CEBO
Programming Reference

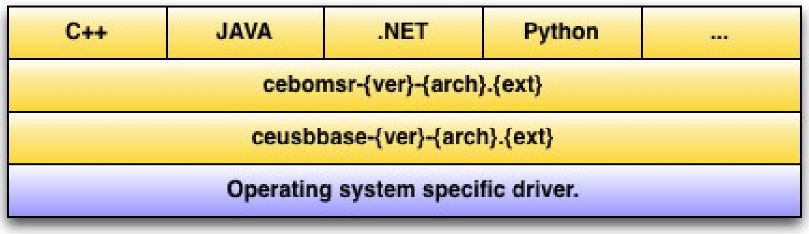
This document describes .NET, C++, Java, and Python interfaces for the Cebo-MSR API (Cebo-LC and Cebo-Stick).
It contains general information, examples and library interfaces for each of the supported languages.

Programming Interface

The following chapters describe the CeboMsr API in all supported programming languages. All language sections contain examples that show different usages of the API, starting by very basic things up to more complex features. Every language section is finished using a reference of all components.

API Stack

The API consists 3 different layers. From the API-user perspective, only the top layer must be known. To get a better understanding, the image below shows a simplified diagram of the stack.


C++
JAVA
.NET
Python
...
cebomsr-{ver}-{arch}.{ext}
ceusbbase-{ver}-{arch}.{ext}
Operating system specific driver.

The lowest layer varies from system to system and offers a unique USB interface to the upper components.
The component in the middle is responsible to offer a unique, flexible interface to different device types, using the lowest layer for device communication.
The different implementations on top of the stack are just thin layers offering the functionality of the middle layer in their respective language. The API's are designed to be easy and intuitive to use, as well as best tailored for the language in question.
Never try to build upon the middle layer directly, this interface is permanently subject to change, so any update may break compatibility.

Thread safety

All API calls are internally locked by an API managed mutex, so the API can be used in any multi-threaded context without additional locking mechanisms. This means, if one thread enters one function of the CeboMsr API, all other threads using the interface must wait until the first thread returns from its call.
In conclusion, blocking calls prevent other threads from calling API functions, which can be an unwanted feature. To prevent this, the following must be bear in mind:

The only long call is **readBlocking()** of class **Device**, which lasts until the requested amount of data is read from the device. In this case, method **readNonBlocking()** should be used instead.
Besides that, it's always a good idea to use one device in one single thread only.

Application flow

The application flow is quite easy:

The first task is to call the **enumerate()** method of class **LibraryInterface**. This returns a list of possible devices, each instanced as class **Device**. Only devices that are not already opened are reported.

Before working with one of the devices, method **open()** must be called. This can be done with every reported device.

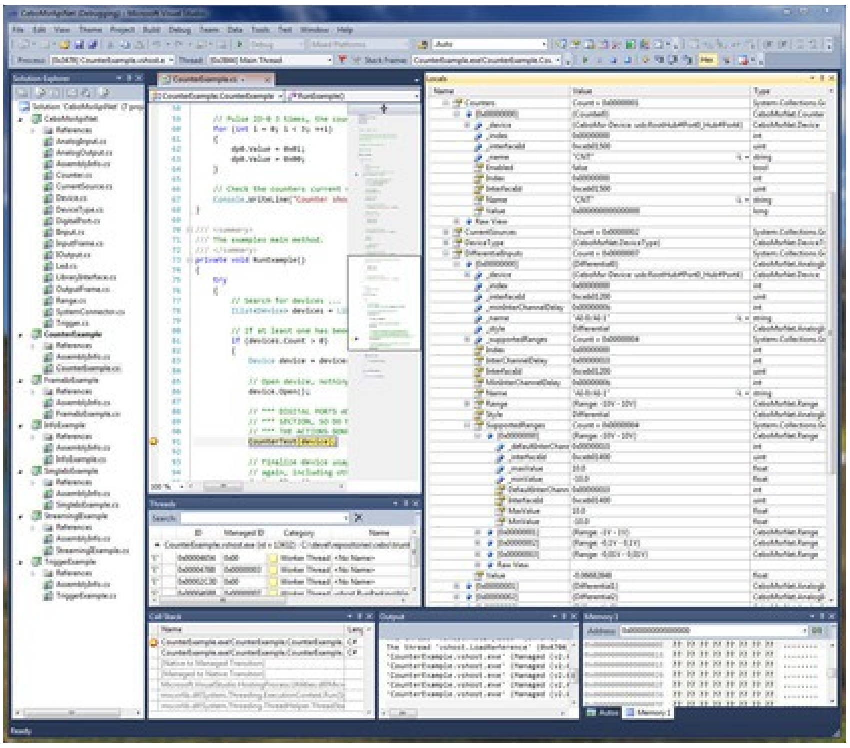
Devices can now be used in any way. If one of the method calls report an error, it may be necessary to close and maybe reconnect the device (e.g. communication error). But this highly depends on the error.

If the device is not needed anymore, method **close()** must be called. Subsequent calls to **enumerate()** report the closed device again from now one (except it is disconnected).

.NET

The .NET API is written in C# and is a thin layer on top of the CeboMsr API. It uses Platform Invoke to use the dynamic link library.

Compatibility



The API is designed so it can be compiled for framework 2.0. Newer frameworks should work as well. The complete source of the API is distributed as part of the API and can be build for different frameworks.
The API is available to any .NET compatible language.

Usage

Simply add **cebomsrnet-{ver}.dll** as reference to your project. This works for both 32 and 64 bit applications. Be sure to have **cebomsr-{ver}-{arch}.dll** (and its dependencies) in the runtime path when starting the application, otherwise an exception is thrown.

Design Difference

In comparison to the other language interfaces, the .NET API uses properties where possible, while the other API's use get/set methods. One of the main reasons for this decision was, using properties, you can inspect the values during a debug session:

Error Handling

Any logical or runtime error that is reported by the CeboMsr API is transformed to an exception. The following table lists all possible exception thrown by the .NET implementation of the CeboMsr API:

Exception	**Circumstances**
ArgumentException	If any of the used parameters has an invalid value in the current context.
IndexOutOfRangeException	When using an invalid index. (Most of these are prevented due to the API design).
InvalidOperationException	Thrown if something is called which is not allowed in this stage.
IOException	Indicates an unexpected behavior like communication problems with the device.

As seen in the table above, all exceptions but IOException are unchecked exceptions, so they are predictable at development time. In the case of IOException, human investigation may be required to clarify the reason of the problem.
All exceptions contain a textual description of the problem.

Basics

This section describes the first steps that are necessary to use the API. We start by showing a basic example that:

- searches for all known devices
- grabs the first available device
- opens the device
- resets its state
- closes the device

```
using CeboMsrNet;

...

// Search for devices ...
IList<Device> devices = LibraryInterface.Enumerate(DeviceType.All);

// If at least one has been found, use the first one ...
if (devices.Count > 0)
{
  Device device = devices[0];

  // Open device, nothing can be done without doing this.
  device.Open();

  ...

  // After some processing, the device can be resetted to its
  // power up condition.
  device.ResetDevice();

  ...

  // Finalize device usage, this free's up the device, so it can be used
  // again, including other applications.
  device.Close();
}
```

Description

The first lines in the example import the complete namespace to the source unit.

In the application flow, enumeration must be always the first task when using the API. It searches the system for known, not yet used devices and offers the caller a list of device instances. The parameter that is used calling Enumerate() of LibraryInterface specifies which devices should be searched:

- All known devices (DeviceType.All)
- Device classes (e.g. DeviceType.Usb)
- Specific types (e.g. DeviceType.CeboLC)
-

We search for all devices in the example above. The returned list contains candidates of type Device that can be used subsequently.

We check if **Count** of the list is greater than zero to see if at least one device has been found. If a device has been found, the if branch is entered, were we continue by using the first device.

The call to Open() flags the device for use. Any new enumeration (before calling Close() on the same instance) will filter this instance. This is just a simple mechanism to prevent double access to the same device.

After the device has been opened, it can be used in any way. Method ResetDevice() is shown here, because it has a superior role. It stops all active processing inside the device and resets all configuration values to their startup condition. It is not necessary to call it, but is a good way to reach a defined state i.e. in an error condition. It is implicitly invoked when calling Open().

At the end of the current scope, method Close() is called to signal API and system that the device is not used anymore. This is an important call, because leaving it in opened state will prevent it from getting enumerated again.

Single Value I/O

This section describes simple single input and output of values. It continues the example from section Basics, so an opened device instance is available. Single value I/O is described in detail in the CEBO-LC manual (data acquisition, single value I/O).

```
// Read digital port #0 and write result to digital port #1 (mirror pins).
// At first, we need references to both ports.
DigitalPort dp0 = device.DigitalPorts[0];
DigitalPort dp1 = device.DigitalPorts[1];

// Configure port first, all bits of digital port #0 as input (default)
// and all bits of digital port #1 as output.
dp1.OutputEnableMask = 0xff;

// Read from #0 ...
int value = dp0.Value;

// Write the value to #1 ...
dp1.Value = value;

// Now some analog I/O, do it without any additional local references.
// Read single ended #0 input voltage ...
float voltageValue = device.SingleEndedInputs[0].Value;

// Write value to analog output #1
device.AnalogOutputs[1].Value = voltageValue;
```

Description

If you want work with device peripherals, get instances using the various component properties of class Device. If more then one method must be called, create a local reference, this reduces the amount of code to write. After you have an instance, invocations to its methods affects this specific peripheral.

The example outlines how to read all I/O's of digital port #0 and mirror the result on port #1. The first requirement to accomplish this is to define the direction of the individual I/O's. By default, all I/O's are configured to be inputs, so the example shows how to set all I/O's on port #1 as output, modifying OutputEnableMask on its reference. All bits that are '1' in the specified mask signal the I/O to be an output. This is all you have to do for configuration.

The mirroring step is quite easy, read the Value from the instance that represents port #0 and store it in a local integer. The result is subsequently used writing it as Value on the instance the represents port #1.

The second half of the example outlines how to access peripherals without local copies or references. This requires fewer but longer lines of code but has no behavioral difference. Choose which style you prefer on your own.

The example continues doing direct single value I/O using analog peripherals. Like the digital port part, an input is mirrored to an output.

The Value read from the input is the calibrated voltage value. Assigning it to the Value property of the AnalogOutput instance that represents analog output #1 will directly modify the real output on the device.

Single Frame Input

This section presents an example that samples a selection of inputs in a single process. The difference to single value read is not that much on the coding side, but there's a large improvement when it comes to performance. Reading more than one input costs only an insignificant higher amount of time in comparison to single I/O.
Configuration of the individual peripherals is exactly the same, so setting the I/O direction if digital ports or set the input range on an analog input is almost identical.

```
// Prepare and fill the list of inputs to read.
IInput[] inputs = new IInput[]
{
  device.SingleEndedInputs[0],
  device.SingleEndedInputs[1],
  device.DifferentialInputs[1],
  device.DigitalPorts[0],
  device.DigitalPorts[1],
  device.Counters[0]
};

// Setup device with this selection.
device.SetupInputFrame(inputs);

// Read the values multiple times and write them to the console.
for (int i = 0; i < 100; ++i)
{
  // Read all inputs into the instance of InputFrame.
  InputFrame inFrame = device.ReadFrame();

  // Write results to the console.
  Console.WriteLine(
    "DigitalPort #0: " + inFrame.GetDigitalPort(0) + ", " +
    "DigitalPort #1: " + inFrame.GetDigitalPort(1) + ", " +
    "SingleEnded #0: " + inFrame.GetSingleEnded(0) + " V, " +
    "SingleEnded #1: " + inFrame.GetSingleEnded(1) + " V, " +
    "Differential #1: " + inFrame.GetDifferential(1) + " V, " +
    "Counter #0: " + inFrame.GetCounter(0));
}
```

Description

The primary work for frame input I/O is to setup the device with the list of inputs that are involved. An array which contains the inputs to sample is required.
The example does this in the upper part. It creates the array and add several inputs to it:

- Single ended analog input #0 and #1
- Differential analog input #1
- Digital ports #0 and #1
- Counter #0

This list is than used calling SetupInputFrame(), which prepares the device with this setup. This is active until:

- A call to ResetDevice()
- The device is closed
- Or a new list is specified

In the subsequent loop, the specified inputs are sampled in every loop cycle using method ReadFrame(). The call lasts until all inputs are sampled and returns the values immediately in an instance of class InputFrame.
This instance holds all sampled values and offers methods to request them, which is shown in the lower part of the example.

Single Frame Output

In this section, concurrent modification of outputs will be outlined. In comparison to single value I/O, this technique is very efficient when working with more than one single output. As you can see in the following example, using it is quite straightforward.

```
// Write to analog out #1 and digital out #2 in one call.

// Prepare and set output selection.
IOutput[] outputs = new IOutput[]
{
  device.DigitalPorts[2],
  device.AnalogOutputs[1]
};

// Prepare device ...
device.SetupOutputFrame(outputs);

// Create instance of OutputFrame. There's no direct construction, as this
// instance may vary between different device types.
OutputFrame outFrame = device.CreateOutputFrame();

// Write it to hardware, modify contents in every looping.
for (int i = 0; i < 100; ++i)
{
  outFrame.SetAnalogOutput(1, (float)(3 * Math.Sin((float)i / 100.0f)));
  outFrame.SetDigitalPort(2, i);

  device.WriteFrame(outFrame);
}
```

Description

First of all, the example assumes that an opened instance of class Device is available, as well as that DigitalPort #2 is configured to be output.

Similar to the other direction, the device must be set up with a list of outputs. This setup is active until:

- A call to ResetDevice()
- The device is closed
- Or a new list is set up

An array that contains the outputs to set is required. In the upper part of the example, you can see the creation of this array, adding outputs DigitalPort #2 and AnalogOutput #1 to it and activate the setup using SetupOutputFrame().

