of each *InvokeEvent* method. For example:

```
void __fastcall TUeiAnalogRawReader::InvokeEvent(int id,
Oleserver::TVariantArray& params)
{
  /*switch(id)
  {
    case 1: {
      if (OnDataReady16) {
        (OnDataReady16)(this, TVariant(params[0]));
      }
      break;
      }
    case 2: {
      if (OnDataReady32) {
        (OnDataReady32)(this, TVariant(params[0]));
      }
      break;
      }
    case 3: {
      if (OnError) {
        (OnError)(this, TVariant(params[0]), TVariant(params[1]));
      }
      break;
      }
    default:
      break;
  }*/
}
```

4.5. UeiDaq Framework Java API

The UeiDaq framework implements a java API that gives you access to the UeiDaq class library from the Java programming language.

The UeiDaq framework Java classes are contained in a Jar file named ueidaq.jar, which is located in the following directory:
<Program Files>\UEI\Framework\Java
The java examples are located in the examples directory:
<Program Files>\UEI\Framework\Java\examples

The UeiDaq framework classes for Java matches the C++ classes one to one. Please refer to the C++ reference manual to find information about each class.

In order to use the UeiDaq framework Java classes in your program, you need to import the package “com.ueidaq.framework.*”, you also need to load the UeiDaq JNI wrapper at the beginning of your program:

```
import com.ueidaq.framework.*;

public class DeviceInfo
{
   static
   {
      // The UeiDaq framework class library is implemented using JNI
      // Load the JNI wrapper DLL that does the interface between
      // Java and the framework
      System.loadLibrary("UeiDaqJava");
   }

   public static void main(String argv[])
   {
      // Create a Session
      CueiSession mySession = new CueiSession();
      MySession.CreateAIChannel(…);
      …
   }
```

You can then build your program from the command line with the following:

```
javac –cp ueidaq.jar MyProgram.java
```

If you use an IDE such as Eclipse or Jbuilder, make sure that “ueidaq.jar” is present in the list of libraries to use when building your project.

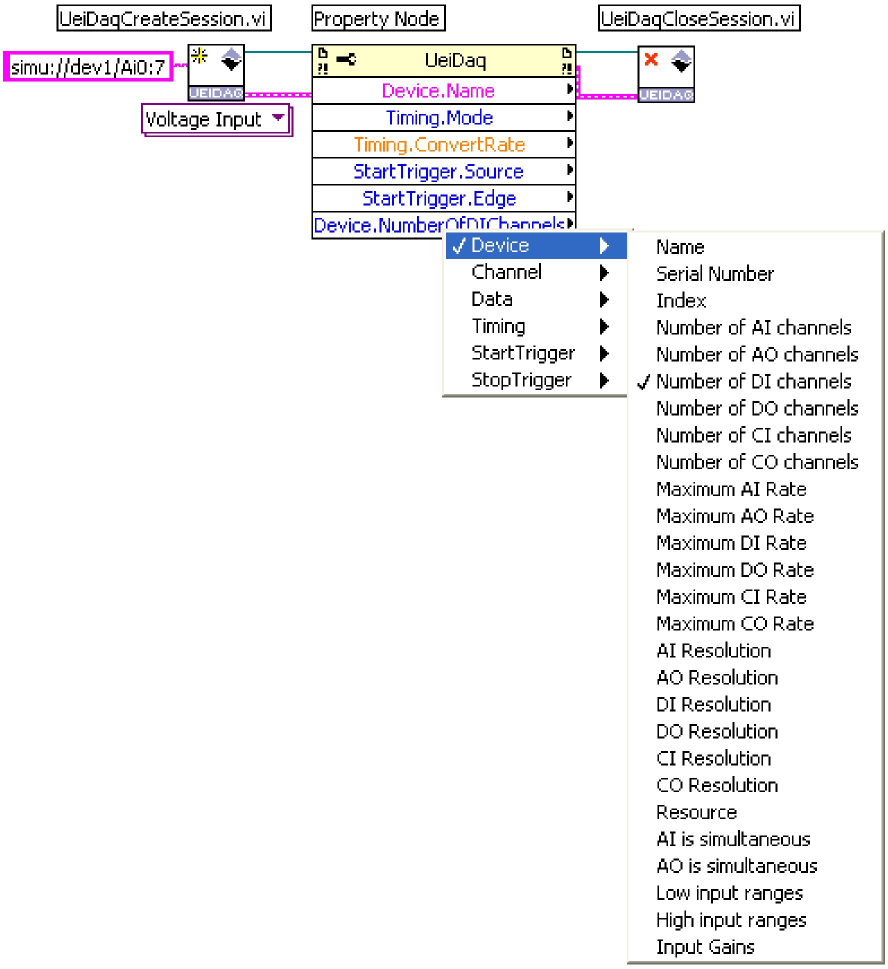
You can run your test program with the following command:

```
Java –cp ueidaq.jar MyProgram
```

4.6. *UeiDaq framework LabVIEW library*

The framework concept and class hierarchy explained in chapter 2 are still valid in LabVIEW. The session object is represented by a session refnum that is used by the UeiDaq VIs to reference a given session.

The session child objects, such as channels, device, timing and triggers, don't have a dedicated refnum but their properties can be accessed through a property node:



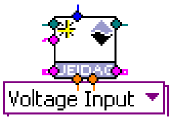
Please refer to the manual **UeiDaq Framework LabVIEW User Manual** for a detailed description of the UeiDaq LabVIEW VIs.

4.6.1. Step-by-step tutorial – Trivial Analog Input example – LabVIEW

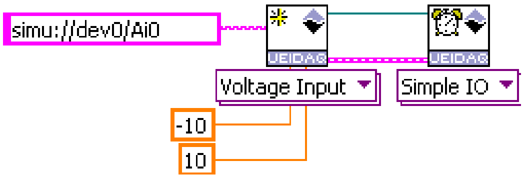
To create a simple analog input ranging between -10 to 10 V to render on-screen, begin by creating a new VI in LabVIEW. Window > Show Block View Mode.

The process of operating a session in chapter 3 applies to LabVIEW: create a session, configure session, configure timing, configure triggers, start session operation, read or write (once or repeatedly), and end acquisition.

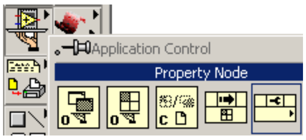
Create a session using the UeiDaqCreateSession.vi:
The relevant inputs for our VI are the minimum and maximum voltages as floating-point numbers and a resource to set as a string.



Configure the session:
Create a string constant, "simu://dev0/Ai0", and connect the string to the resource port. Create two numeric constants, "-10" and "10", and connect them to the minimum and maximum range, respectively.
Create a UeiDaqConfigureTiming.vi and *configure the timing* for Simple IO.
No *triggering* is required for this simple continuous analog input sampling.



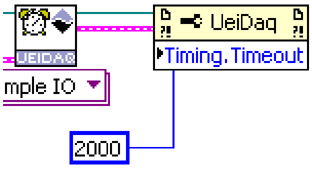
The UeiDaq LabVIEW VIs only provide the most commonly used functions. Less common functions are only available as generic Property Node blocks. To access these functions, create a Property Node as seen on the right.



Attach the refnum and error lines in from the session's Timing block; this will change the generic "App" Property Node to a "UeiDaq" Property Node.

Left-click on Property, select Timing > Timeout.

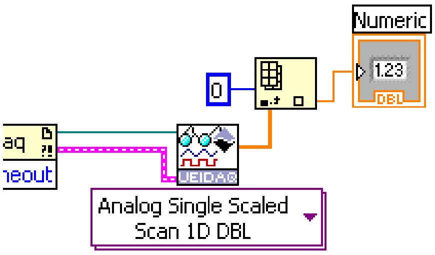
This property is an input; by default Property Node properties are set to output. Set the UeiDaq Property Node to input by right-clicking the block and selecting Change to Write. Finally, to create the constant: right-click Timing.Timeout > Create > Constant. Set the constant to 2000 milliseconds.



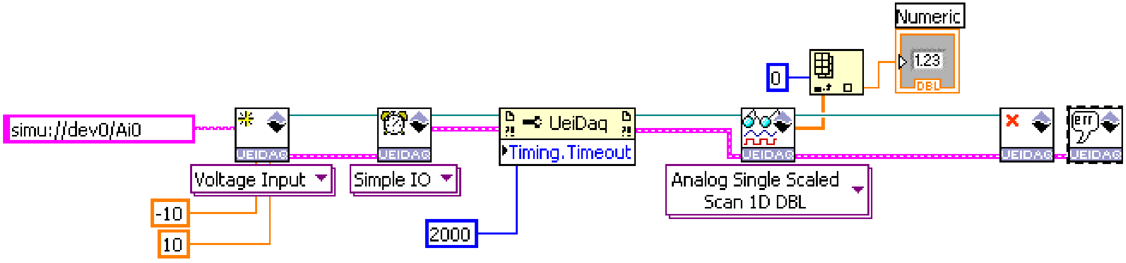
Note: to see more variables that can be changed (after the tutorial is finished) drag the edge of the UeiDaq Product Node down to resize it.

To *read the data*, create a UeiDaqRead.vi node. Set the node to “Analog Single Scaled Scan 1D DBL”, and connect the refnum and error lines.

On the front panel, create a numeric indicator. On the back panel, move the numeric display close to the UeiDaqRead node. The UeiDaqRead node generates an array of scaled doubles; each index of the array is the channel (for example, channel 0 is index 0). To output only this index, create an Index Array node. Connect the output from the UeiDaqRead to the “Array” input of the Index Array node. Connect the output of the Index array to the Numeric Indicator node. To define the channel for this operation, create an index. Right-click the index connector on the Index Array node and Create > Constant; set the index to the channel number, 0.