The subsequent call to CreateOutputFrame() creates an instance of type OutputFrame, which fits to the device metrics. There's no other way to create this type.

In the loop below, every cycle modifies the values of the outputs specified during setup. This does not do anything other than storing the values inside the frame. The active modification of the outputs is done using WriteFrame(), which transfers the values to the respective peripherals.

Multi Frame Input

This section describes how to read many frames at once, a very useful feature when you want sample inputs at a constant frequency or using an external trigger. The API offers very handy methods. The most problematic thing is to choose the right start method including its parameters.
The example below samples five different inputs at a constant rate of 300 Hz, 20 x 25 frames, completely cached inside the device buffer.

```
// Construct selection that contains inputs to read from.
IInput[] inputs = new IInput[] {
  device.SingleEndedInputs[0],
  device.SingleEndedInputs[1],
  device.DifferentialInputs[1],
  device.DigitalPorts[0],
  device.DigitalPorts[1]
};

// Prepare device with this collection ...
device.SetupInputFrame(inputs);

// Start sampling ...
device.StartBufferedDataAcquisition(300, 20 * 25, false);

// Read 20 x 25 frames using blocked read,
// this function returns after *all* (25) requested frames are collected.
for (int i = 0; i < 20; ++i)
{
  // Read 25 frames ...
  IList<InputFrame> frames = device.ReadBlocking(25);

  // Write out the 1st one.
  InputFrame inFrame = frames[0];
  Console.WriteLine(
    "DigitalPort #0: " + inFrame.GetDigitalPort(0) + ", " +
    "DigitalPort #1: " + inFrame.GetDigitalPort(1) + ", " +
    "SingleEnded #0: " + inFrame.GetSingleEnded(0) + " V, " +
    "SingleEnded #1: " + inFrame.GetSingleEnded(1) + " V, " +
    "Differential #1: " + inFrame.GetDifferential(1) + " V");
}
```

```
// Stop the DAQ.
device.StopDataAcquisition();
```

Description

The first lines in this example are similar to single frame example. The device is set up to sample the specified inputs into a single call. Single ended input #0 and #1, differential input #1 and digital ports #0 and #1 are used here.

The real data acquisition is than started calling StartBufferedDataAcquition(). Choosing this method to start up, combined with the used parameters means the following:

- Data has to be completely stored in the device memory (buffered).
- Sampling is done using hardware timed capture at 300 Hz.
- The number of frames to capture is 20 x 25 = 500.
- No external trigger is necessary to initiate the process.

This method does not only configure the DAQ process, but start it as well. In the case of buffered mode, this is really uncritical. If you require continuous mode, a buffer overrun may occur shortly after starting the DAQ, so it is very important to either:

- Start to read the sampled frames immediately, as shown in the example.
- Use a second thread to read the frames, start this thread before the DAQ is started.

The example continues with a for loop where every cycle reads 25 frames and outputs the sampled values of the first frame in this block. The used method ReadBlocking() returns after the given amount of frames has been read and stalls the thread up to this point. Use ReadBlocking() with care, because it has two downsides:

- It blocks the access to the whole API due to internal thread locking mechanisms.
- If the specified amount of frames is too small, especially at higher frame rates, buffer overruns on the device side will occur, as the call and transfer overhead is too high.

It is a very convenient way to read a defined number of frames. The alternative is the use ReadNonBlocking(), which returns immediately with all captured frames at the moment of calling.

Both read methods return the list of captured frames. The example uses the first frame of this block (which is guaranteed to contain 25 frames in the example), and output the sampled values of all specified inputs to the console.

At the end of the example, DAQ is stopped using StopDataAcquisition().

Counter

The example below shows how to use a counter. To allow this using software, a wired connection between IO-0 and CNT is necessary. IO-0 is used to generate the events that the counter should count.

```
// Create local copies, this shortens the source.
DigitalPort dp0 = device.DigitalPorts[0];
Counter cnt = device.Counters[0];

// Set IO-0 as output.
dp0.OutputEnableMask = 0x01;

// Enable the counter.
cnt.Enabled = true;

// Check the counters current value.
Console.WriteLine("Counter before starting: " + cnt.Value);

// Pulse IO-0 3 times.
for (int i = 0; i < 3; ++i)
{
  dp0.Value = 0x01;
  dp0.Value = 0x00;
}

// Check the counters current value.
Console.WriteLine("Counter should be 3: " + cnt.Value);

// Reset and ...
cnt.Reset();

// ... disable the counter.
cnt.Enabled = false;

// Check the counters current value.
Console.WriteLine("Counter should be 0: " + cnt.Value);

// Pulse IO-0 3 times, the counter should ignore these pulses.
for (int i = 0; i < 3; ++i)
{
  dp0.Value = 0x01;
  dp0.Value = 0x00;
}

// Check the counters current value.
Console.WriteLine("Counter should still be 0: " + cnt.Value);
```

Description

Like most of the previous examples, an instance of class Device is required. This can be retrieved using the procedure described here. To demonstrate the counter without external peripherals, the software controls the events for the counter as well. This extends the example to around twice its size, but its still easy to understand.

To reduce the amount of code, dp0 and cnt are constructed as local copies of both the first DigitalPort and the Counter.

The LSB of digital port #0 represents IO-0. To modify IO-0 via software, the example continues to set this I/O as output using property OutputEnableMask.

Counters are disabled by default, so the next required task is to enable it, otherwise no event will be detected. The example does this by setting property Enabled of the Counter instance to **true**.

The first counter value that is printed out will be zero, which is the default value after startup or reset.

The counter reacts on every rising edge. The example shows this by pulsing IO-0 three times, setting its level high and than low in every loop cycle. The result is than printed to the console. It is expected to be three, based on the pulses in the previous loop (If not, verify the wired connection between IO-0 and CNT).

The value of the counter is than set to zero calling its Reset() method. In addition, setting Enabled to false deactivates the counter to any event.

The remaining example code outlines this, by pulsing IO-0 three times again, but the console output shows that now flanks have been counted in this state.

Trigger

The following example is much longer than the previous, but outlines various new things:

- Working with multiple devices.
- Use different multi frame modes.
- Show how to use triggers in input and output mode.

You will need two devices for this example to run, as both devices are chained together using triggers. The first device acts as master, while the second device is the slave. Every time, the master samples a frame, an output trigger is generated, while the slave captures its frame if this trigger has been raised.

Devices must be connected to each other using two wires. Ground (GND) to ground and trigger (TRG) to trigger.

TIP #1: This example can easily be extended to support more than one slave, you only have to set up each slave the same way as the slave in the example.

TIP #2: At high bandwidth, it may be necessary to put the individual readNonBlocking() calls to separate threads. Otherwise reading the data from one device may last as long as the device side buffer on the second device will need to overflow.

The description is located below the example.

```
class TriggerExample
{
  /// <summary>
  /// Write frames to console.
  /// </summary>
  /// <param name="device">Device string.</param>
  /// <param name="frames">Frame to dump.</param>
  private void DumpFrames(string device, IList<InputFrame> frames)
  {
    foreach (InputFrame f in frames)
    {
      Console.WriteLine(device + ": " +
        "se#0: " + f.GetSingleEnded(0) + " V, " +
        "dp#0: " + f.GetDigitalPort(0));
    }
```

```
}

/// <summary>
/// Start DAQ, read frames and output the results.
/// </summary>
/// <param name="master">Master device.</param>
/// <param name="slave">Slave device.</param>
private void RunDataAcquisition(Device master, Device slave)
{
  // The slave must be started first, as it reacts on the master's
  // trigger. Continuous DAQ is used. Timed using an external trigger.
  slave.StartContinuousExternalTimedDataAcquisition();

  // The master uses hardware timed DAQ, continuous at 50 Hz.
  // The 'false' here signals that the process should start immediately.
  master.StartContinuousDataAcquisition(50, false);

  // The example reads at least 10 frames from
  // both devices and subsequently output the samples.
  int masterFrames = 0, slaveFrames = 0;
  while (masterFrames < 10 || slaveFrames < 10)
  {
    // Start by reading frames from master,
    // output it and increment counter.
    IList<InputFrame> frames = master.ReadNonBlocking();
    DumpFrames("master", frames);
    masterFrames += frames.Count;

    // Do the same with the slave.
    frames = slave.ReadNonBlocking();
    DumpFrames("slave", frames);
    slaveFrames += frames.Count;

    // Don't poll to frequent, this would fully utilize one core.
    Thread.Sleep(1);
  }

  // Finished, gracefully stop DAQ.
  slave.StopDataAcquisition();
  master.StopDataAcquisition();
}

/// <summary>
/// Both devices are fully configured here.
/// </summary>
/// <param name="master">Master device.</param>
/// <param name="slave">Slave device.</param>
private void Configure(Device master, Device slave)
{
  // The trigger for the master must be set to alternating output.
  master.Triggers[0].Config = Trigger.TriggerConfig.OutputAlternating;
```

```
    // The slave's trigger must be set to alternating as well.
    slave.Triggers[0].Config = Trigger.TriggerConfig.InputAlternating;

    // Both devices now gets configured to the same input frame layout.
    master.SetupInputFrame(new IInput[] {
      master.SingleEndedInputs[0],
      master.DigitalPorts[0]
    });

    // ... slave
    slave.SetupInputFrame(new IInput[] {
      slave.SingleEndedInputs[0],
      slave.DigitalPorts[0]
    });
  }

  /// <summary>
  /// The examples main method.
  /// </summary>
  private void RunExample()
  {
    try
    {
      // Search for the devices, exactly two are required.
      IList<Device> devices = LibraryInterface.Enumerate(DeviceType.All);
      if (2 != devices.Count)
      {
        Console.WriteLine("Exactly two devices are required.");
        return;
      }

      // As both devices can act as master,
      // we simply use the first one for this role.
      Device master = devices[0];
      master.Open();
      Console.WriteLine("Master: " + master.SerialNumber +
          "@" + master.Identifier);

      // ... and the second as slave.
      Device slave = devices[1];
      slave.Open();
      Console.WriteLine("Slave: " + slave.SerialNumber +
          "@" + slave.Identifier);

      // Configure both master and slave ...
      Configure(master, slave);

      // ... and do the DAQ.
      RunDataAcquisition(master, slave);

      // Close both.
      master.Close();
      slave.Close();
    }
    catch (IOException ex)
    {
      Console.WriteLine(ex.ToString());
    }
  }

  static void Main(string[] args)
  {
    new TriggerExample().RunExample();
  }
}
```

Description

Read the example bottom up, starting in the **RunExample()** method. Nothing really new in comparison to the previous examples, except that two devices are used concurrently. Both devices gets opened and a short information about master and slave is printed to the console.

What follows is the configuration, which is shown in method **Configure()**. Trigger #0 of the master device is set to output, alternating. This means, every time, the master captures a frame, its first trigger toggles the level.

Trigger #0 of the slave is configured to work as input, alternating as well. So the slave captures a frame if its first trigger detects that the level has been toggled.

That's all for the trigger configuration. Method **Configure()** completes the process by setting up a frame using single ended input #0 and digital port #0 as described in detail here.

Returned to **RunExample()**, method **RunDataAcquisition()** is invoked, which shows how data from both master and slave is read. The most important part here is, how both DAQ processes are started. The slave is started first, otherwise he may miss one or more events. The slave is set to sample frames every time a trigger has been detected, without any count limits. This is done calling StartContinuousExternalTimedDataAcquisition().

The master's DAQ is than started calling StartContinuousDataAcquisition(), which means, no frame count limitation at a specific frequency. The example uses a very low frequency, 50 Hz, and starts immediately (By using **false** for parameter **externalStarted**).

At runtime, the master will than start to sample frames at the given 50 Hz, toggle its Trigger every time, while the slave captures a frame every time this event is received at his input trigger pin. Both capture frames synchronously.

The example continues by reading frames from both devices one after another until both have returned at least 10 frames. Captured values are written to the console using method **DumpFrames()**. The loop contains a 1 ms sleep, otherwise the usage of one CPU core would go to 100%, which is never a good idea.

After the frames have been read, DAQ is stopped on both devices, calling StopDataAcquisition(). The program returns to **RunExample()** and Close() both devices.

Info

This section simply outlines what information you can get from the API about the API itself, the device and its peripherals.

```
// Put out some device specific information.
Console.WriteLine("Device type: " + device.DeviceType.Name);
Console.WriteLine("USB base ver: " + LibraryInterface.UsbBaseVersion);
Console.WriteLine("API vers: " + LibraryInterface.ApiVersion);
Console.WriteLine("Firmware ver: " + device.FirmwareVersion);
Console.WriteLine("Identifier: " + device.Identifier);
Console.WriteLine("Serial number: " + device.SerialNumber);
Console.WriteLine("Temperature: " + device.Temperature);

// Get the real current of the reference current sources.
foreach (CurrentSource source in device.CurrentSources)
{
  Console.WriteLine("Reference current of "
    + source.Name + ": "
    + source.ReferenceCurrent + " uA");
}

// Retrieve information about single ended input #0.
// This works on all analog inputs and outputs.
AnalogInput se0 = device.SingleEndedInputs[0];
```

```
Console.WriteLine("Info for single ended input " + se0.Name + ":");
Console.WriteLine("Min. ICD: " + se0.MinInterChannelDelay + " us");
Console.WriteLine("  Current range: "
  + se0.Range.MinValue + " V to "
  + se0.Range.MaxValue + " V");
Console.WriteLine("Supported ranges:");
for (int i = 0; i < se0.SupportedRanges.Count; ++i)
{
  Range range = se0.SupportedRanges[i];
  Console.WriteLine("    Range #" + i + " range: "
    + range.MinValue + " V to "
    + range.MaxValue + " V, "
    + "Def. ICD: "
    + se0.GetDefaultInterChannelDelay(range) + " us");
}

// Get info for digital port #1.
DigitalPort dp1 = device.DigitalPorts[1];
Console.WriteLine("Info for digital port " + dp1.Name + ":");
Console.WriteLine("  Count of I/O's: " + dp1.IoCount);
for (int i = 0; i < dp1.IoCount; ++i)
{
  Console.WriteLine("    I/O #" + i + ":" + dp1.GetIoName(i));
}
```

Description

As most of the previous examples, this assumes an opened device as well. How to do this is described here.

The block in the upper part of the example writes any API or device specific information to the console. The printed information is self-explanatory.

The first loop iterates over all CurrentSources of the connected device. For every source, its real current value is printed out. These values were determined during device calibration.

In the following block, information about single ended input #0 is printed out, which is the active range setting, as well as all ranges that are valid for this port. All ranges report their lower and upper bound using properties MinValue and MaxValue. This can be done for every AnalogInput and AnalogOutput.

The last part of the example prints information about digital port #1. The value retrieved from IoCount is the number of I/O's that can be accessed by this DigitalPort. This may vary between the individual ports a device has. The names of all its single I/O's are printed as well.

Class Reference

The class reference is a complete but short overview of all classes and their methods used in the .NET API. It does not outline how the components have to be used.
The sections parallel to this topic show many practical examples and should be the first you have to read to understand and use the API. A good starting point is this topic.

Interfaces

Both interfaces below are used to group input- and output peripherals.

interface IInput
Used to group peripherals which can act as input.

interface IOutput
Used to group peripherals which can act as output.

DeviceType

This class is an enumeration of device types and device classes. Its static instances can be used to control the enumeration process. Besides that, each device reports its class using this type by property DeviceType.

static readonly DeviceType All
Includes all known devices, independent which bus type is used. In the current stage, this equals the following instance, Usb.

static readonly DeviceType Usb
By using this instance in the enumeration process, all known devices connected via USB are reported.

static readonly DeviceType CeboLC
This instance must be specified if CEBO LC devices should be searched. In addition, property DeviceType of Device returns this if the device is of this type.

static readonly DeviceType CeboStick
This instance must be specified if CEBO STICK devices should be searched. In addition, getDeviceType() of Device returns this if the device is of this type.

string Name { get; }
This property is the name, e.g. "CeboLC" for a CeboLC instance.

AnalogInput

This class is the host side interface to any analog input, so you have to use its methods to request or modify the peripheral that this instance is assigned to. Get an instance of this class from properties SingleEndedInputs or DifferentialInputs of the corresponding Device instance.

IList<Range> SupportedRanges { get; }
This property lists the supported Ranges this input supports. You can use each of these calling SetParameters().

int GetDefaultInterChannelDelay(Range range)
Return the default interchannel delay (ug105-cebo-lc.pdf) at the specified Range in microseconds.

int MinInterChannelDelay { get; }
The minimal interchannel delay (ug105-cebo-lc.pdf) for this input in microseconds.

void SetParameters(Range range)
Sets the Range level on the analog input. Overwrites any previously adjusted interchannel delay (ug105-cebo-lc.pdf) with the default value.

int SetParameters(Range range, int interChannelDelay)
Set Range for this input. In addition, the interchannel delay (ug105-cebo-lc.pdf) in

microseconds is adjusted manually. The returned value is the interchannel delay that is really used (as not all specified values can be handled by the hardware).

Range Range { get; set; }
This property can be used to define or read the active Range setting. Setting it will be similar calling SetParameters(Range).

int InterChannelDelay { get; }
This property can be used to read the active interchannel delay(ug105-cebo-lc.pdf) .

float Value { get; }
The voltage value read from input directly (ug105-cebo-lc.pdf) .

string Name { get; }
Name of the input.

AnalogOutput

This class represents an analog output. You can get an instance of this class from property AnalogOutputs from the corresponding Device.

IList<Range> SupportedRanges { get; }
Returns the list of Ranges this input supports. You can use each of these calling SetParameters().

void SetParameters(Range range)
Sets the Range on the analog output.

Range Range { get; set; }
This property can be used to define or read the active Range setting. Setting it will be similar calling SetParameters(Range).

float Value { set; }
Assigning a value to this property sets the voltage on the output directly (ug105-cebo-lc.pdf) .

string Name { get; }
Name of the output.

DigitalPort

This class is the interface to work with digital ports. Retrieve instances from DigitalPorts of the respective Device.

int OutputEnableMask { set; }
Set bitwise mask that defines which of the I/O's on the specified port are input and output. A bit of value 1 defines the specific I/O as output, e.g. mask = 0x03 means that I/O 0 and 1 are set to output, while I/O 2 to n are inputs.

int IoCount { get; }
The count of I/O's of the specific port.

int Value { get; set; }
Set output I/O's or read them. Only the bits that have been defined as output using OutputEnableMask are modified during write.

string Name { get; }
Name of the port.

string GetIoName(int io)
Returns the name of the I/O as specified by parameter io. The range of io is 0 <= io < IoCount.

Counter

Interface class to counters inside the device. Class instances can be retrieved from property Counters of the specific Device.

void Reset()
Reset the counter to value 0.

bool Enabled { get; set; }

Define or request the counters enabled state.

CounterConfig Config { get; set; }
Define or retrieve the current counter configuration.

enum CounterConfig

- RisingEdge: Counter event is rising edge.
- FallingEdge: Counter event is falling edge.
- Alternating: Counter event are both rising and falling edges.

long Value { get; }
Get the current value of the counter.

string Name { get; }
Name of the counter.

Trigger

Class that represents a trigger inside the device. Instances can be retrieved by from Triggers of the respective Device.

bool Enabled { get; set; }
Define or request the triggers enabled state.

TriggerConfig Config { get; set; }
Define or retrieve the current trigger configuration.

enum TriggerConfig

- OutputPulse: Trigger is output. Every trigger-event generates a positive pulse
- OutputAlternating: Trigger is output. Every trigger-event toggles the level
- InputRisingEdge: Trigger is input, reacts on a rising edge
- InputFallingEdge: Trigger is input, reacts on falling edge
- InputAlternating: Trigger is input, reacts on rising and falling edge.

string Name { get; }

Name of the trigger.

Range

Use when handling range settings. Valid setting for an AnalogInput or AnalogOutput can be retrieved from their SupportedRanges properties.

float MinValue { get; }
The lower voltage of the specific range.

float MaxValue { get; }
The upper voltage of the specific range.

Led

Interface to the LED's on the board. Instances are accessed from Leds of the corresponding Device.

bool Enabled { set; }
Enable or disable the LED by setting this property.

string Name { get; }
Name of the LED.

CurrentSource

Class that represents the interface to the Fixed Current Outputs. Instances can be retrieved from CurrentSources of the respective Device.

float ReferenceCurrent { get; }
Returns the actual value of the Fixed Current Output, which is determined during manufacturing process and stored in onboard flash. The returned value is given in micro ampere.

string Name { get; }
Name of the current source.

InputFrame

The input frame is a data class which stores measured samples from all inputs a device has (and which meet the frame concept (ug105-cebo-lc.pdf). All samples inside a single frame are captured in a very short time span (which depends on the underlying device). Only values that have been selected using SetupInputFrame() before sampling the frame are valid, the other are set to 0 by default.

float GetSingleEnded(int index)
Return the voltage value of single ended analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

float GetDifferential(int index)
Return the voltage value of differential analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

int GetDigitalPort(int index)
Return the I/O state of the digital port indicated by the given **index** at the moment the frame has been sampled.

bool GetTrigger(int index)
Return the state of the Trigger as specified by the given **index** in the moment the frame has been sampled.

long GetCounter(int index)
Return the value of the Counter as specified by the given **index**.

OutputFrame

This class stores all values the should be set to the outputs as specified using SetupOutputFrame(). Calling one of its methods does not affect the hardware. This is done when calling WriteFrame(). An OutputFrame instance can't be created directly, as its metrics depends on the underlying device. The instance can retrieved calling CreateOutputFrame() of the device instance that should be used.

void SetDigitalPort(int index, int value)
Set the bit mask for the DigitalPort as specified using **index**. Only bits set to output using OutputEnableMask are affected.

void SetAnalogOutput(int index, float value)
Define the voltage value that should be set to the AnalogOutput as specified by the given **index**. The value will be calibrated before it is active.

Device

This is the primary class of the framework. Each instance corresponds to one physical device. The class is not intended to be instanced directly. Use the method Enumerate() of the class LibraryInterface to access instances. Many of the methods are related to the device itself. Besides that, instances to all device peripherals can be accessed.

void Open()
This method must be called before doing anything with the device. Internally a communication to the device is constructed and several initialization is done. Various device and firmware specific constants are read and method ResetDevice() is called. To guarantee compatibility between API and device, if the major number of the device firmware is higher than the major number of the API, open() will fail, a newer version of the API must be used with this device.

void Close()
This closes the connection to the device, frees all internal resources and allows the device to be used by others (in the application instance and in different applications). Calling close will never throw an exception, it will always fail silently.

void ResetDevice()
Calling this will stop any hardware controlled processing and reset the device to its power up settings. This is usually necessary if errors occur (Except logic or communication errors). Invoking this method in a multithreaded context should be done with extra care.

void ResetPeripherals(int mask)

Resets specific peripherals on the device side. Parameter mask specifies which elements using a bit mask. The bit mask must be constructed using the following flags, which will be enhanced in future versions:

int&nsbp;Device.FlagResetInputFifo	Clears the FIFO used during data acquisition and is error flagged

string Identifier { get; }
This returns a unique identifier to the device which is constructed using the physical connection properties. It is guaranteed to be unique at runtime, as well as constant between reboots of the operating system (except updates to the operating system change the behavior how physical properties are enumerated).

Be aware: Plugging a device to a different location (e.g. different USB port) will change this specifier, it is not device dependent. Use SerialNumber in this case.
This can be requested in front of invoking Open().

DeviceType DeviceType { get; }
The specific type of the device that is bound to the instance. This is one of class DeviceType static members.
This can be requested before invoking Open().

string FirmwareVersion { get; }
The firmware version encoded as string, e.g. "1.0".

string SerialNumber { get; }
The device serial number as string. This string is unique for each device.

float Temperature { get; }
Current temperature in °C of the device.

uint WatchdogTimeout
Sets or reads the watchdog timeout in 250ms steps. If enabled, the device reboots if no data transfer has been done in the given time frame. Default value is 0xffffffff, which disables this feature.

int CalculateMaxBufferedInputFrames()
Depending on the specified inputs the device site buffer can store a limited amount of frames. This method calculates the count of frames that will fit into this buffer using the current frame setup, which is last set using setupInputFrame(). Modifying the frame setup invalidates this value and must be updated invoking this method again. This value is primary intended to be used in context with startBufferedDataAcquisition().

void SetupInputFrame(IEnumerable<IInput> inputs)
When doing any form of frame based data acquisition, which is described in the user guide CBO-LC, this method must be used to select the inputs to be read. There's no limit which inputs can be selected, but every input can only be specified once. This method can only be called if no multi frame based data acquisition (ug105-cebo-lc.pdf) is active.
Any subsequent frame based call is affected by this setup.
The input array can contain any instance that implements IInput, AnalogInput, DigitalPort, Counter or Trigger.

float StartBufferedDataAcquisition(float frameRate, int frameCount, bool externalStarted)
Starts a buffered data acquisition at the specified **frameRate** for exactly frameCount frames. If **externalStarted** is true, the acquisition is started at the moment an external trigger event has been detected, immediately otherwise. Parameter **frameRate** must be equal or smaller than the value reported by CalculateMaxBufferedInputFrames(). The returned value is the frame rate that is really used, as not every possible frame rate can be handled by the device. This value is as near as possible to the specified frame rate. Detailed description for this can be found in the user guide CEBO-LC (ug105-cebo-lc.pdf).

void StartBufferedExternalTimedDataAcquisition(int frameCount)
Very similar to the method above, except that no frame rate is used the sample frame, but every time an external trigger has been detected one single frame is sampled. Acquisition is automatically stopped after the specified amount of frames has been sampled. Detailed description for this can be found in the user guide CEBO-LC (ug105-cebo-lc.pdf).

float StartContinuousDataAcquisition(float frameRate, bool externalStarted)

Starts a data acquisition without frame limit. The host must read the sampled data as fast as possible, otherwise buffer overflow is signaled and the process has been failed. Frames are sampled at the specified **frameRate**. If **externalStarted** is true, sampling starts at the first detected trigger event, immediately otherwise. The return value is the frame rate that is really used, as not every possible rates are possible.
In a multithreaded context, it is advised to start reading frames before calling this method.
More details about data acquisition can be found in in the user guide CEBO-LC (ug105-cebo-lc.pdf).

void StartContinuousExternalTimedDataAcquisition()
Similar to the method above, but instead of a fixed frame rate, frames are sampled for every detected trigger. Detailed information about this can be found in the user guide CEBO-LC (ug105-cebo-lc.pdf).

void StopDataAcquisition()
Stops a currently active data acquisition. The device buffer is not modified calling this, so unread data can be fetched subsequently.

IList<InputFrame> ReadBlocking(int frameCount)
When doing multi frame data acquisition (more: ug105-cebo-lc.pdf), this is one method to read the sampled frames from device to host. This version will block the current thread until the specified amount of frames has been read, there's no timeout. After the call returns without an exception, the returned list contains exactly the specified amount of InputFrames. The alternative to this call is described below.

IList<InputFrame> ReadNonBlocking()
This method is similar to the one above, except that it always returns immediately, while trying to read as much frames as possible. The returned list contains all sampled InputFrames since the start of the data acquisition or the last frame read. Especially for high data amounts, this method should be called cyclically without to high delays.