To *stop the session* cleanly, create a UeiDaqCloseSession.vi to close the session. Connect the refnum and error wires to the Reader node. Optionally, you may also want to add a UeiDaqShowError.vi to translate errors into meaningful text (if any); connect the final Error Out wire to this VI. The final diagram should resemble this one:

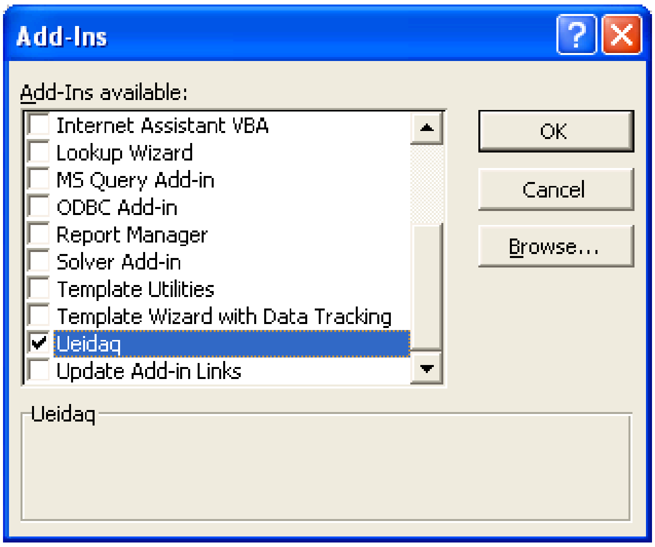


Congratulations – you have built a trivial analog input reader that takes a single reading. On the front-panel, click “Run” to try it.
This VI is ready to be used with an analog-input capable device; substitute the *simu* driver with *pwrdaq* or *pdna* device to get a value from your attached hardware.

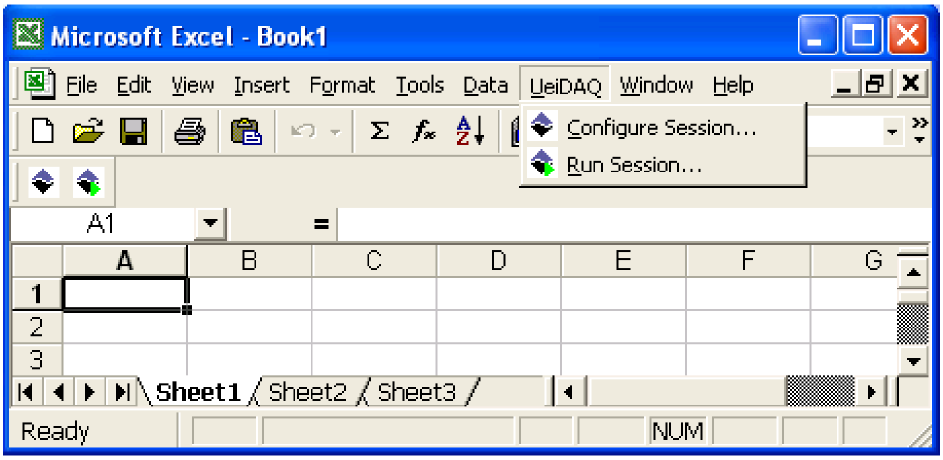
Instead of using a numeric indicator, a chart can be used (just replace the indicator) – open “Charting Acquire & Chart PointByPoint.vi” to use as an excellent example of simple software analog input. The “Charting Acquire & Chart Continuous.vi” provides an excellent example of hardware analog input; you will notice the use of the UeiDaqStart.vi node and the UeiDaqStop.vi node. These nodes are used to explicitly start and stop data acquisition; and should be used in normal acquisition once you are more familiar with the framework.

4.7. UeiDaq framework Excel Add-In

The UeiDaq framework comes with an Excel add-in allowing you to acquire data directly into an excel spreadsheet.
To use the add-in you first need to activate it. In Excel go to the tools/Add-ins… menu option and make sure that the UeiDaq add-in is activated:



After clicking OK, a new menu and toolbar should appear:



The “Configure Session…” option lets you configure the device and session parameters to use (see section about the session configurator).

The “Run Session…” option runs the last configured session and copy the acquired data in the active spreadsheet starting at the selected cell.

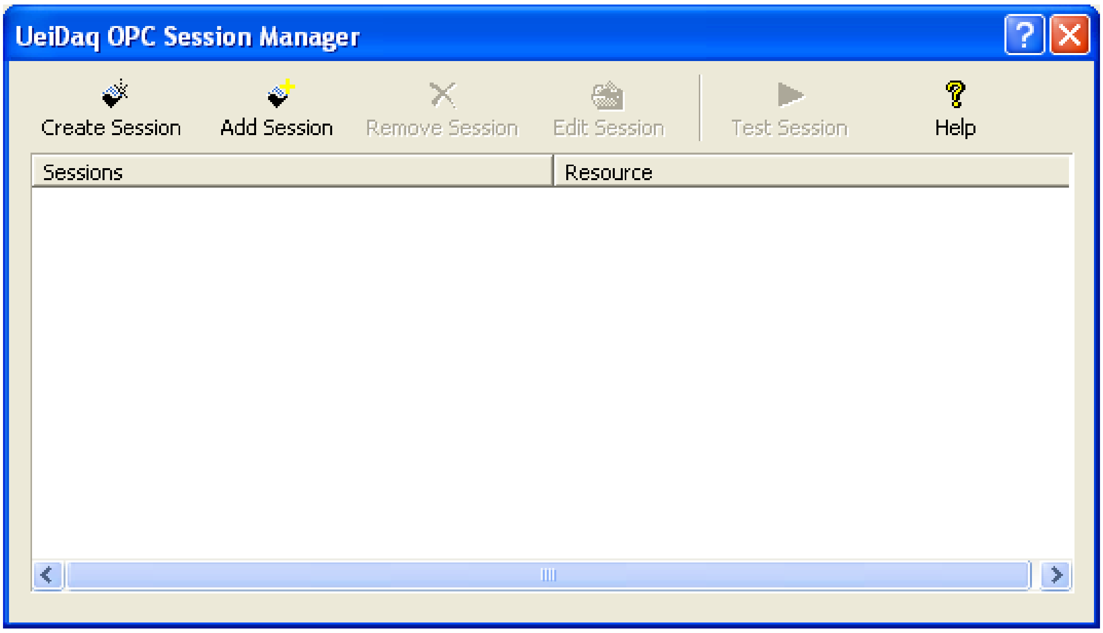
Session parameters are automatically saved so you don’t have to reconfigure the session each time you open Excel.

4.8. UeiDaq Framework OPC server

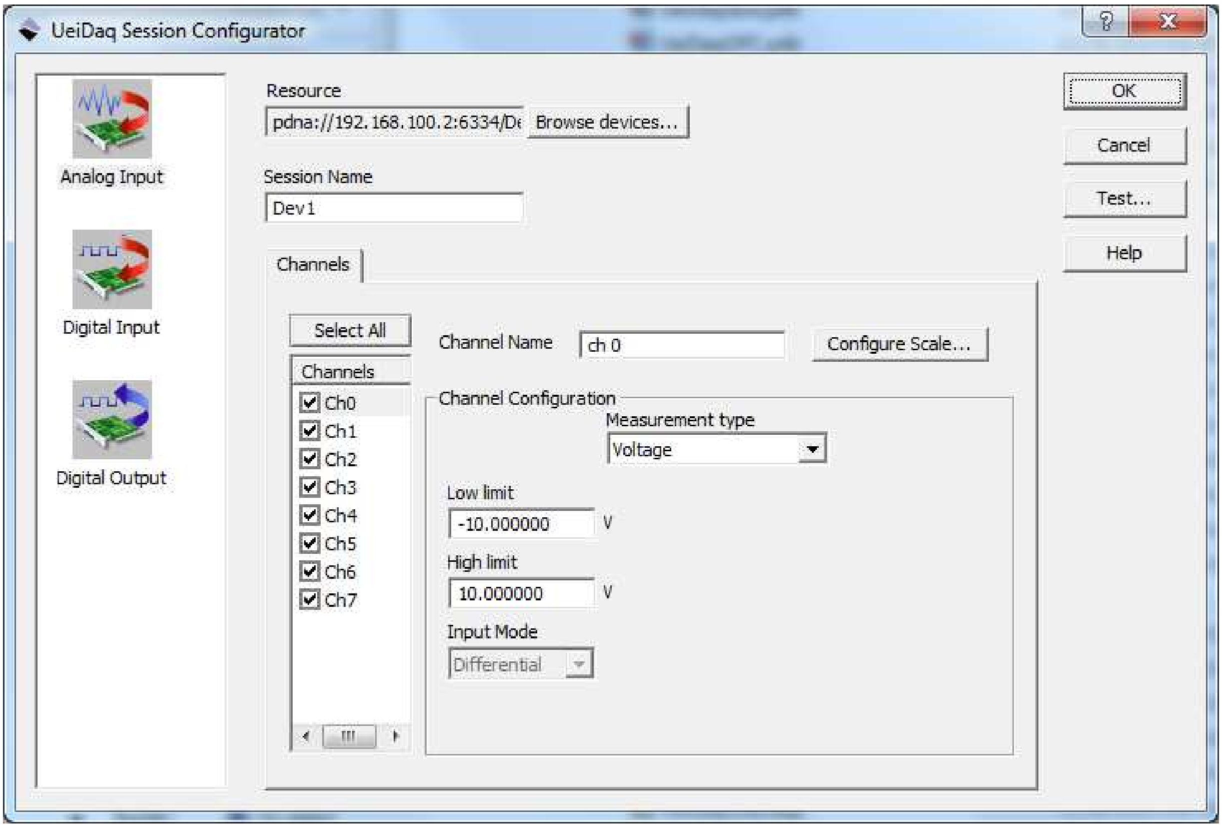
4.8.1. Configuring OPC items

The UeiDaq Framework’s OPC server allows you to acquire data from any software package that can be an OPC client. Most of the HMI and Supervisory software packages such as Wonderware Intouch, RSView or iFix are OPC clients and will be able to access data acquired from the UeiDaq Framework’s OPC server.

You first need to run the UeiDaq OPC Configurator to configure which devices and channels will be visible from an OPC client. You can run it from the Start/Programs/UEI/Framework/OPC menu.



If it is the first time you run the OPC configurator, the list of sessions will be empty and you first need to create one. Clicking on the “Create Session” button opens up a dialog box to configure the new session’s parameters.