InputFrame ReadFrame()
Read single frame as set up calling SetupInputFrame() and return immediately. This cannot be called if multi frame data acquisition (ug105-cebo-lc.pdf) is active.

void SetupOutputFrame(IEnumerable<IOutput> outputs)
Specifies which Outputs are involved in the next frame output process. There's no limit of the specified outputs, but no single instance can be used more than once. Valid outputs that can be added to the list are AnalogOutput and DigitalPort.

void WriteFrame(OutputFrame frame)
Set all outputs that have been selected using SetupOutputFrame() in a single call. Data to set on the respective outputs must be set in the given OutputFrame. The frame must be constructed using method CreateOutputFrame() of the same device instance were it is used.

OutputFrame CreateOutputFrame()
Create an OutputFrame instance that fits to the device metrics from which instance the method is called. The frame can than be filled with data and used calling WriteFrame() of the same device instance.

IList<AnalogInput> SingleEndedInputs { get; }
List of all single ended inputs.

IList<AnalogInput> DifferentialInputs { get; }
List of differential analog inputs.

IList<AnalogOutput> AnalogOutputs { get; }
List of the analog outputs.

IList<DigitalPort> DigitalPorts { get; }
List of digital ports.

IList<Counter> Counters { get; }
List of counters.

IList<Trigger> Triggers { get; }
List of triggers.

IList<CurrentSource> CurrentSources { get; }
List of current sources.

IList<Led> Leds { get; }
List of LED's.

LibraryInterface

This class contains static functionality only. Its responsibility is to serve as interface to methods that are not bound to any device.

string ApiVersion { get; }
Report the version number of the underlying CeboMsr API string, i.e. "1.0".

string UsbBaseVersion { get; }
The version number of the system interface USB layer string, i.e. "1.0".

IList<Device> Enumerate(DeviceType type)
The specified **type** parameter must be one of class DeviceType static members. The system is than scanned for known devices and subsequently filtered to meet the specified **type**. Devices that are already opened were skipped too. The returned list contains all candidates that are ready for use.
For each instance that should be used, Open() must be called.
Starting a new enumeration invalidates the list from the previous invocation.

C++

The C++ API is an ultra-thin layer on top of the CeboMsr API. It is distributed as a pair of files (**cebomsrpp.cpp**, **cebomsrpp.h**), which contain all functionality of the API mapped to the C++ language.

To use the API in a project, add both files as well as the C library header file (**cebomsr.h**) to it. Depending on the used compiler, library files may be necessary as well (check the compatibility topic below).

How the API itself is used is described chapter “basics”.

Compatibility

The API has been tested on the compilers in the table below. It is expected that newer versions of these compilers are compatible as well.
As the API is designed to only use standard language elements as well as only uses the standard library, there is a good chance that older compilers can be used too, but they are not officially supported.

OS	Compiler	Library files	Version/ Exp. Release
Windows*1	MS VC ++ 2005 MS VC ++ 2008 MS VC ++ 2010	cebomsr-{V}-{A}.lib	1.0
Windows*1	MinGW G ++ 4.4	cebomsr-{V}-x86.dll	1.0
Linux	G ++ 4.4	-	Oct 2012
Mac OS X Lion*2	G ++ 4.4	-	Oct/ Nov 2012

{V} - Version number of the API, e.g. 1.0.
{A} - Architecture, depends on the target executable, -x86 for 32 bit, -x64 for 64 bit.
*1 - Windows® is a trademark of Microsoft Corporation.
*2 - Mac OS X Lion is a trademark of Apple Inc.

Namespace

All components are located in namespace **CeboMsr**.

Error handling

Any logical or runtime error that is reported by the CeboMsr API is transformed to an exception. Only exceptions defined by the C++ standard library are used. The following table lists all possible exceptions thrown by the C++ implementation of the CeboMsr API:

Exception	Circumstances
std::invalid_argument	If any of the used parameters has an invalid value in the current context.
std::out_of_range	When using an invalid index. (Most of these are prevented due to the API design).
std::logic_error	Thrown if something is called which is not allowed in this stage.
std::runtime_error	Indicates an unexpected behavior like communication problems with the device.

As seen in the table above, all exceptions but std::runtime_error are predictable during development. In the case of std::runtime_error, human investigation may be required to clarify the reason of the problem.
All exception objects contain a textual description of the problem, which can be retrieved by calling the what() method.

Basics

This section describes the first steps that are necessary to use the API. We start by showing a basic example that:

- includes the required header file
- searches for all known devices
- grabs the first available device
- opens the device
- resets its state
- closes the device

```
#include <cebomsrpp.h>

...

using namespace CeboMsr;

...
```

```
// Search for devices ...
DeviceVector devices = LibraryInterface::enumerate(DeviceType::All);

// If at least one has been found …
if (!devices.empty()) {

  // Use the first one ...
  Device device = devices[0];

  // Open device, nothing can be done without doing this.
  device.open();

  ...

  // After some processing, the device can be resetted to its
  // power up condition.
  device.resetDevice();

  ...

  // Finalize device usage, this free's up the device,
  // so it can be used again, including other applications.
  device.close();
}
```

Description

The first lines in the example above include the C++ API to the active source document. The using namespace directive brings the CeboMsr API to global scope, so i.e. **class Device** can be referenced as "**Device**" instead of "**CeboMsr::Device**".

In the application flow, enumeration must be always the first task when using the API. It searches the system for known, not yet used devices and offers the caller a list of device instances. The parameter that is used
calling enumerate() of LibraryInterface specifies which devices should be searched:

- All known devices (DeviceType::All)
- Device classes (e.g. DeviceType::Usb)
- Specific types (e.g. DeviceType::CeboLC)

We search for all devices in the example above. The returned DeviceVector contains candidates of type Device that can be used subsequently.

As DeviceVector is just an alias to std::vector<Device>, all methods from std::vector<>

can be used. For those who are not familiar with the standard library, **empty()** checks if the list contains at least one element and return false otherwise. If a device has been found, the if branch is entered, were we create a local copy of the first device. This instance must be used subsequently.

The call to open() flags the device for use. Any new enumeration (before calling close() on the same instance) will filter this instance. This is just a simple mechanism to prevent double access to the same device.
After the device has been opened, it can be used in any way. Method resetDevice() is shown here, because it has a superior role. It stops all active processing inside the device and resets all configuration values to their startup condition. It is not necessary to call it, but is a good way to reach a defined state i.e. in an error condition. It is implicitly invoked when calling open().

At the end of the current scope, method close() is called to signal API and system that the device is not used anymore. This is an important call, because leaving it in opened state will prevent it from getting enumerated again.

Single Value I/O

This section describes simple single input and output of values. It continues the example from section Basics, so an opened device instance is available. Single value I/O is described in the user guide CEBO-LC (ug105-cebo-lc.pdf) in detail.

```
// Read digital port #0 and write result to digital port #1 (mirror pins).
// At first, we need references to both ports.
// Port #0 will be a copy of the original instance.
DigitalPort dp0 = device.getDigitalPorts().at(0);

// As an alternative, get a reference to Port #1.
const DigitalPort &dp1 = device.getDigitalPorts().at(1);

// Configure port first, all bits of digital port #0 as input (default)
// and all bits of digital port #1 as output.
dp1.setOutputEnableMask(0xff);

// Read from #0 ...
int value = dp0.read();

// Write the value to #1 ...
dp1.write(value);
```

```
// Now some analog I/O, do it without any additional local references.
// Read single ended #0 input voltage ...
float voltageValue = device.getSingleEndedInputs().at(0).read();

// Write value to analog output #1
device.getAnalogOutputs().at(1).write(voltageValue);
```

Description

If you want work with device peripherals, get instances using the various component access methods of class Device. If more then one method must be called, either create a local copy or assign it to a local reference, this reduces the amount of code to write. Both is shown in the example above accessing DigitalPort #0 and #1. After you have an instance, invocations to its methods affects this specific peripheral.

The example outlines how to read all I/O's of digital port #0 and mirror the result on port #1. The first requirement to accomplish this is to define the direction of the individual I/O's. By default, all I/O's are configured to be inputs, so the example shows how to set all I/O's on port #1 as output, calling setOutputEnableMask() on its reference. All bits that are '1' in the specified mask signal the I/O to be an output. This is all you have to do for configuration.

The mirroring step is quite easy, read() the value from the instance that represents port #0 and store it in a local integer. The result is subsequently used calling write() on the instance the represents port #1.

The second half of the example outlines how to access peripherals without local copies or references. This requires fewer but longer lines of code but has no behavioral difference. Choose which style you prefer on your own.

The example continues doing direct single value I/O using analog peripherals. Like the digital port part, an input is mirrored to an output.

The call to read() returns the calibrated voltage value on this input as float.
Calling write() on the AnalogOutput instance that represents analog output #1 using the returned voltage value will directly modify the real output on the device.

Single Frame Input

This section presents an example that samples a selection of inputs in a single process. The difference to single value read is not that much on the coding side, but there's a large improvement when it comes to performance. Reading more than one input costs only an insignificant higher amount of time in comparison to single I/O.
Configuration of the individual peripherals is exactly the same, so setting the I/O direction if digital ports or set the input range on an analog input is almost identical.

```
// Prepare and fill the list of inputs to read.
InputVector inputs;
inputs
  << device.getSingleEndedInputs().at(0)
  << device.getSingleEndedInputs().at(1)
  << device.getDifferentialInputs().at(1)
  << device.getDigitalPorts().at(0)
  << device.getDigitalPorts().at(1)
  << device.getCounters().at(0);

// Setup device with this selection.
device.setupInputFrame(inputs);

// Read the values multiple times and write them to the console.
for (int i = 0; i < 100; ++i) {
  // Read all inputs into the instance of InputFrame.
  InputFrame inFrame = device.readFrame();

  // Write results to the console.
  cout
    << "DigitalPort #0: " << inFrame.getDigitalPort(0) << ", "
    << "DigitalPort #1: " << inFrame.getDigitalPort(1) << ", "
    << "SingleEnded #0: " << inFrame.getSingleEnded(0) << " V, "
    << "SingleEnded #1: " << inFrame.getSingleEnded(1) << " V, "
    << "Differential #1: " << inFrame.getDifferential(1) << " V, "
    << "Counter #0: " << inFrame.getCounter(0) << endl;
}
```

Description

The primary work for frame input I/O is to setup the device with the list of inputs that are involved. The C++ API offers the type InputVector, which is just an alias to std::vector<Input>. To simplify the list creation, **operator<<** has been implemented to present a clean way to add inputs to this vector.

The example does this in the upper part. It creates a local instance of type InputVector and add several inputs to it:

- Single ended analog input #0 and #1
- Differential analog input #1
- Digital ports #0 and #1
- Counter #0

This list is than used calling setupInputFrame(), which prepares the device with this setup. This is active until:

- A call to resetDevice()
- The device is closed
- Or a new list is specified

In the subsequent loop, the specified inputs are sampled in every loop cycle using method readFrame(). The call lasts until all inputs are sampled and returns the values immediately in an instance of class InputFrame.
This instance holds all sampled values and offers methods to request them, which is shown in the lower part of the example.

Single Frame Output

In this section, concurrent modification of outputs will be outlined. In comparison to single value I/O, this technique is very efficient when working with more than one single output. As you can see in the following example, using it is quite straightforward.

```
// Write to analog out #1 and digital out #2 in one call.
// Prepare and set output selection.
OutputVector outputs;
outputs
  << device.getDigitalPorts().at(2)
  << device.getAnalogOutputs().at(1);

// Prepare device ...
device.setupOutputFrame(outputs);

// Create instance of OutputFrame. There's no direct construction, as this
// instance may vary between different device types.
OutputFrame outFrame = device.createOutputFrame();

// Write it to hardware, modify contents in every looping.
```

```
for (int i = 0; i < 100; ++i) {
  outFrame.setAnalogOutput(1, (float)(3 * sin((float)i / 100.f)));
  outFrame.setDigitalPort(2, i);

  device.writeFrame(outFrame);
}
```

Description

First of all, the example assumes that an opened instance of class Device is available, as well as that DigitalPort #2 is configured to be output.
Similar to the other direction, the device must be set up with a list of outputs. This setup is active until:

- A call to resetDevice()
- The device is closed
- Or a new list is set up

In the C++ API, OutputVector alias std::vector<Output> is used as list type. The API implements **operator<<** for this type, so the list can be filled easily.

In the upper part of the example, you can see the creation of this list, adding outputs DigitalPort #2 and AnalogOutput #1 to it and activate the setup using setupOutputFrame().

The subsequent call to createOutputFrame() creates an instance of type OutputFrame, which fits to the device metrics. There's no other way to create this type.

In the loop below, every cycle modifies the values of the outputs specified during setup. This does not do anything other than storing the values inside the frame. The active modification of the outputs is done using writeFrame(), which transfers the values to the respective peripherals.

Multi Frame Input

This section describes how to read many frames at once, a very useful feature when you want sample inputs at a constant frequency or using an external trigger. The API offers very handy methods. The most problematic thing is to choose the right start method including its parameters.

The example below samples five different inputs at a constant rate of 300 Hz, 20 x 25 frames, completely cached inside the device buffer.

```
// Construct selection that contains inputs to read from.
InputVector inputs;
inputs
  << device.getSingleEndedInputs().at(0)
  << device.getSingleEndedInputs().at(1)
  << device.getDifferentialInputs().at(1)
  << device.getDigitalPorts().at(0)
  << device.getDigitalPorts().at(1);

// Prepare device with this collection ...
device.setupInputFrame(inputs);

// Start sampling ...
device.startBufferedDataAcquisition(300, 20 * 25, false);

// Read 20 x 25 frames using blocked read,
// this function returns after *all* (25) requested frames are collected.
for (int i = 0; i < 20; ++i) {
  // Read 25 frames ...
  InputFrameVector frames = device.readBlocking(25);

  // Write out the 1st one.
  const InputFrame &inFrame = frames[0];
  cout
    << "DigitalPort #0: " << inFrame.getDigitalPort(0) << ", "
    << "DigitalPort #1: " << inFrame.getDigitalPort(1) << ", "
    << "SingleEnded #0: " << inFrame.getSingleEnded(0) << " V, "
    << "SingleEnded #1: " << inFrame.getSingleEnded(1) << " V, "
    << "Differential #1: " << inFrame.getDifferential(1) << " V"
    << endl;
}

// Stop the DAQ.
device.stopDataAcquisition();
```

Description

The first lines in this example are similar to single frame example. The device is set up to sample the specified inputs into a single call. Single ended input #0 and #1, differential input #1 and digital ports #0 and #1 are used here.
The real data acquisition is than started calling startBufferedDataAcquition(). Choosing this method to start up, combined with the used parameters means the following:
Data has to be completely stored in the device memory (buffered).