Resource: pdna://192.168.100.2:6334/Dε

Session Name: Dev1

Channels:
- ☑ Ch0
- ☑ Ch1
- ☑ Ch2
- ☑ Ch3
- ☑ Ch4
- ☑ Ch5
- ☑ Ch6
- ☑ Ch7

Channel Name: ch 0

Measurement type: Voltage

Low limit: -10.000000 V

High limit: 10.000000 V

Input Mode: Differential

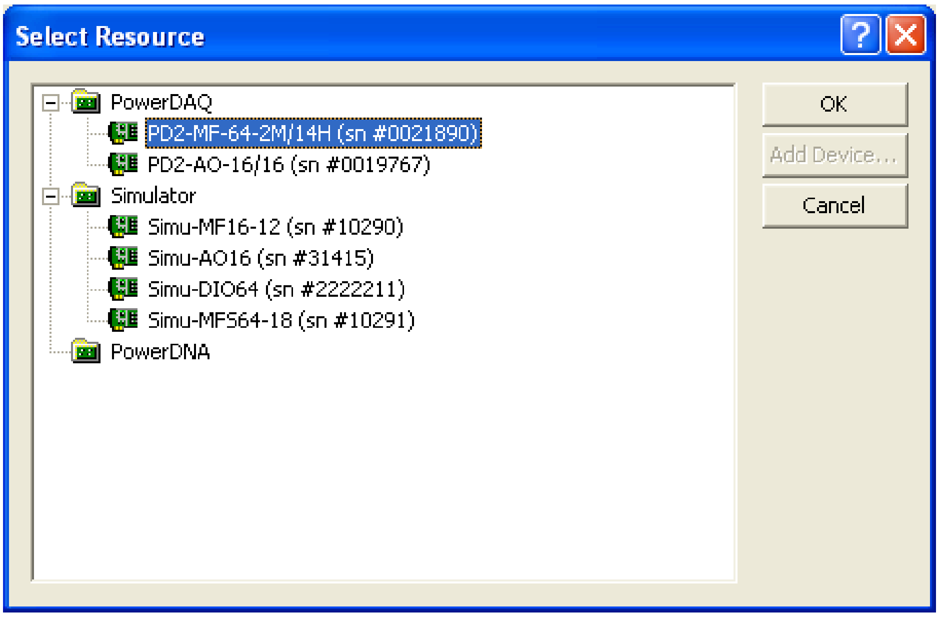
Here is a typical session configuration sequence:

- Select the session type by clicking on the left side session types list control.
- Select the device by clicking on the “Browse devices…” button.
- Select the channels in the channel list control.
- Configure channels parameters. The channel configuration parameters pertain to the selected channel(s) in the channel list. You can select multiple channels by holding down the CTRL key while clicking on each channel.

Click Ok to validate the new session or Cancel to go back to the session configurator without creating any new session. You can also click on the “Test…” button to open up the session test panel and verify that you can read or write data from or to the device.

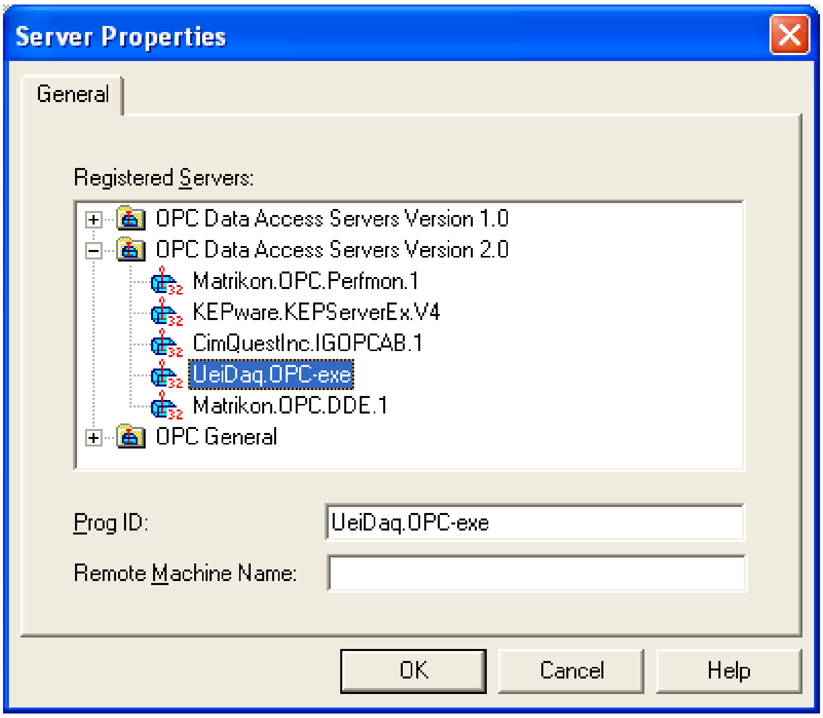
Each configured channel will be accessible through a dedicated OPC item. The OPC item is named by default using the following pattern: **device class/device id/subsystem name/channel id**
You can use an arbitrary name instead by setting the **Session Name** and **Channel Name** fields. The OPC item will then be named <**Session Name>/<Channel Name>**

The device selection dialog only lists available PowerDAQ and Simulator devices. To select PowerDNA devices, you need to select the "PowerDNA" item and click on the "Add Device…" to enter the IP address of the PowerDNA cube you wish to use. After entering the IP address, the PowerDNA devices will show-up in the device selection dialog as follows:



4.8.2. Selecting the UeiDaq OPC server in your client

When browsing for available servers on a remote machine or your local computer select the OPC server “UeiDaq.OPC-exe”.



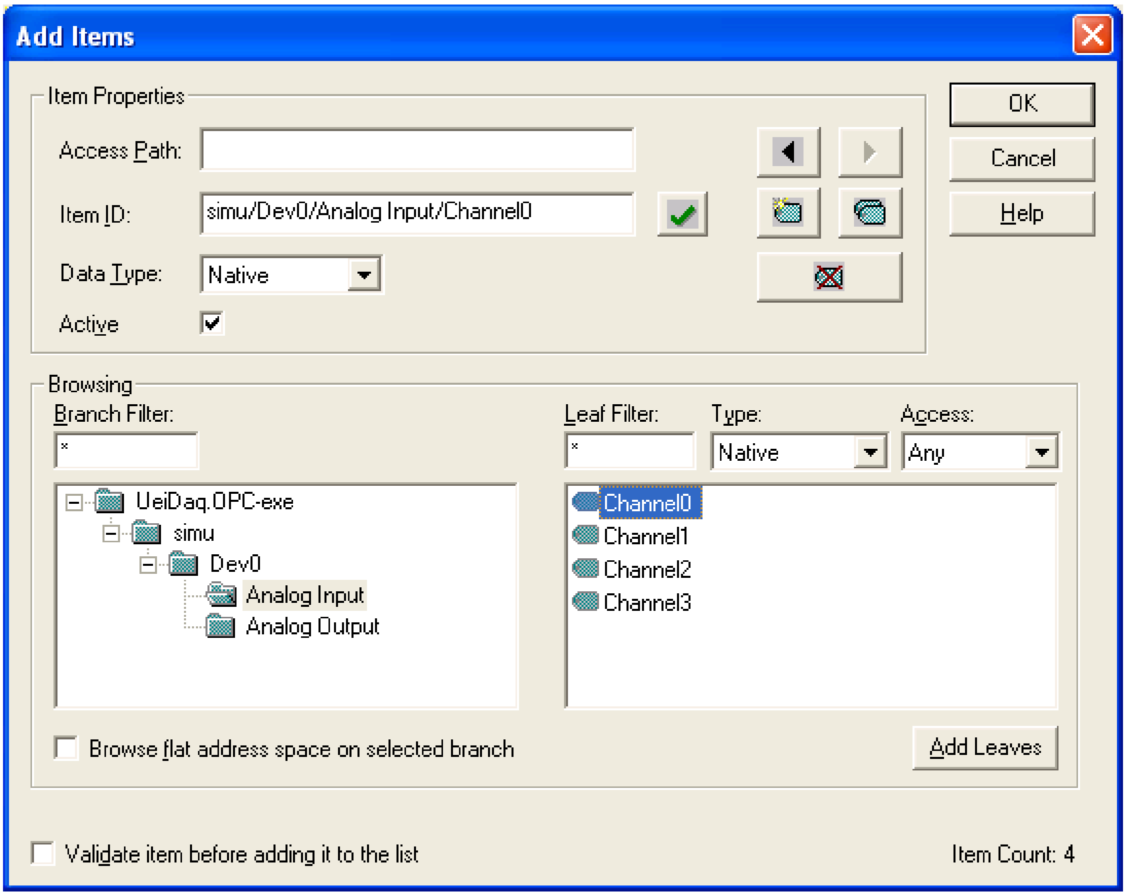
Registered Servers:

- OPC Data Access Servers Version 1.0
- OPC Data Access Servers Version 2.0
  - Matrikon.OPC.Perfmon.1
  - KEPware.KEPServerEx.V4
  - CimQuestInc.IGOPCAB.1
  - UeiDaq.OPC-exe
  - Matrikon.OPC.DDE.1
- OPC General

Prog ID: UeiDaq.OPC-exe

Remote Machine Name:

Then browse through the OPC items provided by the server and you should see list of items corresponding to the channels you configured in the "UeiDaq OPC Session manager".