Sampling is done using hardware timed capture at 300 Hz.
The number of frames to capture is 20 x 25 = 500.
No external trigger is necessary to initiate the process.
This method does not only configure the DAQ process, but start it as well. In the case of buffered mode, this is really uncritical. If you require continuous mode, a buffer overrun may occur shortly after starting the DAQ, so it is very important to either:
Start to read the sampled frames immediately, as shown in the example.
Use a second thread to read the frames, start this thread **before** the DAQ is started.
The example continues with a for loop where every cycle reads 25 frames and outputs the sampled values of the first frame in this block. The used method readBlocking() returns after the given amount of frames has been read and stalls the thread up to this point. Use readBlocking() with care, because it has two downsides:
It blocks the access to the whole API due to internal thread locking mechanisms.
If the specified amount of frames is too small, especially at higher frame rates, buffer overruns on the device side will occur, as the call and transfer overhead is too high.
It is a very convenient way to read a defined number of frames. The alternative is the use readNonBlocking(), which returns immediately with all captured frames at the moment of calling.
Both read methods return the list of captured frames as type InputFrameVector, which is an alias to std::vector<InputFrame>. The example uses the first frame of this block (which is guaranteed to contain 25 frames in the example), and output the sampled values of all specified inputs to the console.
At the end of the example, DAQ is stopped using stopDataAcquisition().

Counter

The example below shows how to use a counter. To allow this using software, a wired connection between IO-0 and CNT is necessary. IO-0 is used to generate the events that the counter should count.

```
// Create local copies, this shortens the source.
DigitalPort dp0 = device.getDigitalPorts().at(0);
Counter cnt = device.getCounters().at(0);

// Set IO-0 as output.
dp0.setOutputEnableMask(0x01);
```

```
// Enable the counter.
cnt.setEnabled(true);

// Check the counters current value.
cout << "Counter before starting: " << cnt.read() << endl;

// Pulse IO-0 3 times.
for (int i = 0; i < 3; ++i) {
  dp0.write(0x01);
  dp0.write(0x00);
}

// Check the counters current value.
cout << "Counter should be 3: " << cnt.read() << endl;

// Reset and ...
cnt.reset();

// ... disable the counter.
cnt.setEnabled(false);

// Check the counters current value.
cout << "Counter should be 0: " << cnt.read() << endl;

// Pulse IO-0 3 times, the counter should ignore these pulses.
for (int i = 0; i < 3; ++i) {
  dp0.write(0x01);
  dp0.write(0x00);
}

// Check the counters current value.
cout << "Counter should still be 0: " << cnt.read() << endl;
```

Description

Like most of the previous examples, an instance of class Device is required. This can be retrieved using the procedure described in the programming reference C++. To demonstrate the counter without external peripherals, the software controls the events for the counter as well. This extends the example to around twice its size, but its still easy to understand.

To reduce the amount of code, **dp0** and **cnt** are constructed as local copies of both the first DigitalPort and the Counter.

The LSB of digital port #0 represents IO-0. To modify IO-0 via software, the example continues to set this I/O as output using method setOutputEnableMask().

Counters are disabled by default, so the next required task is to enable it, otherwise no event will be detected. The example does this by calling setEnabled() of the Counter instance.

The first counter value that is printed out will be zero, which is the default value after startup or reset.

The counter reacts on every rising edge. The example shows this by pulsing IO-0 three times, setting its level high and than low in every loop cycle. The result is than printed to the console. It is expected to be three, based on the pulses in the previous loop (If not, verify the wired connection between IO-0 and CNT).

The value of the counter is than set to zero calling its reset() method. In addition, setEnabled() deactivates the counter to any event.
The remaining example code outlines this, by pulsing IO-0 three times again, but the console output shows that now flanks have been counted in this state.

Trigger

The following example is much longer than the previous, but outlines various new things:

- Working with multiple devices.
- Use different multi frame modes.
- Show how to use triggers in input and output mode.

You will need two devices for this example to run, as both devices are chained together using triggers. The first device acts as master, while the second device is the slave. Every time, the master samples a frame, an output trigger is generated, while the slave captures its frame if this trigger has been raised.

Devices must be connected to each other using two wires. Ground (GND) to ground and trigger (TRG) to trigger.

TIP #1: This example can easily be extended to support more than one slave, you only have to set up each slave the same way as the slave in the example.

TIP #2: At high bandwidth, it may be necessary to put the individual readNonBlocking() calls to separate threads. Otherwise reading the data from one device may last as long as the device side buffer on the second device will need to overflow.

The description is located below the example.

```
// Write frames to console.
void dumpFrames(const string &device, const InputFrameVector &frames) {
  for (size_t i = 0; i < frames.size(); ++i) {
    cout << device << ": "
        << "se#0: " << frames.at(i).getSingleEnded(0) << " V, "
        << "dp#0: " << frames.at(i).getDigitalPort(0) << endl;
  }
}

// Start DAQ, read frames and output the results.
void runDataAcquisition(Device &master, Device &slave) {
  // The slave must be started first, as it reacts on the master's trigger.
  // Continuous DAQ is used. Timed using an external trigger.
  slave.startContinuousExternalTimedDataAcquisition();

  // The master uses hardware timed DAQ, continuous at 50 Hz.
  // The 'false' here signals that the process should start immediately.
  master.startContinuousDataAcquisition(50, false);

  // The example reads at least 10 frames from
  // both devices and subsequently output the samples.
  int masterFrames = 0, slaveFrames = 0;
  while (masterFrames < 10 || slaveFrames < 10) {
    // Start by reading frames from master,
    // output it and increment counter.
    InputFrameVector frames = master.readNonBlocking();
    dumpFrames("master", frames);
    masterFrames += (int)frames.size();

    // Do the same with the slave.
    frames = slave.readNonBlocking();
    dumpFrames("slave", frames);
    slaveFrames += (int)frames.size();

    // Don't poll to frequent, this would fully utilize one core.
    cesleep(1);
  }

  // Finished, gracefully stop DAQ.
  slave.stopDataAcquisition();
  master.stopDataAcquisition();
```

```
}

// Both devices are fully configured here.
void configure(Device &master, Device &slave) {
  // The trigger for the master must be set to alternating output.
  master.getTriggers().at(0).setConfig(Trigger::OutputAlternating);

  // The slave's trigger must be set to alternating as well.
  slave.getTriggers().at(0).setConfig(Trigger::InputAlternating);

  // Both devices now gets configured to the same input frame layout.
  InputVector inputs;
  inputs
    << master.getSingleEndedInputs().at(0)
    << master.getDigitalPorts().at(0);
  master.setupInputFrame(inputs);

  // Use the same list, clear it before use.
  inputs.clear();
  inputs
    << slave.getSingleEndedInputs().at(0)
    << slave.getDigitalPorts().at(0);
  slave.setupInputFrame(inputs);
}

// Application entry point.
int main() {
  try {
    // Search for the devices, exactly two are required.
    DeviceVector devices = LibraryInterface::enumerate(DeviceType::All);
    if (2 != devices.size()) {
      cout << "Exactly two devices are required." << endl;
      return 1;
    }

    // As both devices can act as master,
    // we simply use the first one for this role.
    Device master = devices[0];
    master.open();
    cout << "Master: " << master.getSerialNumber()
        << "@" << master.getIdentifier() << endl;

    // ... and the second as slave.
    Device slave = devices[1];
    slave.open();
    cout << "Slave: " << slave.getSerialNumber()
        << "@" << slave.getIdentifier() << endl;

    // Configure both master and slave ...
    configure(master, slave);

    // ... and do the DAQ.
    runDataAcquisition(master, slave);

    // Close both.
    master.close();
    slave.close();
  } catch (std::exception &ex) {
    cout << "Exception: " << ex.what() << endl;
  }
  return 0;
}
```

Description

Read the example bottom up, starting in the **main()** function. Nothing really new in comparison to the previous examples, except that two devices are used concurrently. Both devices gets opened and a short information about master and slave is printed to the console.

What follows is the configuration, which is shown in function **configure()**. Trigger #0 of the master device is set to output, alternating. This means, every time, the master captures a frame, its first trigger toggles the level.

Trigger #0 (more: programming reference C++) of the slave is configured to work as input, alternating as well. So the slave captures a frame if its first trigger detects that the level has been toggled.
That's all for the trigger configuration. Function **configure()** completes the process by setting up a frame using single ended input #0 and digital port #0 as described in detail here.

Returned to **main()**, function **runDataAcquisition()** is invoked, which shows how data from both master and slave is read. The most important part here is, how both DAQ processes are started. The slave is started first, otherwise he may miss one or more events. The slave is set to sample frames every time a trigger has been detected, without any count limits. This is done calling startContinuousExternalTimedDataAcquisition().
The master's DAQ is than started calling startContinuousDataAcquisition(), which means, no frame count limitation at a specific frequency. The example uses a very low frequency, 50 Hz, and starts immediately (By using **false** for parameter **externalStarted**).

At runtime, the master will than start to sample frames at the given 50 Hz, toggle its Trigger every time, while the slave captures a frame every time this event is received at his input trigger pin. Both capture frames synchronously.

The example continues by reading frames from both devices one after another until both have returned at least 10 frames. Captured values are written to the console using function **dumpFrames()**. The loop contains a 1 ms sleep, otherwise the usage of one CPU core would go to 100%, which is never a good idea.

After the frames have been read, DAQ is stopped on both devices, calling stopDataAcquisition(). The program returns to **main()** and close() both devices.

Info

This section simply outlines what information you can get from the API about the API itself, the device and its peripherals.

```
// Put out some device specific information.
cout << "Device type: " << device.getDeviceType().getName() << endl;
cout << "USB base ver: " << LibraryInterface::getUsbBaseVersion() << endl;
cout << "API vers: " << LibraryInterface::getApiVersion() << endl;
cout << "Firmware ver: " << device.getFirmwareVersion() << endl;
cout << "Identifier: " << device.getIdentifier() << endl;
cout << "Serial number: " << device.getSerialNumber() << endl;
cout << "Temperature: " << device.getTemperature() << endl;

// Get the real current of the reference current sources.
for (size_t i = 0; i < device.getCurrentSources().size(); ++i) {
  CurrentSource source = device.getCurrentSources().at(i);
  cout << "Reference current of "
    << source.getName() << ": "
    << source.getReferenceCurrent() << " uA" << endl;
}

// Retrieve information about single ended input #0.
// This works on all analog inputs and outputs.
AnalogInput se0 = device.getSingleEndedInputs().at(0);
cout << "Info for single ended input " << se0.getName() << ":" << endl;
cout << "  Current range: "
  << se0.getRange().getMinValue() << " V to "
  << se0.getRange().getMaxValue() << " V" << endl;
cout << "  Supported ranges:" << endl;
for (size_t i = 0; i < se0.getSupportedRanges().size(); ++i) {
  Range range = se0.getSupportedRanges().at(i);
  cout << "    Range #" << i << " range: "
    << range.getMinValue() << " V to "
    << range.getMaxValue() << " V, "
    << "def. icd: "
    << se0.getDefaultInterChannelDelay(range) << " us"
    << endl;
}

// Get info for digital port #1.
DigitalPort dp1 = device.getDigitalPorts().at(1);
cout << "Info for digital port " << dp1.getName() << ":" << endl;
cout << "  Count of I/O's: " << dp1.getIoCount() << endl;
for (int i = 0; i < dp1.getIoCount(); ++i) {
  cout << "    I/O #" << i << ":" << dp1.getIoName(i) << endl;
}
```

Description

As most of the previous examples, this assumes an opened device as well. How to do this is described in the programming reference C++.

The block in the upper part of the example writes any API or device specific information to the console. The printed information is self-explanatory.
The first loop iterates over all CurrentSources of the connected device. For every source, its real current value is printed out. These values were determined during device calibration.

In the following block, information about single ended input #0 is printed out, which is the active range setting, as well as all ranges that are valid for this port. All ranges report their lower and upper bound using methods getMinValue() and getMaxValue(). This can be done for every AnalogInput and AnalogOutput.

The last part of the example prints information about digital port #1. The value retrieved by getIoCount() is the number of I/O's that can be accessed by this DigitalPort. This may vary between the individual ports a device has. The names of all its single I/O's are printed as well.

Class Reference

The class reference is a complete but short overview of all classes and their methods used in the C++ API. It does not outline how the components have to be used.

The sections parallel to this topic show many practical examples and should be the first you have to read to understand and use the API. A good starting point is topic “Basics” in the programming reference C++.

Simple Types

Simple types are not classes that offer any functionality, but types that are just aliases or empty base classes used to group elements.
These types are C++ specific and not all have similar counterparts in different language implementations of the API.

Input

Used to group peripherals which can act as input.

Output

Used to group peripherals that can act as output.