Add Items

Item Properties

Access Path:

Item ID: simu/Dev0/Analog Input/Channel0

Data Type: Native

Active ☑

Browsing

Branch Filter: *

Leaf Filter: *

Type: Native

Access: Any

UeiDaq.OPC-exe
  simu
    Dev0
      Analog Input
      Analog Output

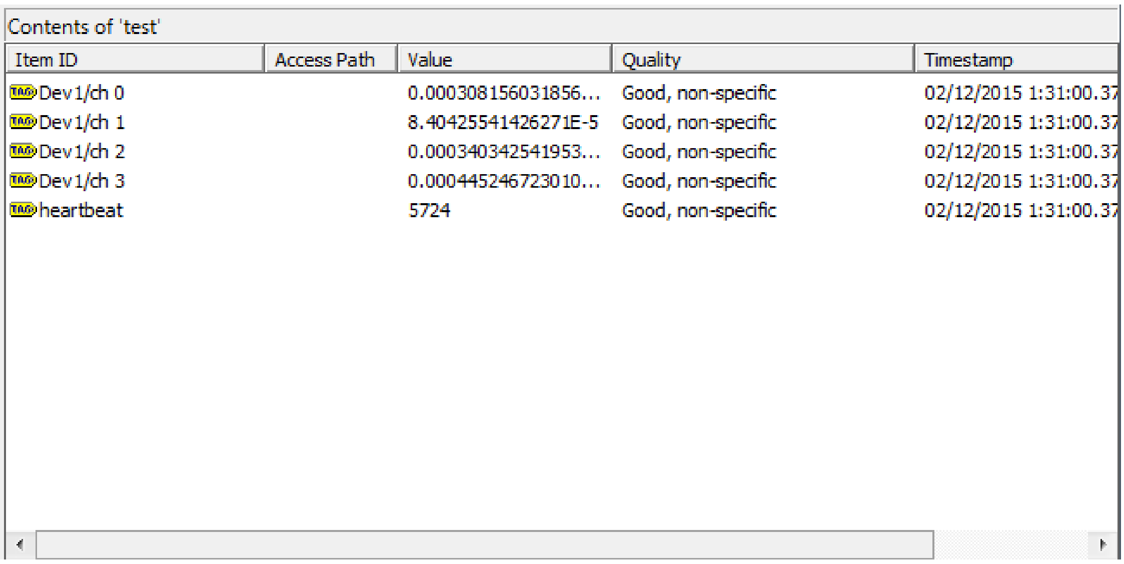
Channel0
Channel1
Channel2
Channel3

☐ Browse flat address space on selected branch

☐ Validate item before adding it to the list

Item Count: 4

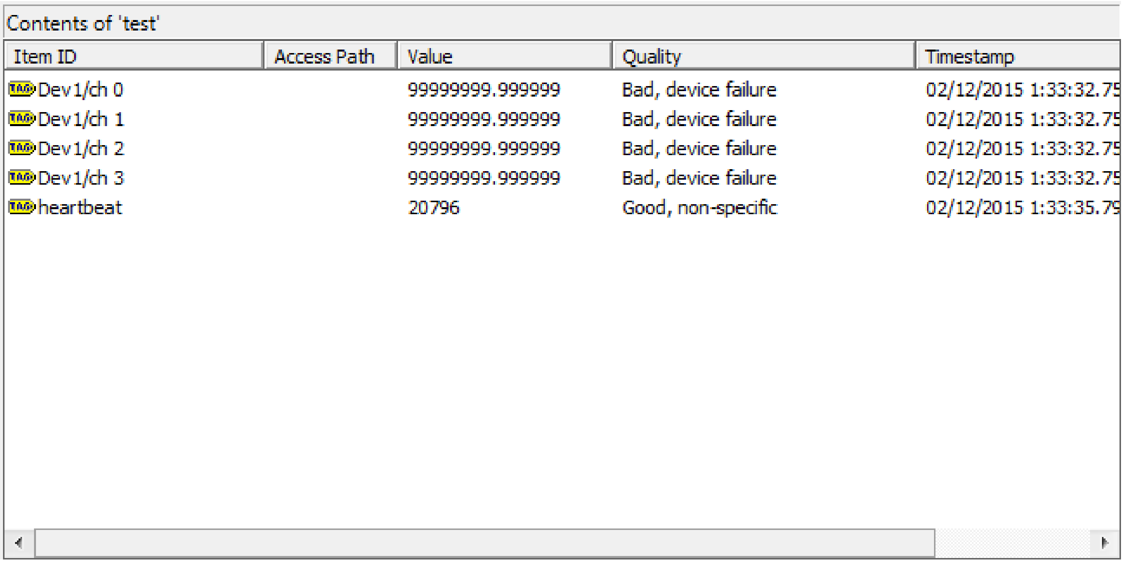
The heartbeat item is a software counter that starts incrementing once the server is started. It will stop incrementing immediately if for some reason the server can’t read new values from the device.



Contents of 'test'

Item ID	Access Path	Value	Quality	Timestamp
Dev1/ch 0		0.000308156031856...	Good, non-specific	02/12/2015 1:31:00.37
Dev1/ch 1		8.40425541426271E-5	Good, non-specific	02/12/2015 1:31:00.37
Dev1/ch 2		0.000340342541953...	Good, non-specific	02/12/2015 1:31:00.37
Dev1/ch 3		0.000445246723010...	Good, non-specific	02/12/2015 1:31:00.37
heartbeat		5724	Good, non-specific	02/12/2015 1:31:00.37

The UeiDaq OPC server automatically sets the OPC item quality to “bad” if the connection is lost with Ethernet based devices. This can take a few seconds which is why it is important to monitor the heartbeat tag.

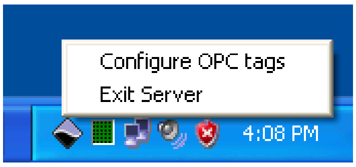


Contents of 'test'

Item ID	Access Path	Value	Quality	Timestamp
Dev1/ch 0		99999999.999999	Bad, device failure	02/12/2015 1:33:32.75
Dev1/ch 1		99999999.999999	Bad, device failure	02/12/2015 1:33:32.75
Dev1/ch 2		99999999.999999	Bad, device failure	02/12/2015 1:33:32.75
Dev1/ch 3		99999999.999999	Bad, device failure	02/12/2015 1:33:32.75
heartbeat		20796	Good, non-specific	02/12/2015 1:33:35.79

The OPC server automatically reconnects with devices once they are back online.

Once the UeiDaq OPC server is started it creates an icon in your Windows's system tray. Right-click on the icon and a menu appears allowing you to force the server to shut down or launch the "UeiDaq OPC Session manager" to re-configure the items exported by the server.



5. Synchronization using the UeiDaq framework

This chapter provides instructions for synchronizing multiple layers in multiple I/O modules using the framework API.

The following synchronization use cases are described in subsequent sections:

- Starting I/O layers simultaneously within one IO module with a software trigger.
- Synchronizing multiple I/O modules using an external clock connected to the Sync connector (the clock can optionally be provided by one of the I/O modules).
- Synchronizing multiple I/O modules using a 1PPS timing signal or the IEEE-1588/PTP standard.

5.1. Starting I/O layers simultaneously with a software trigger

This method of synchronization only applies to I/O layers located in the same cube or rack. The I/O layers derive their clock from the same timebase so it is enough to start them simultaneously to ensure that they remain synchronized over time.

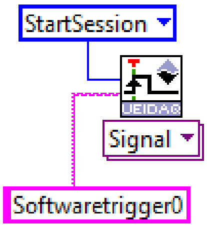
There are four software trigger signals. Each trigger is selected using its name: “softwaretrigger0”, “softwaretrigger1”, “softwaretrigger2”, “softwaretrigger3”.

Each session must be configured to start upon a software trigger.

C++:

```
pSession->CreateAIChannel("pdna://192.168.100.2/dev0/ai0:3", -10.0,10.0,
                          UeiAIChannelInputModeDifferential);

pSession->ConfigureTimingForBufferedIO(1000, UeiTimingClockSourceInternal,100.0,
                                       UeiDigitalEdgeRising,
                                       UeiTimingDurationContinuous);

pSession->ConfigureSignalTrigger(UeiTriggerActionStartSession,
                                       "SoftwareTrigger0");
```

LabVIEW:

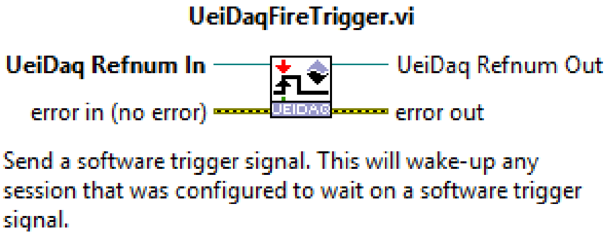


Any of the triggered sessions can emit the trigger signal that will start all sessions simultaneously.

C++:

```
pSession->Start();
pSession->GetStartTrigger()->Fire();
```

LabVIEW:



Send a software trigger signal. This will wake-up any session that was configured to wait on a software trigger signal.

5.2. Synchronizing I/O modules with external clock connected to sync connector

This method uses an external clock connected to the sync input connector. The external clock can come from some external equipment or can be generated by one of the I/O modules using phase-locked loop (PLL) circuitry on the CPU board.

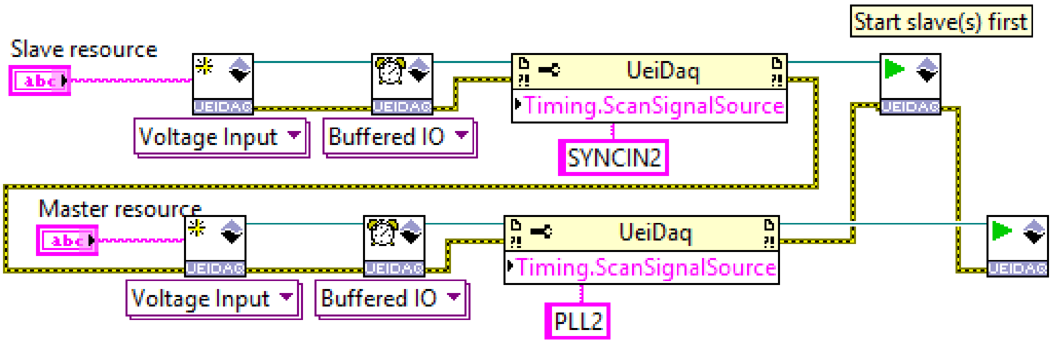
The CPU layer is equipped with a digital PLL that can produce a user-specified frequency signal. The PLL signal can be routed to the I/O layers using one of the four synchronization lines available on the backplane.
The PLL signal can also be shared with other IO modules via the sync connector to synchronize I/O layers across multiple racks or cubes.

I/O layers on the master IOM (the one generating the external clock with its PLL) must set the scan clock signal to “PLLx” where x is 0, 1, 2 or 3 to specify the synchronization line used to connect the PLL output to each I/O layer. When omitted, the default sync line is 0.

I/O layers on the slave IOMs must set the scan clock signal to “SYNCINx” where x is 0, 1, 2 or 3 to specify the synchronization line used to connect the Sync connector clock input to each I/O layer. When omitted, the default sync line is 0.

C++:
```
pMasterSession->CreateAIChannel("pdna://192.168.100.2/dev0/ai0:3", -10.0,10.0,
                                 UeiAIChannelInputModeDifferential);
pMasterSession->ConfigureTimingForBufferedIO(1000,
                                 UeiTimingClockSourceExternal,100.0,
                                 UeiDigitalEdgeRising,
                                 UeiTimingDurationContinuous);
pMasterSession->GetTiming()->SetScanClockSourceSignal("PLL2");


pSlaveSession->CreateAIChannel("pdna://192.168.100.3/dev0/ai0:3", -10.0,10.0,
                                 UeiAIChannelInputModeDifferential);
pSlaveSession->ConfigureTimingForBufferedIO(1000,
                                 UeiTimingClockSourceExternal,100.0,
                                 UeiDigitalEdgeRising,
                                 UeiTimingDurationContinuous);
pMasterSession->GetTiming()->SetScanClockSourceSignal("SYNCIN2");

// start slave session first
pSlaveSession->Start();
pMasterSession->Start();
```

LabVIEW:



An inconvenience of this method is that the frequency of the clock signal limits the length of the sync cable between IO modules to a few feet.

Some I/O layers are over-clocking their ADC and need an external scan clock that runs faster (usually 8x) than the rate configured by the user. You can’t use this method synchronize over-clocked and non over-clocked I/O layers.

5.3. Synchronizing I/O modules with 1PPS timing signal or IEEE-1588 PTP standard

This section describes multi-chassis synchronization:

- using a pulse-per-second (PPS) timing signal connected via the sync input connector
- using the IEEE-1588 Precision Time Protocol (PTP) standard over Ethernet

For PPS synchronization, the timing signal can come from external equipment or can be generated by one of the I/O modules using 1PPS generation circuitry on the CPU board.

For PTP synchronization, a PTP master sends Sync timing packets over Ethernet. Firmware on the CPU of each PTP slave chassis processes the PTP Sync packets, which are used by CPU circuitry to generate a local PPS timing signal for internal I/O board clock generation and synchronized triggers.

The PPS timing signal is usually a one pulse per second (1PPS) signal, but it doesn't have to be. I/O modules can work with any nPPS timing signal, where n is a positive integer.

PPS/PTP synchronization uses an adaptive digital phase-locked loop (ADPLL) circuit and an event module to produce a clock at a user-programmable rate that stays synchronized with the 1PPS input timing signal.

Multiple racks can easily be synchronized when using the same 1PPS synchronization clock or, for PTP synchronization, a PPS synchronization clock derived internally from PTP packets from the same PTP master.

The ADPLL and event module on each IOM produce synchronized scan clocks.

The following describe 1PPS synchronization mechanics:

- The source of the 1PPS sync clock can be an internally or externally generated 1PPS or derived from PTP packets
- The raw 1PPS clock is routed via one of the four internal sync lines to the ADPLL
- The ADPLL locks on the raw 1PPS and outputs its own 1PPS that is an average of the original 1PPS clock
- The ADPLL can maintain its 1PPS output even if the original 1PPS clock gets disconnected
- The ADPLL 1PPS output provides a synchronized reference signal to the event module

- The ADPLL 1PPS output can also be routed to one of the four internal sync lines
- The event module produces a user selectable number of pulses upon every 1PPS pulse coming from the ADPLL to be used as a synchronized clock source
- The event module clock output is routed to the I/O boards via one of the remaining sync lines

5.3.1. Creating a 1PPS or PTP Synchronization session

5.3.1.1. 1PPS Synchronization session

Synchronization to a 1PPS pulse uses a specialized session to configure the ADPLL and event module of each I/O module.

A session for synchronization must be created on each IO module. The resource string must point to the CPU device (device 14).

The following are examples of supported resource strings:

- `pdna://192.168.100.2/dev14/sync0`
- `pdna://192.168.100.2/cpu/sync0`

The following are 1PPS synchronization session parameters (also used for PTP synchronization as described in section 5.3.1.2):

Parameter	Description
Mode	The method used to obtain the 1PPS signal • **Clock** uses a 1PPS signal produced internally or received from an external source • **NTP** derives the 1PPS from an NTP server • **1588** derives the 1PPS from a 1588 time master • **IRIG** uses the 1PPS produced by an IRIG-650
Source	The source of the 1PPS clock in SyncClock mode: • **Internal** when 1PPS is generated internally • **Input0** when external 1PPS is connected to input 0 of sync connector (clock input) • **Input1** when external 1PPS is connected to input 1 of sync connector (trigger input)

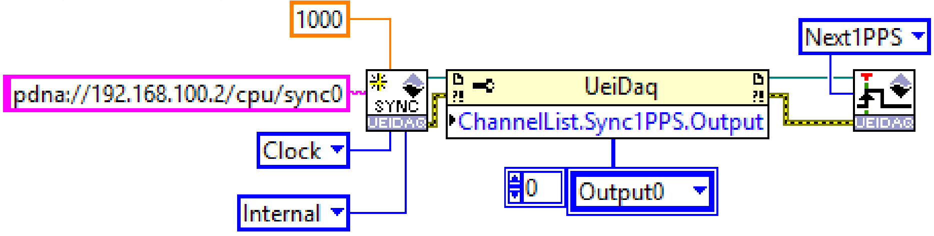
Parameter	Description
Output	The pin used to output the 1PPS sync clock • **Output0** when 1PPS is emitted out of output 0 of sync connector (clock output) • **Output1** when 1PPS is emitted out of output 1 of sync connector (trigger output)
NumPPS	The number of pulses per second used by the synchronization clock signal
PPSAccuracyUs	The required accuracy in microseconds of the 1PPS timing signal. Timing signals that are out of range are ignored
ADPLLSyncLine	The internal sync line connecting the internal or external 1PPS clock to the ADPLL
ADPLLOutputSyncLine	The ADPLL locked 1PPS output can optionally be routed to a sync line for debug purposes
EMOutputSyncLine	The internal sync line connecting the event module output to the I/O layers
TriggerOutputSyncLine	The sync line connecting the synchronized trigger to the I/O layers. The CPU is capable of emitting a trigger signal simultaneously with the next 1PPS pulse
EMOutputRate	The rate (Hz) of the clock produced by the event module

The example below creates a 1PPS synchronization session for a master:

C++ (PPS master):

```
// Create session using internal 1PPS synchronization signal
CUeiSync1PPSPort* pMasterSyncPort =
  pMasterSyncSession->CreateSync1PPSPort(
                    "pdna://192.168.100.2/cpu/sync0",
                    UeiSyncClock,
                    UeiSync1PPSInternal,
                    1000.0);

// Output 1PPS out of sync connector
pMasterSyncPort->Set1PPSOutput(UeiSync1PPSOutput0);

// Configure trigger to start session on a 1PPS edge
pMasterSyncSession->GetStartTrigger()->
                    SetTriggerSource(UeiTriggerSourceNext1PPS);
```

LabVIEW (PPS master):

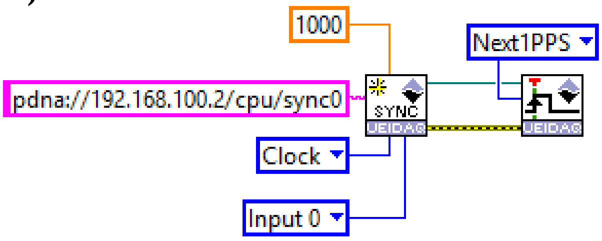


The example code below creates a 1PPS synchronization session for a slave:

C++ (PPS slave):

```
// Create session using external 1PPS synchronization signal
CUeiSync1PPSPort* pSlaveSyncPort =
      pSlaveSyncSession->CreateSync1PPSPort(
                          "pdna://192.168.100.3/cpu/sync0",
                          UeiSyncClock,
                          UeiSync1PPSInput0,
                          1000.0);

// Configure trigger to start session on a 1PPS edge
pSlaveSyncSession->GetStartTrigger()->
                          SetTriggerSource(UeiTriggerSourceNext1PPS);
```

LabVIEW (PPS slave):



5.3.1.2. PTP Synchronization session

Synchronization to the PTP standard uses the same specialized session as synchronization to a 1PPS, with several additional PTP-specific parameters to be configured.

The session for synchronization must be created on each IO module. The resource string must point to the CPU device (device 14).

The following are examples of supported resource strings:

- `pdna://192.168.100.2/dev14/sync0`
- `pdna://192.168.100.2/cpu/sync0`

To determine which device in the system will be the master, firmware on the CPU board of each I/O chassis uses the IEEE-1588 best master clock algorithm (BMCA) to compare PTP clock parameters of all announced masters. If only UEI chassis are the master-capable devices in a system, the master is determined by the Priority value: the chassis assigned the lowest Priority value will be the master (see PTPPriority1/2 in table below).