InputVector

Alias for std::vector<Input>. To use this alias in a convenient way, operator<< has been overloaded inside the API. So a list of inputs can be created as shown below:

```
// Build selection of
// - Single Ended Analog Input #0
// - Digital Port #1
// - Counter #0
InputVector inputs;
inputs
  << device.getSingleEndedInputs().at(0)
  << device.getDigitalPorts().at(1)
  << device.getCounters().at(0);
```

OutputVector

Alias for std::vector<Output>. To fill a list in a convenient way, operator<< has been overloaded, so an instance can be filled in the same way as show above in the example of inputs.

RangeVector

Alias for std::vector<Range>.

AnalogInputVector
Alias to std::vector<AnalogInput>.

AnalogOutputVector
Alias to std::vector<AnalogOutput>.

DigitalPortVector
Alias to std::vector<DigitalPort>.

CounterVector
Alias to std::vector<Counter>.

TriggerVector
Alias to std::vector<Trigger>.

LedVector
Alias to std::vector<Led>.

CurrentSourceVector
Alias to std::vector<CurrentSource>.

InputFrameVector
Alias for std::vector<InputFrame>.

DeviceVector
Alias for std::vector<Device>.

DeviceType

This class is an enumeration of device types and device classes. Its static instances can be used to control the enumeration process. Besides that, each device reports its class using this type in method getDeviceType().

static DeviceType All
Includes all known devices, independent which bus type is used. In the current stage, this equals the following instance, Usb.

static DeviceType Usb
By using this instance in the enumeration process, all known devices connected via USB are reported.

static DeviceType CeboLC
This instance must be specified if CEBO LC devices should be searched. In addition, getDeviceType() of Device returns this if the device is of this type.

static DeviceType CeboStick
This instance must be specified if CEBO STICK devices should be searched. In addition, getDeviceType() of Device returns this if the device is of this type.

std::string getName()
Returns the name of the instance as std::string, e.g. "CeboLC" for the CeboLC instance.

AnalogInput

This class is the host side interface to any analog input, so you have to use its methods to request or modify the peripheral that this instance is assigned to.
Get an instance of this class by calling either getSingleEndedInputs() or getDifferentialInputs() of the corresponding Device instance.

const RangeVector &getSupportedRanges() const
Returns the list of Ranges this input supports. You can use each of these calling setParameters().

int getDefaultInterChannelDelay(const Range &range) const
Return the default interchannel delay (more: user guide CEBO-LC, Data acquisition, interchannel delay) at the specified Range in microseconds.

int getMinInterChannelDelay() const
Return the minimal interchannel delay more: user guide CEBO-LC, Data acquisition,

interchannel delay) for this input in microseconds.

void setParameters(const Range &range) const
Sets the Range level on the analog input. Overwrites any previously adjusted interchannel delay (more: user guide CEBO-LC, Data acquisition, interchannel delay) with the default value.

int setParameters(const Range &range, int interChannelDelay) const
Set Range for this input. In addition, the interchannel delay (more: user guide CEBO-LC, Data acquisition, interchannel delay) in microseconds is adjusted manually. The returned value is the interchannel delay that is really used (as not all specified values can be handled by the hardware).

Range getRange() const
Returns active range setting.

int getInterChannelDelay() const
Returns active interchannel delay (more: user guide CEBO-LC, Data acquisition, interchannel delay).

float read() const
Returns voltage value from input directly (more: user guide CEBO-LC, Data acquisition, single value I/O).

std::string getName() const
Returns the name of the input.

AnalogOutput

This class represents an analog output. You can get an instance of this class calling getAnalogOutputs() from the corresponding Device.

const RangeVector &getSupportedRanges() const
Returns the list of Ranges this input supports. You can use each of these calling setParameters().

void setParameters(const Range &range) const
Sets the Range on the analog output.

Range getRange() const
Returns active range setting.

void write(float value) const
Directly (more: user guide CEBO-LC, Data acquisition, single value I/O) set the specified voltage value on the output.

std::string getName() const
Returns the name of the output.

DigitalPort

This class is the interface to work with digital ports. Retrieve instances calling getDigitalPorts() of the respective Device.

void setOutputEnableMask(int mask) const
Set bitwise mask that defines which of the I/O's on the specified port are input and output. A bit of value 1 defines the specific I/O as output, e.g. mask = 0x03 means that I/O 0 and 1 are set to output, while I/O 2 to n are inputs.

int getIoCount() const
Returns the count of I/O's of the specific port.

int read() const
Read the state of the I/O's of the port.

void write(int value) const
Modify the output I/O's of the specified port.

std::string getName() const
Returns the name of the port.

std::string getIoName(int io) const
Returns the name of the I/O as specified by parameter **io**. The range of **io** is 0 <= **io** < getIoCount().

Counter

Interface class to counters inside the device. Class instances can be retrieved calling getCounters() of the specific Device.

void reset() const
Reset the counter to value 0.

void setEnabled(bool enabled) const
Enable or disable the counter. A disabled counter stays at the last value.

bool isEnabed() const
Return whether the counter is enabled or not.

void setConfig(CounterConfig counterConfig) const
Defines the counter behavior by using one of the constants specified in **Counter::CounterConfig**, listed in the following table.

Counter::CounterConfig getConfig() const
Returns the current setup. One of the constants in enumerator **Counter::CounterConfig**.

enum CounterConfig

- RisingEdge: Counter event is rising edge.
- FallingEdge: Counter event is falling edge.
- Alternating: Counter event are both rising and falling edges.

int64_t read() const
Read the current value of the counter.

std::string getName() const

Returns the name of the counter.

Trigger

Class that represents a trigger inside the device. Instances can be retrieved by calling getTriggers() of the respective Device.

void setEnabled(bool enabled) const
Enable or disable the trigger.

bool isEnabled() const
Return whether the trigger is enabled or not.

void setConfig(TriggerConfig triggerConfig) const
Defines the trigger behavior by using one of the constants specified in **Trigger::TriggerConfig**, listed in the following table.

Trigger::TriggerConfig getConfig() const
Returns the current setup. One of the constants in enumerator **Trigger::TriggerConfig**.

enum TriggerConfig

- OutputPulse: Trigger is output. Every trigger-event generates a positive pulse
- OutputAlternating: Trigger is output. Every trigger-event toggles the level
- InputRisingEdge: Trigger is input, reacts on a rising edge
- InputFallingEdge: Trigger is input, reacts on falling edge
- InputAlternating: Trigger is input, reacts on rising and falling edge.

std::string getName() const
Returns the name of the trigger.

Range

Use when handling range settings. Valid setting for an AnalogInput or AnalogOutput can be retrieved using their getSupportedRanges() method.

float getMinValue() const

Returns the lower voltage of the specific range.

float getMaxValue() const
Returns the upper voltage of the specific range.

Led

Interface to the LED's on the board. Instances are accessed calling getLeds() of the corresponding Device.

void setEnabled(bool enabled) const
Enable or disable the LED.

std::string getName() const
Returns the name of the LED.

CurrentSource

Class that represents the interface to the Fixed Current Outputs. Instances can be retrieved using getCurrentSources() of the respective Device.

float getReferenceCurrent() const
Returns the actual value of the Fixed Current Output, which is determined during manufacturing process and stored in onboard flash. The returned value is given in micro ampere.

std::string getName() const
Returns the name of the current source.

InputFrame

The input frame is a data class which stores measured samples from all inputs a device has (and which meet the frame concept). All samples inside a single frame are captured in a very short time span (which depends on the underlying device). Only values that have been selected using setupInputFrame() before sampling the frame are valid, the other are set to 0 by default.

float getSingleEnded(int index) const
Return the voltage value of single ended analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

float getDifferential(int index) const
Return the voltage value of differential analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

int getDigitalPort(int index) const
Return the I/O state of the digital port indicated by the given **index** at the moment the frame has been sampled.

bool getTrigger(int index) const
Return the state of the Trigger as specified by the given **index** in the moment the frame has been sampled.

int64_t getCounter(int index) const
Return the value of the Counter as specified by the given **index**.

OutputFrame

This class stores all values the should be set to the outputs as specified using setupOutputFrame(). Calling one of its methods does not affect the hardware. This is done when calling writeFrame(). An OutputFrame instance can't be created directly, as its metrics depends on the underlying device. The instance can retrieved calling createOutputFrame() of the device instance that should be used.

void setDigitalPort(int index, int value)
Set the bit mask for the DigitalPort as specified using **index**. Only bits set to output using setOutputEnableMask() are affected.

void setAnalogOutput(int index, float value)
Define the voltage value that should be set to the AnalogOutput as specified by the given **index**. The value will be calibrated before it is active.

Device

This is the primary class of the framework. Each instance corresponds to one physical device. The class is not intended to be instanced directly. Use the method enumerate() of the class LibraryInterface to access instances.
Many of the methods are related to the device itself. Besides that, instances to all device peripherals can be accessed.

void open()
This method must be called before doing anything with the device. Internally a communication to the device is constructed and several initialization is done. Various device and firmware specific constants are read and method resetDevice() is called.
To guarantee compatibility between API and device, if the major number of the device firmware is higher than the major number of the API, open() will fail, a newer version of the API must be used with this device.

void close()
This closes the connection to the device, frees all internal resources and allows the device to be used by others (in the application instance and in different applications). Calling close will never throw an exception, it will always fail silently.

void resetDevice()
Calling this will stop any hardware controlled processing and reset the device to its power up settings. This is usually necessary if errors occur (Except logic or communication errors).
Invoking this method in a multithreaded context should be done with extra care.

void resetPeripherals(int mask)
Resets specific peripherals on the device side. Parameter **mask** specifies which elements using a bit mask. The bit mask must be constructed using the following flags, which will be enhanced in future versions:

int Device::FlagResetInputFifo	Clears the FIFO used during data acquisition and is error flagged

std::string getIdentifier() const

This returns a unique identifier to the device which is constructed using the physical connection properties. It is guaranteed to be unique at runtime, as well as constant between reboots of the operating system (except updates to the operating system change the behavior how physical properties are enumerated).

Be aware: Plugging a device to a different location (e.g. different USB port) will change this specifier, it is not device dependent. Use getSerialNumber() in this case.
This method can be called in front of invoking open().

DeviceType getDeviceType() const
Returns the specific type of the device that is bound to the instance. This is one of class DeviceType static members.
This method can be called before invoking open().

std::string getFirmwareVersion() const
Returns the firmware version encoded as ASCII string, e.g. "1.0".

std::string getSerialNumber()
Returns the device serial number as string. This string is unique for each device.

float getTemperature()
Returns current temperature in °C of the device.

void setWatchdogTimeout(unsigned int &timeout)
Sets the watchdog timeout in 250ms steps. If enabled, the device reboots if no data transfer has been done in the given time frame. Default value is 0xffffffff, which disables this feature.

unsigned int getWatchdogTimeout() const
Read back current watchdog timeout value.

int calculateMaxBufferedInputFrames()
Depending on the specified inputs the device site buffer can store a limited amount of frames. This method calculates the count of frames that will fit into this buffer using the current frame setup, which is last set using setupInputFrame(). Modifying the frame setup invalidates this value and must be updated invoking this method again. This value

is primary intended to be used in context with startBufferedDataAcquisition().

void setupInputFrame(const InputVector &inputs)
When doing any form of frame based data acquisition, which is described in the user guide for CEBO-LC (topic data acquisition), this method must be used to select the inputs to be read. There's no limit which inputs can be selected, but every input can only be specified once. This method can only be called if no multi frame based data acquisition (more: user guide CEBO-LC, Data acquisition, multi frame DAQ) is active. Any subsequent frame based call is affected by this setup.
The InputVector can contain any Instance that is derived from class Input, AnalogInput, DigitalPort, Counter or Trigger.

float startBufferedDataAcquisition(float frameRate, int frameCount, bool externalStarted)
Starts a buffered data acquisition at the specified **frameRate** for exactly **frameCount** frames. If **externalStarted** is true, the acquisition is started at the moment an external trigger event has been detected, immediately otherwise. Parameter **frameRate** must be equal or smaller than the value reported by calculateMaxBufferedInputFrames(). The returned value is the frame rate that is really used, as not every possible frame rate can be handled by the device. This value is as near as possible to the specified frame rate. Detailed description for this can be found in the user guide CEBO-LC.

void startBufferedExternalTimedDataAcquisition(int frameCount)
Very similar to the method above, except that no frame rate is used the sample frame, but every time an external trigger has been detected one single frame is sampled. Acquisition is automatically stopped after the specified amount of frames has been sampled. Detailed description for this can be found in the user guide CEBO-LC, topic data acquisition, multi frame DAQ.

float startContinuousDataAcquisition(float frameRate, bool externalStarted)
Starts a data acquisition without frame limit. The host must read the sampled data as fast as possible, otherwise buffer overflow is signaled and the process has been failed. Frames are sampled at the specified **frameRate**. If **externalStarted** is true, sampling starts at the first detected trigger event, immediately otherwise. The return value is the frame rate that is really used, as not every possible rates are possible.

In a multithreaded context, it is advised to start reading frames before calling this method.
More details about data acquisition can be found in the user manual CEBO-LC, topic data acquisition, multi frame DAQ.

void startContinuousExternalTimedDataAcquisition()
Similar to the method above, but instead of a fixed frame rate, frames are sampled for every detected trigger. Detailed information about this can be found in the user manual, topic data acquisition, multi frame DAQ.

void stopDataAcquisition()
Stops a currently active data acquisition. The device buffer is not modified calling this, so unread data can be fetched subsequently.

InputFrameVector readBlocking(int frameCount)
When doing multi frame data acquisition (more: user manual CEBO-LC, topic data acquisition, multi frame DAQ), this is one method to read the sampled frames from device to host. This version will block the current thread until the specified amount of frames has been read, there's no timeout. After the call returns without an exception, the returned InputFrameVector contains exactly the specified amount of InputFrames. The alternative to this call is described below.

InputFrameVector readNonBlocking()
This method is similar to the one above, except that it always returns immediately, while trying to read as much frames as possible. The returned InputFrameVector contains all sampled InputFrames since the start of the data acquisition or the last frame read. Especially for high data amounts, this method should be called cyclically without to high delays.