The session parameters listed in the previous *1PPS synchronization session* section also apply to PTP synchronization sessions. The following table lists PTP-specific parameters:

PTP-only Parameters	**Description**
PTPEthernetPort	The Ethernet port used to send and receive PTP packets: • 0 for NIC1 (default) • 1 for NIC2
PTPSubdomain	The PTP subdomain (0 default)
PTPPriority1	The first order pre-emptive PTP priority: used in BMCA to determine slave/master. (Smaller numbers indicate higher priority. Set to 255 to configure a chassis in slave only mode)
PTPPriority2	The second order PTP priority: used in BMCA to determine slave/master. (Smaller numbers indicate higher priority)
PTPLogSyncInterval	log2(period of sync messages) How often the PTP master clock sends Sync messages
PTPLogMinDelayRequestInterval	log2(minimum space between delay requests) Minimum interval allowed between PTP delay-request messages

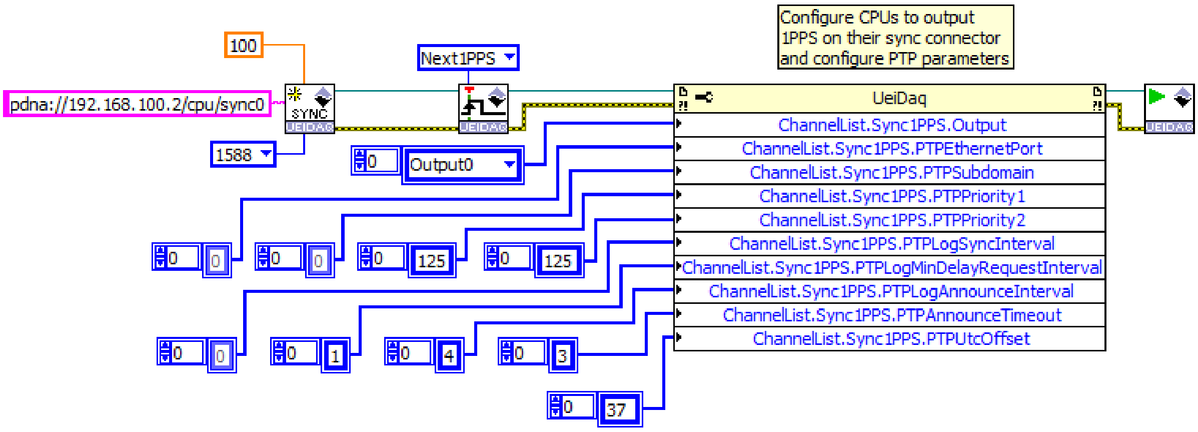
PTP-only Parameters	Description
PTPLogAnnounceInterval	log2(period of announce messages) How often the PTP master clock sends Announce messages
PTPAnnounceTimeout	Number of announce intervals allowed to transpire without the slave receiving an Announce message. After this delay, the slave will timeout and return to the state of listening for a new master PTP source
PTPUTCOffset	The UTC offset

The example below creates a PTP synchronization session for a master:

C++ (PTP master):

```
// Create master session using PTP synchronization
ptpMasterSession = new CUeiSession();
CUeiSync1PPSPort* masterPort =
     ptpMasterSession->CreateSync1PPSPort(
                          "pdna://192.168.100.2/cpu/sync0",
                          UeiSync1588,
                          UeiSync1PPSInternal,
                          100.0);

// configure port to output 1PPS signal out of
//     sync connector output 0 (clock out)
masterPort->Set1PPSOutput(UeiSync1PPSOutput0);

// Configure PTP master parameters
// In this example, the master is Priority 128-3 and
//    the slave is Priority 128+3
masterPort->SetPTPEthernetPort(0);
masterPort->SetPTPSubdomain(0);
masterPort->SetPTPPriority1(128 - 3);
masterPort->SetPTPPriority2(128 - 3);
masterPort->SetPTPLogSyncInterval(0);
masterPort->SetPTPLogMinDelayRequestInterval(1);
masterPort->SetPTPLogAnnounceInterval(4);
masterPort->SetPTPAnnounceTimeout(3);
masterPort->SetPTPUTCOffset(37);

// Configure trigger to start session on a 1PPS edge
ptpMasterSession->GetStartTrigger()->
                           SetTriggerSource(UeiTriggerSourceNext1PPS);
```

LabVIEW (PTP master):


Configure CPUs to output 1PPS on their sync connector and configure PTP parameters
100
Next1PPS
pdna://192.168.100.2/cpu/sync0
SYNC
1588
0
Output0
UeiDaq
ChannelList.Sync1PPS.Output
ChannelList.Sync1PPS.PTPEthernetPort
ChannelList.Sync1PPS.PTPSubdomain
ChannelList.Sync1PPS.PTPPriority1
ChannelList.Sync1PPS.PTPPriority2
ChannelList.Sync1PPS.PTPLogSyncInterval
ChannelList.Sync1PPS.PTPLogMinDelayRequestInterval
ChannelList.Sync1PPS.PTPLogAnnounceInterval
ChannelList.Sync1PPS.PTPAnnounceTimeout
ChannelList.Sync1PPS.PTPUtcOffset
0 0
0 0
0 125
0 125
0 0
0 1
0 4
0 3
0 37

The example code below creates a PTP synchronization session for a slave:

C++ (PTP slave):

```
// Create session using PTP synchronization
ptpSlaveSession = new CUeiSession();
CUeiSync1PPSPort* slavePort =
     ptpSlaveSession->CreateSync1PPSPort(
                         "pdna://192.168.100.3/cpu/sync0",
                         UeiSync1588,
                         UeiSync1PPSInternal, 100.0);

// Configure PTP slave parameters (configure slave as higher priority)
slavePort->SetPTPEthernetPort(0);
slavePort->SetPTPSubdomain(0);
slavePort->SetPTPPriority1(128 + 3);
slavePort->SetPTPPriority2(128 + 3);
slavePort->SetPTPLogSyncInterval(0);
slavePort->SetPTPLogMinDelayRequestInterval(1);
slavePort->SetPTPLogAnnounceInterval(4);
slavePort->SetPTPAnnounceTimeout(3);
slavePort->SetPTPUTCOffset(37);
// Configure trigger to start session on a 1PPS edge
ptpSlaveSession->GetStartTrigger()->
                          SetTriggerSource(UeiTriggerSourceNext1PPS);
```

LabVIEW (PTP slave):

Refer to the LabVIEW (PTP master) example above. In this example, the properties that distinguish a master from a slave are PTPPriority1/2. The slave will have a numerically higher Priority value than the master (e.g. the slave will have Priority 131 instead of 125).

5.3.2. Creating I/O sessions

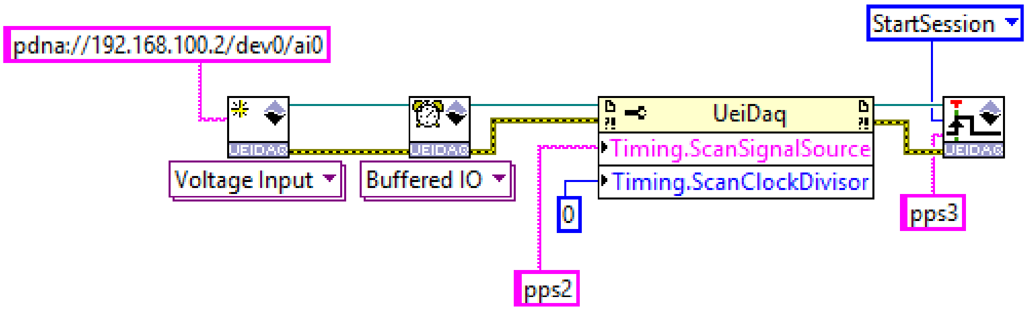
The I/O layer sessions must be configured to use an external clock provided by the 1PPS circuitry. This is done setting the scan clock signal to “PPSx” where x is 0, 1, 2 or 3 to specify the synchronization line used to connect the event module output to each I/O layer. When omitted, the default sync line is 2.

The start trigger signal must also be configured using a similar scheme.

C++:

```cpp
pIOSession->CreateAIChannel("pdna://192.168.100.2/dev0/ai0:3",
                             -10.0,10.0,
                             UeiAIChannelInputModeDifferential);

pIOSession->ConfigureTimingForBufferedIO(1000,
                             UeiTimingClockSourceExternal,100.0,
                             UeiDigitalEdgeRising,
                             UeiTimingDurationContinuous);

pIOSession->GetTiming()->SetScanClockSourceSignal("PPS2");

// Configure timebase divisor, this is necessary to allow
// non-over-clocked layers to be synchronized with over-clocked layers.
pIOSession->GetTiming()->SetScanClockTimebaseDivisor(0);

// Configure AI layers to wait for 1PPS trigger signal on sync line 3
pIOSession->
      ConfigureSignalTrigger(UeiTriggerActionStartSession, "pps3");
```

LabVIEW:



5.3.3. Starting sessions

The IO sessions must start first: this will initialize and start the I/O layers hardware that will be waiting for the trigger signal.

C++:
```
pIOSession->Start();
pIOSession2->Start();
…
```

The synchronization sessions are started next:

C++ (1PPS example):
```
// Start sync sessions
pMasterSyncSession->Start();
pSlaveSyncSession->Start();
…
```

C++ (PTP example):
```
// Start sync sessions
ptpMasterSession ->Start();
ptpSlaveSession ->Start();
…
```

5.3.1. Checking PTP status

On startup, each master-capable device in your system will announce its clock properties, and each device will compare these properties to determine which has the best properties and will be master. This process takes a few seconds to several minutes to complete, depending on the number of devices in your system.

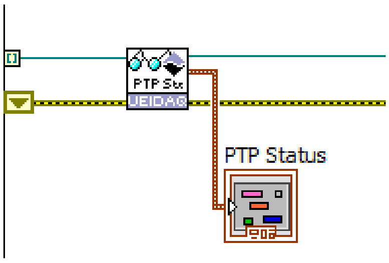
You can use a controller object to read the PTP status, which includes the PTP state (Listening, Slave, Master, etc.), MasterClockID and more. Refer to the UeiDaq Framework Reference Manual for more information about the UeiSync1PPSPTPStatus structure members.

The code below shows how to check the PTP status on the slave sync session (checking status on the master session is identical).

C++:

```
CUeiSync1PPSController* ptpSlaveController =
      new CUeiSync1PPSController(ptpSlaveSession->GetDataStream())

tUeiSync1PPSPTPStatus status;
ptpSlaveController->ReadPTPStatus(&status);

std::cout << "PTP slave state = " << status.State << std::endl;

std::cout << "Master clock ID = " << std::hex << status.MasterClockID
<<  std::endl;

std::cout << "Mean path delay = " << status.MeanPathDelay << std::endl;
```

LabVIEW:



5.3.2. Reading UTC time

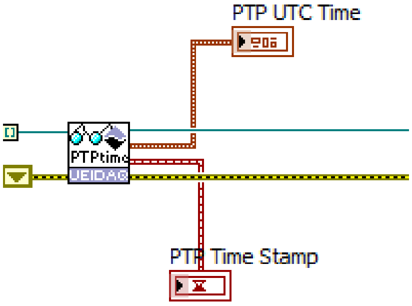
You can read the UTC time on IO devices using a controller object. Slave devices derive their time from the synchronization packets from the PTP master.

The code below shows how to read the UTC time on the slave sync session (checking time on the master session is identical).

C++:

```
CUeiSync1PPSController* ptpSlaveController =
      new CUeiSync1PPSController(ptpSlaveSession->GetDataStream())

tUeiPTPTime slaveTime;
ptpSlaveController->ReadPTPUTCTime(&slaveTime);

std::cout << "PTP UTC Time = " << slaveTime.sec << "s / ";
std::cout << " slaveTime.nsec << "ns" << std::endl;
```

LabVIEW:



5.3.3. Checking ADPLL status

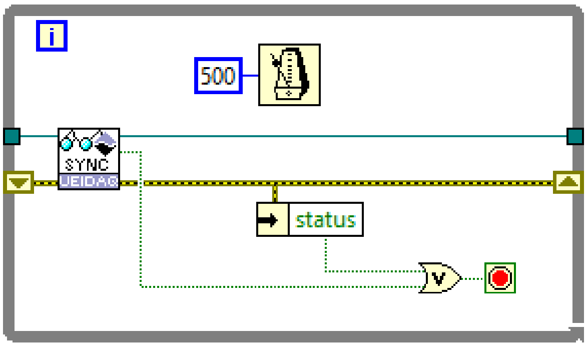
The ADPLL and event module will take a few seconds to become stable. You can use a controller object to read the ADPLL status.

The code below shows how to check the status on the master sync session (checking status on the slave session is identical).

C++:

```
CUeiSync1PPSController* pMasterSyncController =
      new CUeiSync1PPSController(pMasterSyncSession->GetDataStream())
int lockCount = 0;
int loopCount = 0;

while (lockCount < 10 && loopCount < 30)
{
   bool locked;
   pMasterSyncController->ReadLockedStatus(&locked);
   if (locked)
   {
      lockCount++;
   }
   UeiPalSleep(500);
}
if (lockCount < 10)
{
    std::out << "could not lock master 1PPS" << std::endl;
    return -1;
}
```

LabVIEW:



5.3.4. Sending trigger on next PPS

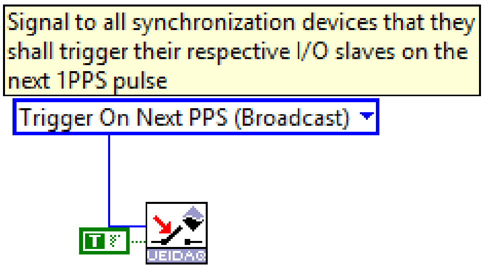
Once all IO modules are locked on the 1PPS timing signal (either from an external source or derived from internal / PTP circuitry), the host can broadcast a “trigger on next PPS” command to all configured IO modules (slaves and/or master).

After receiving the command, the trigger circuitry on the CPU boards of each IO module waits for the next PPS edge to assert the trigger sync line, leaving one full second for all IO modules to receive and process the command and ensuring that all I/O layers across the multiple IO modules are started simultaneously.

C++:

```
// Send trigger signal to start clocking the AI layers on the next 1PPS pulse
pMasterSyncController->TriggerDevices(UeiSync1PPSTriggerOnNextPPSBroadCast, true);
```

LabVIEW:



Signal to all synchronization devices that they shall trigger their respective I/O slaves on the next 1PPS pulse

Trigger On Next PPS (Broadcast)

Appendix A: Error Codes

C/C++ Constant **(*.NET Constant*)**	**Code**	**Description**
`UEIDAQ_SUCCESS` (*UeiDaq.Error.Success*)	0	Success
`UEIDAQ_ATTRIBUTE_INVALID_ERROR` (*UeiDaq.Error.AttributeInvalid*)	0x80290001	The specified attribute doesn't exist
`UEIDAQ_ATTRIBUTE_BAD_TYPE_ERROR` (*UeiDaq.Error.AttributeBadType*)	0x80290002	The specified attribute exists but the type is wrong
`UEIDAQ_TIMEOUT_ERROR` (*UeiDaq.Error.Timeout*)	0x80290003	Timeout occurred
`UEIDAQ_BAD_PARAMETER_ERROR` (*UeiDaq.Error.BadParameter*)	0x80290004	One of the specified parameter(s) is invalid
`UEIDAQ_DEVICE_INVALID_ERROR` (*UeiDaq.Error.DeviceInvalid*)	0x80290005	The specified device doesn't exist
`UEIDAQ_CHANNEL_INVALID_ERROR` (*UeiDaq.Error.ChannelInvalid*)	0x80290006	The specified channel doesn't exist
`UEIDAQ_ATTRIBUTE_OUT_OF_RANGE_ERROR` (*UeiDaq.Error.AttribureOutOfRange*)	0x80290007	The specified attribute's value is out of range
`UEIDAQ_DEVICE_CAPABILITY_ERROR` (*UeiDaq.Error.DeviceCapability*)	0x80290008	The device is not capable of such an operation
`UEIDAQ_BAD_RESOURCE_STRING_ERROR` (*UeiDaq.Error.BadResourceString*)	0x80290009	The resource string is incorrectly formatted
`UEIDAQ_DRIVER_INVALID_ERROR` (*UeiDaq.Error.DriverInvalid*)	0x8029000A	The specified driver doesn't exist
`UEIDAQ_SESSION_INVALID_ERROR` (*UeiDaq.Error.SessionInvalid*)	0x8029000B	The session is invalid
`UEIDAQ_SUBSYSTEM_ALREADY_USED_ERROR` (*UeiDaq.Error.SubSystemAlreadyUsed*)	0x8029000C	The subsystem is already reserved by another session
`UEIDAQ_INVALID_STATE_ERROR` (*UeiDaq.Error.InvalidState*)	0x8029000D	The session is not in a state allowing this operation

C/C++ Constant **(*.NET Constant*)**	**Code**	**Description**
`UEIDAQ_BUFFER_OVERRUN_ERROR` (*UeiDaq.Error.BufferOverrun*)	0x8029000E	Data was lost because it was not read fast enough
`UEIDAQ_BUFFER_UNDERRUN_ERROR` (*UeiDaq.Error.BufferUnderrun*)	0x8029000F	Not enough data was available to keep the generation running
`UEIDAQ_SUBSYSTEM_BUSY_ERROR` (*UeiDaq.Error.SubSystemBusy*)	0x80290010	The subsystem associated with this session is already used by another session
`UEIDAQ_SUBSYSTEM_INTERNAL_ERROR` (*UeiDaq.Error.SubSystemInternalError)*	0x80290011	An unknown error occurred in the plugin
`UEIDAQ_BUFFER_ERROR` (*UeiDaq.Error.BufferError*)	0x80290012	An error occurred while reading/writing the buffer
`UEIDAQ_DEVICE_ERROR` (*UeiDaq.Error.DeviceError*)	0x80290013	An error occurred while accessing the device
`UEIDAQ_SUBSYSTEM_NOT_BUFFERED_ERROR` (*UeiDaq.Error.SubsystemNotBuffered*)	0x80290014	This subsystem doesn't support buffered IO
`UEIDAQ_NOT_ENOUGH_MEMORY_ERROR` (*UeiDaq.Error.NotEnoughMemory*)	0x80290015	There was not enough memory to complete this operation
`UEIDAQ_WRONG_DATA_TYPE_ERROR` (*UeiDaq.Error.WrongDataType*)	0x80290016	The specified data type doesn't match the device capability
`UEIDAQ_NO_MORE_ITEMS_ERROR` (*UeiDaq.Error.NoMoreItems*)	0x80290017	There is no more data available
`UEIDAQ_OPERATION_INVALID_ERROR` (*UeiDaq.Error.OperationInvalid*)	0x80290018	The operation is incompatible with the current session
`UEIDAQ_NOT_ENOUGH_DATA_ERROR` (*UeiDaq.Error.NotEnoughData*)	0x80290019	There were not enough data passed to complete the operation
`UEIDAQ_INVALID_TC_TYPE_ERROR` (*UeiDaq.Error.ThermocoupleTypeInvalid*)	0x8029001A	The thermocouple type is invalid
`UEIDAQ_INVALID_TEMP_SCALE_ERROR` (*UeiDaq.Error.TemperatureScaleInvalid*)	0x8029001B	The temperature scale is invalid
`UEIDAQ_INVALID_BRIDGE_CONFIG_ERROR` (*UeiDaq.Error.BridgeConfigurationInvalid*)	0x8029001C	The strain gauge bridge configuration is invalid

C/C++ Constant (.NET Constant)	Code	Description
`UEIDAQ_INVALID_RTD_CONV_ERROR` (*UeiDaq.Error.RTDConversionInvalid*)	0x8029001D	The RTD conversion is invalid
`UEIDAQ_DIVIDE_BY_ZERO_ERROR` (*UeiDaq.Error.DivideByZero*)	0x8029001E	Division by zero
`UEIDAQ_CONVERSION_ERROR` (*UeiDaq.Error.Conversion*)	0x8029001F	An error happened while converting units
`UEIDAQ_INVALID_RATE_ERROR` (*UeiDaq.Error.RateInvalid*)	0x80290020	The specified rate is above the device maximum rate or violates the settling time
`UEIDAQ_CTR_TMR_MODE_INVALID_ERROR` (*UeiDaq.Error.CounterTimeModeInvalid*)	0x80290021	The counter/timer does not support the specified mode
`UEIDAQ_CTR_TMR_IN_USE_ERROR` (*UeiDaq.Error.CounterTimeModeInUse*)	0x80290022	The counter/timer is already used by another session
`UEIDAQ_CAN_NOT_REGENERATE_ERROR` (*UeiDaq.Error.CanNotRegenerate*)	0x80290023	The device doesn't support regeneration
`UEIDAQ_CHANNEL_LIST_POWER_OF_2_ERROR` (*UeiDaq.Error.ChannelListPowerOf2*)	0x80290024	The Channel list size must be a power of 2
`UEIDAQ_SESSION_XML_ERROR` (*UeiDaq.Error.SessionXMLError*)	0x80290025	The XML session string is invalid
`UEIDAQ_DIGITAL_PORT_IN_USE_ERROR` (*UeiDaq.Error.DigitalPortAlreadyInUse*)	0x80290026	The digital port is already used by another session
`UEIDAQ_INVALID_TIMING_MODE_ERROR` (*UeiDaq.Error.TimingModeInvalid*)	0x80290027	The device associated with the session doesn't support the specified timing mode
`UEIDAQ_ASYNC_IN_PROGRESS_ERROR` (*UeiDaq.Error.AsyncOperationInProgress*)	0x80290028	An asynchronous operation is already in progress
`UEIDAQ_NOT_IMPLEMENTED_ERROR` (*UeiDaq.Error.NotImplemented*)	0x80290029	The feature is not yet implemented
`UEIDAQ_CONFIG_INVALID_ERROR` (*UeiDaq.Error.ConfigurationInvalid*)	0x8029002A	The current configuration settings were rejected by the device
`UEIDAQ_DEVICE_NOT_RESPONDING_ERROR` (*UeiDaq.Error.DeviceNotResponding*)	0x8029002B	The device is not responding, check the connection and the device's status