InputFrame readFrame()
Read single frame as set up calling setupInputFrame() and return immediately. This cannot be called if multi frame data acquisition (more: user guide CEBO-LC, topic data acquisition, multi frame DAQ) is active.

void setupOutputFrame(const OutputVector &outputs)

Specifies which Outputs are involved in the next frame output process. There's no limit of the specified outputs, but no single instance can be used more than once. Valid outputs that can be added to the OutputVector are AnalogOutput and DigitalPort.

void writeFrame(const OutputFrame &frame)
Set all outputs that have been selected using setupOutputFrame() in a single call. Data to set on the respective outputs must be set in the given OutputFrame. The frame must be constructed using method createOutputFrame() of the same device instance were it is used.

OutputFrame createOutputFrame() const
Create an OutputFrame instance that fits to the device metrics from which instance the method is called. The frame can than be filled with data and used calling writeFrame() of the same device instance.

const AnalogInputVector &getSingleEndedInputs() const
Return list of all single ended inputs.

const AnalogInputVector &getDifferentialInputs() const
Return list of differential analog inputs.

const AnalogOutputVector &getAnalogOutputs() const
Return list of the analog outputs.
const DigitalPortVector &getDigitalPorts() const
Return list of digital ports.

const CounterVector &getCounters() const
Return list of counters.

const TriggerVector &getTriggers() const
Return list of triggers.

const CurrentSourceVector &getCurrentSources() const
Return list of current sources.

const LedVector &getLeds() const

Return list of LED's.

LibraryInterface

This class contains static functionality only. Its responsibility is to serve as interface to methods that are not bound to any device.

static std::string getApiVersion()
Report the version number of the underlying CeboMsr API encoded as ASCII string, i.e. "1.0".

static std::string getUsbBaseVersion()
Return the version number of the system interface USB layer encoded as ASCII string, i.e. "1.0".

static DeviceVector enumerate(const DeviceType &type)
The specified **type** parameter must be one of class DeviceType static members. The system is than scanned for known devices and subsequently filtered to meet the specified **type**. Devices that are already opened were skipped too. The returned list contains all candidates that are ready for use.

For each instance that should be used, open() must be called.
Starting a new enumeration invalidates the list from the previous invocation.

Java

The Java API is a thin layer on top of the CeboMsr API. It uses JNA to bind itself to the dynamic link library.

Compatibility

The whole development has been done on Oracle(c) Java SE 6. Other JVM's may be compatible as well.

External Dependencies

The only external dependency is JNA which can be found here on github. The version that is used during development is 3.3.0.

Error handling

Any logical or runtime error that is reported by the CeboMsr API is transformed to an exception. The following table lists all possible exception thrown by the Java implementation of the CeboMsr API:

Exception	Circumstances
IllegalArgumentException	If any of the used parameters has an invalid value in the current context.
IndexOutOfBoundsException	When using an invalid index. (Most of these are prevented due to the API design).
IllegalStateException	Thrown if something is called which is not allowed in this stage.
IOException	Indicates an unexpected behavior like communication problems with the device.

As seen in the table above, all exceptions but IOException are unchecked exceptions, so they are predictable at development time. In the case of IOException, human investigation may be required to clarify the reason of the problem.
All exceptions contain a textual description of the problem.

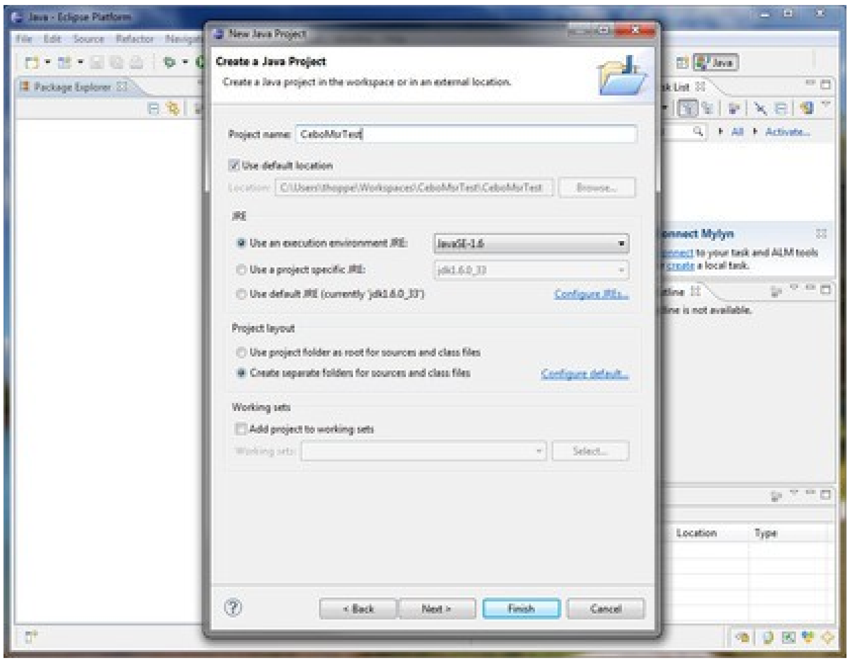
Eclipse Project Setup

The following section will outline how you:

- Create a new Java project using Eclipse.
- Add the CeboMsr API to it.
- Enable inline JavaDoc for the API.
- Write a simple Java program that uses the CeboMsr API.
- Configure debugging to start the application.

The requirements are an installed JDK (1.6.33 used below) and a running Eclipse (3.7 - Indigo used below).

First, create a new **Java Project**, name it **CeboMsrTest** and leave the rest of the settings untouched.



Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: CeboMsrTest

[✓] Use default location

Location: C:\Users\thoppe\Workspaces\CeboMsrTest\CeboMsrTest Browse...

JRE

[✓ Use an execution environment JRE: JavaSE-1.6]

Use a project specific JRE: jdk1.6.0_33

Use default JRE (currently 'jdk1.6.0_33') Configure JREs...

Project layout

Use project folder as root for sources and class files

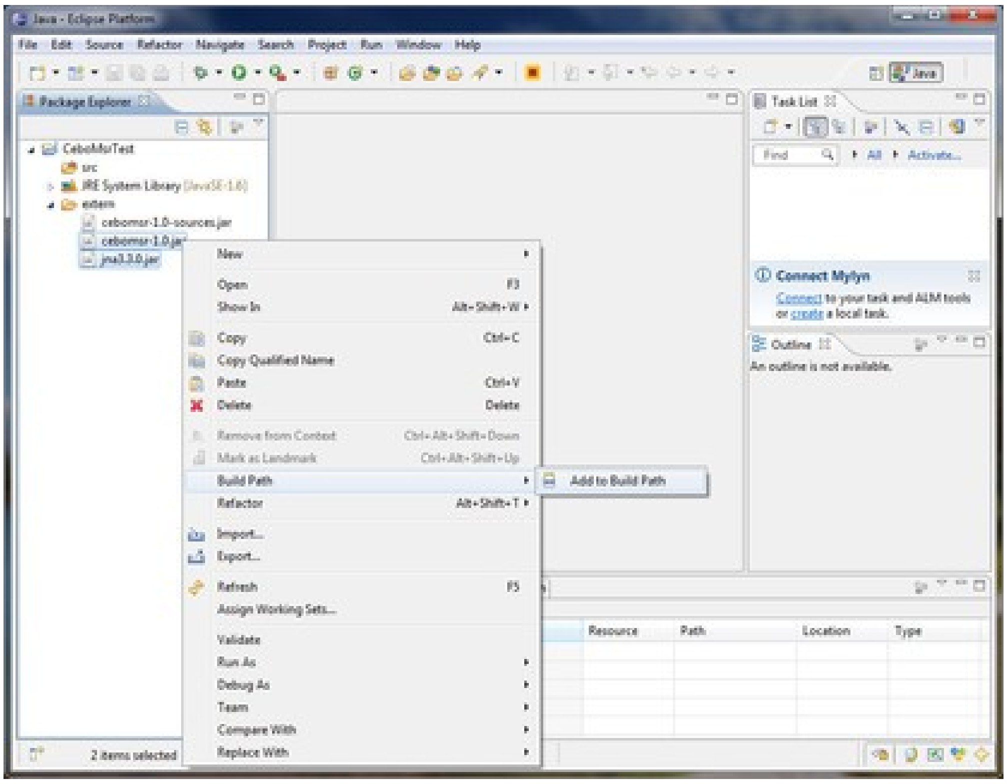
[✓ Create separate folders for sources and class files] Configure default...

Working sets

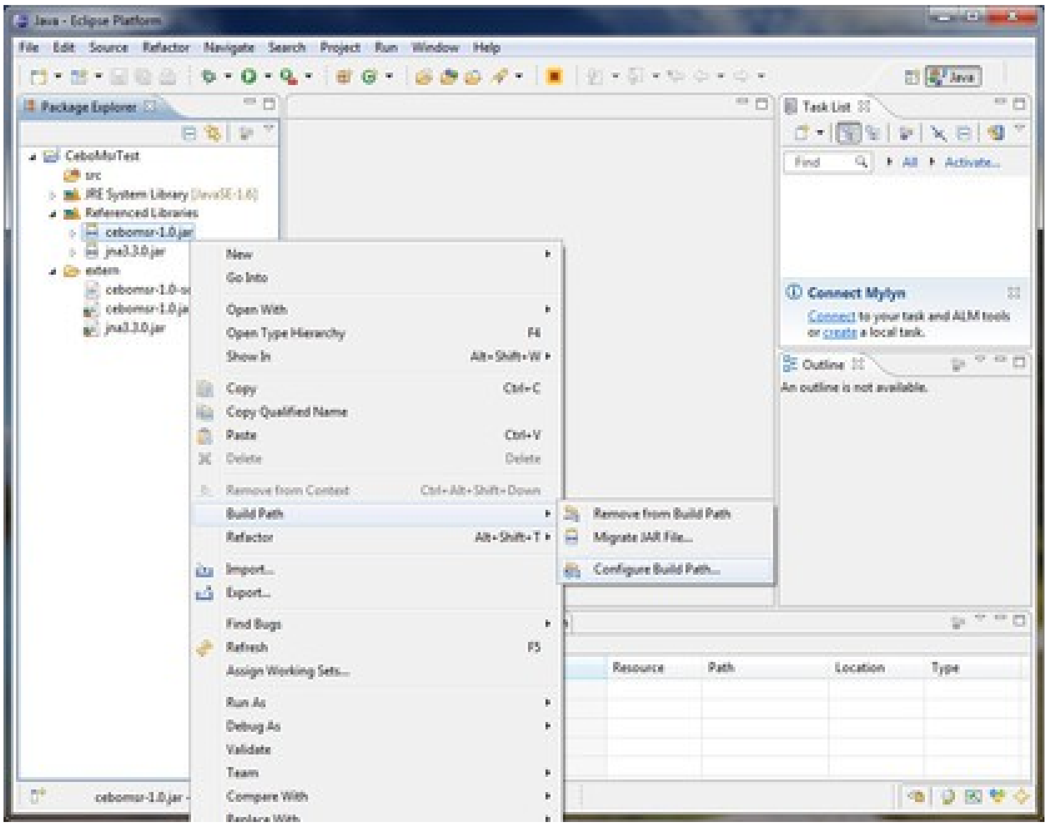
[ ] Add project to working sets

Working sets: Select...

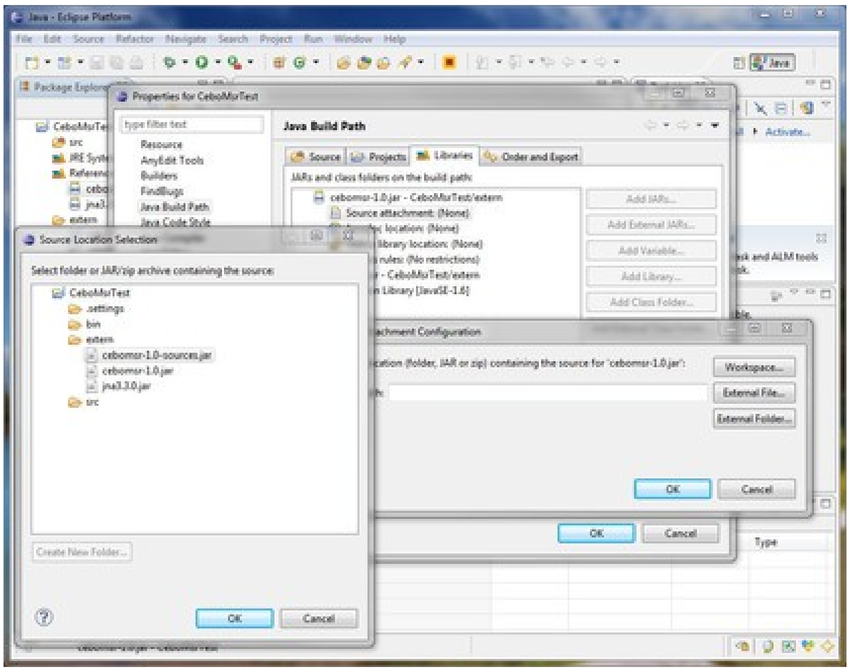
Create a **new folder** inside the project, name it **extern**. Copy the following files into it: **cebomsr-{ver}-sources.jar**, **cebomsr-{ver}.jar** and **jna{var}.jar**. Add the last two files to the **Build Path**.



To be able to debug into the API and to see the JavaDocs of the API, define the sources for the API jar file. Do this bye configuring the **Build Path** for **cebomsr-{ver}.jar**.



Select **Source attachment** beneath **cebomsr-{ver}.jar** and click on the **Edit...** button in the right menu. Click on **Workspace...** in the new dialog and select **cebomsr-{ver}-sources.jar** in the **extern** folder. Confirm all dialogs by clicking **OK**.