C/C++ Constant **(*.NET Constant*)**	**Code**	**Description**
`UEIDAQ_INVALID_TRIGGER_SIGNAL_ERROR` (*UeiDaq.Error.InvalidTriggerSignal*)	0x8029002C	The specified trigger signal is not compatible with the device associated to this session
`UEIDAQ_SIGNAL_BUSY_ERROR` (*UeiDaq.Error.SignalIsBusy*)	0x8029002D	The specified trigger or clock can't be routed to the device
`UEIDAQ_INVALID_INPUT_MODE_ERROR` (*UeiDaq.Error.InvalidInputMode*)	0x8029002E	The device doesn't support the specified input mode
`UEIDAQ_CALIBRATION_ERROR` (*UeiDaq.Error.CalibrationError*)	0x8029002F	An error occurred while performing calibration, verify your wiring
`UEIDAQ_INVALID_CLOCK_SIGNAL_ERROR` (*UeiDaq.Error.InvalidClockSignal*)	0x80290030	The specified clock signal is not compatible with the device associated to this session
`UEIDAQ_CHANNEL_IN_USE_ERROR` (*UeiDaq.Error.ChannelAlreadyInUse*)	0x80290031	The channel is already used by another session
`UEIDAQ_CAN_COMM_ERROR` (*UeiDaq.Error.CANCommError*)	0x80290032	The CAN bus switched to bus off state. There is a faulty node or cable
`UEIDAQ_OFFSET_NULLING_ERROR` (*UeiDaq.Error.OffsetNulling*)	0x80290033	The offset is too high to be nulled
`UEIDAQ_UNEXPECTED_RESULT_ERROR` (*UeiDaq.Error.UnexpectedResult*)	0x80290034	Unexpected result
`UEIDAQ_PARITY_ERROR` (*UeiDaq.Error.Parity*)	0x80290035	A parity error has been detected
`UEIDAQ_FRAMING_ERROR` (*UeiDaq.Error.Framing*)	0x80290036	A framing error has been detected
`UEIDAQ_NOT_SUPPORTED_ERROR` (*UeiDaq.Error.NotSupported*)	0x80290037	Operation is not supported on this sub-system or device
`UEIDAQ_BUFFER_TOO_SMALL_ERROR` (*UeiDaq.Error.BufferTooSmall*)	0x80290038	The buffer is too small to hold all received data
`UEIDAQ_DEVICE_LOCKED_ERROR` (*UeiDaq.Error.Locked*)	0x80290039	Device is already locked by another session
`UEIDAQ_GPS_ANTENNA_ERROR` (*UeiDaq.Error.GPSAntenna*)	0x8029003A	GPS antenna is disconnected

C/C++ Constant (*.NET Constant*)	Code	Description
`UEIDAQ_GPS_LOCK_ERROR` (*UeiDaq.Error.GPSLock*)	0x8029003B	Could not lock on GPS signal
`UEIDAQ_DATA_NOT_READY_WARNING` (*UeiDaq.Error.DataNotReady*)	0x8029003C	Device reports that no data is ready yet
`UEIDAQ_IOMODULE_REBOOTED_ERROR` (*UeiDaq.Error.Rebooted*)	0x8029003D	IO module was rebooted
`UEIDAQ_PPS_LOCK_ERROR` (*UeiDaq.Error.PPSLock*)	0x8029003E	Could not lock on 1PPS signal

Appendix B: Custom Properties

Some PowerDNA devices have features that are not directly accessible through the standard framework API.
It might be because those features are unique, and we didn't want to bloat the Framework's API or because they are so new that we haven't had a chance to integrate them in the API yet.

Custom Properties are used to operate those features; they can be accessed using the Session object's methods "SetCustomProperty" and "GetCustomProperty".

A custom property has a unique name and a value whose type must be an array of integer or floating point elements.

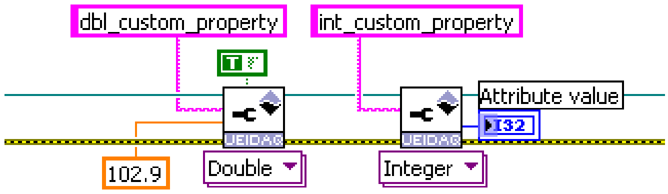
Here is an example showing how to access custom properties in C++:

```
// Set the floating point property "dbl_property_name" to 102.90
double fValue = 102.90;
MySession.SetCustomProperty("dbl_property_name", sizeof(double),
&fValue);

// Get the value of the integer property "int_property_name"
int iValue[64];
MySession.GetCustomProperty("int_property_name", 64*sizeof(int),
iValue);
```

Here is an example showing how to access custom properties in ANSI-C:

```
// Set the floating point property "dbl_property_name" to 102.90
double fValue = 102.90;
UeiDaqSetSessionCustomProperty(sessionHandle, "dbl_property_name",
&fValue, sizeof(double));

// Get the value of the string property "int_property_name"
int iValue[64];
UeiDaqGetSessionCustomProperty(sessionHandle, "int_property_name",
iValue, 64*sizeof(char));
```

You can also access custom properties from LabVIEW using the polymorphic VI "UeiDaqCustomAttribute.vi". This VI has four instance VIs to access scalar or arrays of integer or floating point values.



B-1 PowerDNA AI-205 custom properties:

Each AI-205 analog input channel is equipped with a three-stage FIR filter and decimators.

FIR filters are disabled by default. You can enable and control FIR filters using the following custom properties (each property must be written in the sequence described below):

- “channel”: An integer representing the channel for which you want to configure the FIR filter.
- “stage”: An integer set to 0, 1 or 2 representing the stage to configure for the selected channel.
- “decimation”: An integer representing the decimation for the selected stage.
- “tap”: An array of floating-point values representing the taps for the selected stage. The maximum number of taps is 128.

To disable FIR filters, set decimation to 1 and program one tap with coefficient “1.0” on each stage.

Note that setting a decimation value greater than 1 will slow down the rate at which your application will receive data from the AI-205: you need to adjust the session timeout parameter accordingly.

The following sample code shows how to program the first stage of the FIR filter on channel 0:

```
int firChannel = 0;
int firStage = 0;
int decimation = 1;
double taps[8]= {…};
MySession.SetCustomProperty(“channel”, sizeof(int), &firChannel);
MySession.SetCustomProperty(“stage”, sizeof(int), &firStage);
MySession.SetCustomProperty(“decimation”, sizeof(int), &decimation);
MySession.SetCustomProperty(“tap”, 8*sizeof(double), taps);
```

B-2 PowerDNA AO-302/308 custom properties:

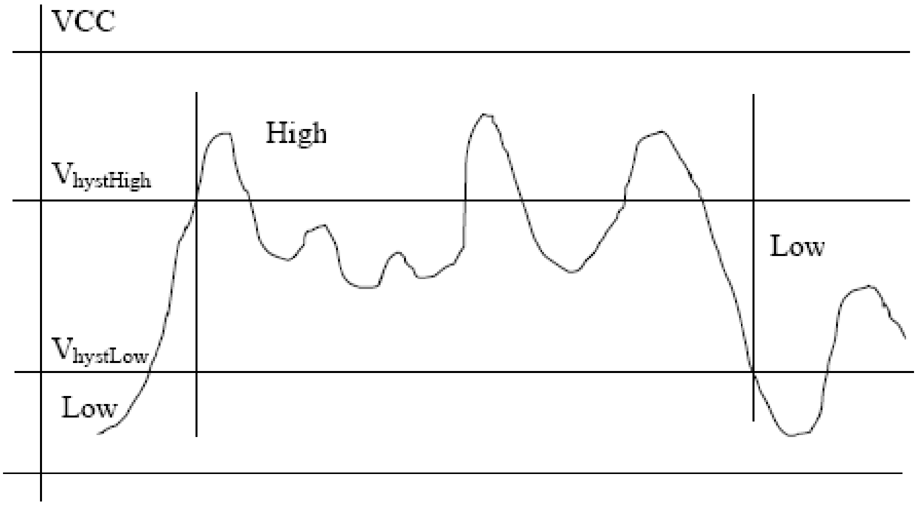
It is possible to program the initial level of the AO-302 analog output channels after powering-up the PowerDNA and the levels before shutting it down.

- “startupvalues”: An array of 8 floating-point elements representing the level on each analog output channel when the cube is starting-up.
- “shutdownvalues”: An array of 8 floating-point elements representing the level on each analog output channel when the cube is shutting down.

B-3 PowerDNA DIO-401/402/404/405/406 custom properties:

The PowerDNA DIO-40x layers are equipped with a hysteresis circuitry whose low and high threshold levels can be programmed using custom properties.

- “lowhysteresis”:A floating-point value representing the low hysteresis voltage as a percentage of the power supply voltage (Vcc).
- “highhysteresis”: A floating-point value representing the high hysteresis voltage as a percentage of the power supply voltage (Vcc).


VCC
High
VhystHigh
Low
VhystLow
Low