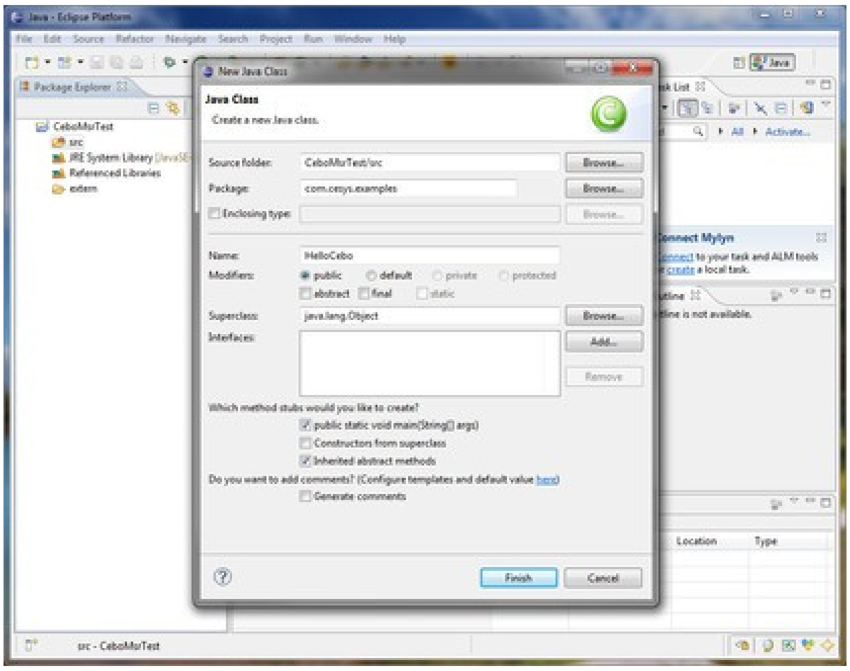

Select folder or JAR/zip archive containing the source:

- CeboMsrTest
  - .settings
  - bin
  - extern
    - cebomsr-1.0-sources.jar
    - cebomsr-1.0.jar
    - jna3.3.0.jar
  - src

JARs and class folders on the build path:

- cebomsr-1.0.jar - CeboMsrTest/extern
  - Source attachment: (None)

Create a new Class, name it **HelloCebo** and put it into any package you like (**com.cesys.examples** here). Select the checkbox to create a **main** method.


Java Class

Create a new Java class.

Source folder: CeboMsrTest/src

Package: com.cesys.examples

Name: HelloCebo

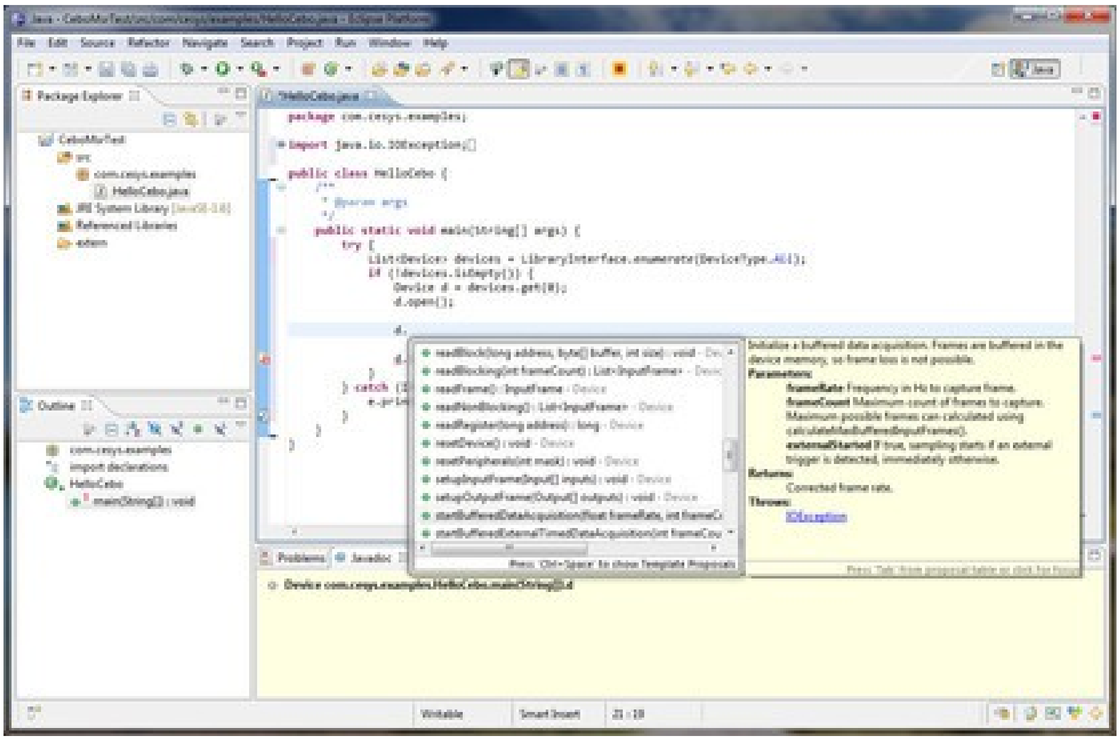
Modifiers: [✓ public] default private protected

Superclass: java.lang.Object

Which method stubs would you like to create?

- [✓] public static void main(String[] args)
- [ ] Constructors from superclass
- [✓] Inherited abstract methods

As you can see in the screenshot, the JavaDoc information is shown while editing.



```
package com.cesys.examples;

import java.io.IOException;

public class HelloCebo {
    /**
     * @param args
     */
    public static void main(String[] args) {
        try {
            List<Device> devices = LibraryInterface.enumerate(DeviceType.All);
            if (!devices.isEmpty()) {
                Device d = devices.get(0);
                d.open();

                d.
```

```
readBlock(long address, byte[] buffer, int size) : void - Dev
readBlocking(int frameCount) : List<InputFrame> - Devic
readFrame() : InputFrame - Device
readNonBlocking() : List<InputFrame> - Device
readRegister(long address) : long - Device
resetDevice() : void - Device
resetPeripherals(int mask) : void - Device
setupInputFrame(Input[] inputs) : void - Device
setupOutputFrame(Output[] outputs) : void - Device
startBufferedDataAcquisition(float frameRate, int frameC
startBufferedExternalTimedDataAcquisition(int frameCou
Press 'Ctrl+Space' to show Template Proposals
```

Initialize a buffered data acquisition. Frames are buffered in the device memory, so frame loss is not possible.
Parameters:
frameRate Frequency in Hz to capture frame.
frameCount Maximum count of frames to capture. Maximum possible frames can calculated using calculateMaxBufferedInputFrames().
externalStarted If true, sampling starts if an external trigger is detected, immediately otherwise.
Returns:
Corrected frame rate.
Throws:
IOException

Press 'Tab' from proposal table or click for focus

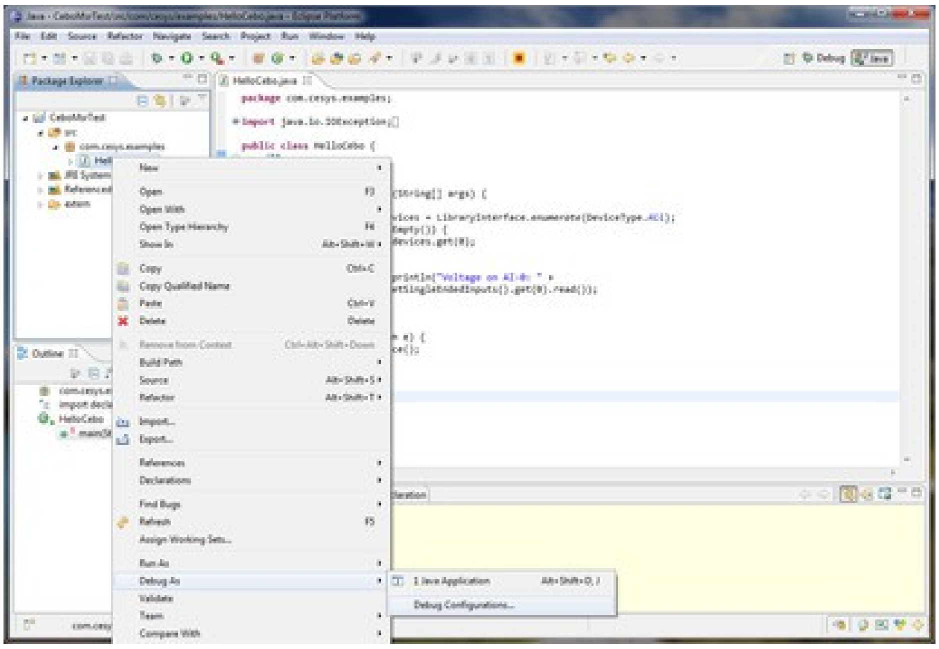
Device com.cesys.examples.HelloCebo.main(String[]).d

Here is the source of the example application. (Take attention if you have specified a different package name and copy and paste the source. Just fix the package specification.)

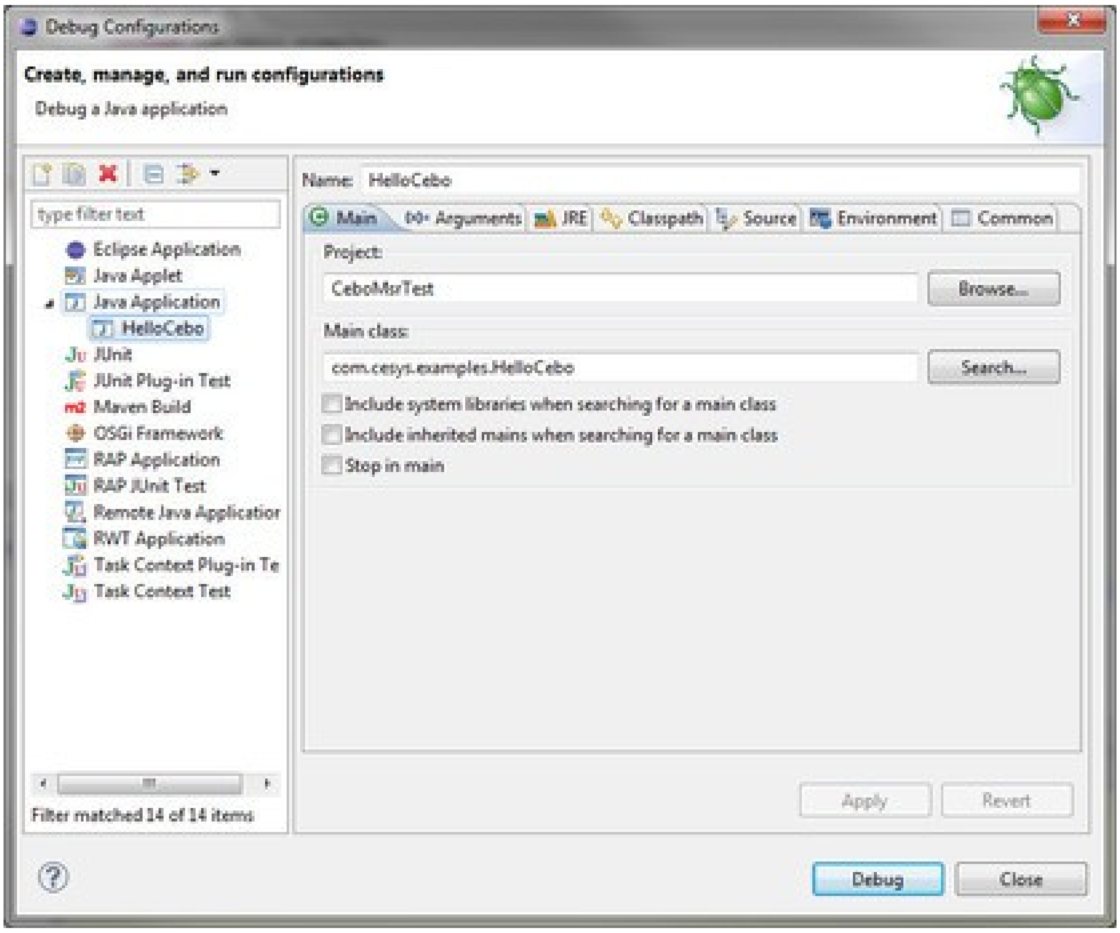
```
package com.cesys.examples;

import java.io.IOException;
import java.util.List;

import com.cesys.cebo.cebomsr.Device;
import com.cesys.cebo.cebomsr.DeviceType;
import com.cesys.cebo.cebomsr.LibraryInterface;

public class HelloCebo {
  /**
   * @param args
 */
```

```
  public static void main(String[] args) {
    try {
      List<Device> devices = LibraryInterface.enumerate(DeviceType.All);
      if (!devices.isEmpty()) {
        Device d = devices.get(0);
        d.open();

        System.out.println("Voltage on AI-0: " +
            d.getSingleEndedInputs().get(0).read());

        d.close();
   }
    } catch (IOException e) {
      e.printStackTrace();
  }
 }
}
```

To start debugging, open the context menu on the example class and select **Debug Configurations...**



Double Click on **Java Application** on the left side, this creates a debug session named **HelloCebo**.


Debug Configurations

Create, manage, and run configurations

Debug a Java application

type filter text

- Eclipse Application
- Java Applet
- Java Application
  - HelloCebo
- JUnit
- JUnit Plug-in Test
- Maven Build
- OSGi Framework
- RAP Application
- RAP JUnit Test
- Remote Java Applicatior
- RWT Application
- Task Context Plug-in Te
- Task Context Test

Filter matched 14 of 14 items

Name: HelloCebo

Main | Arguments | JRE | Classpath | Source | Environment | Common

Project:

CeboMsrTest

Main class:

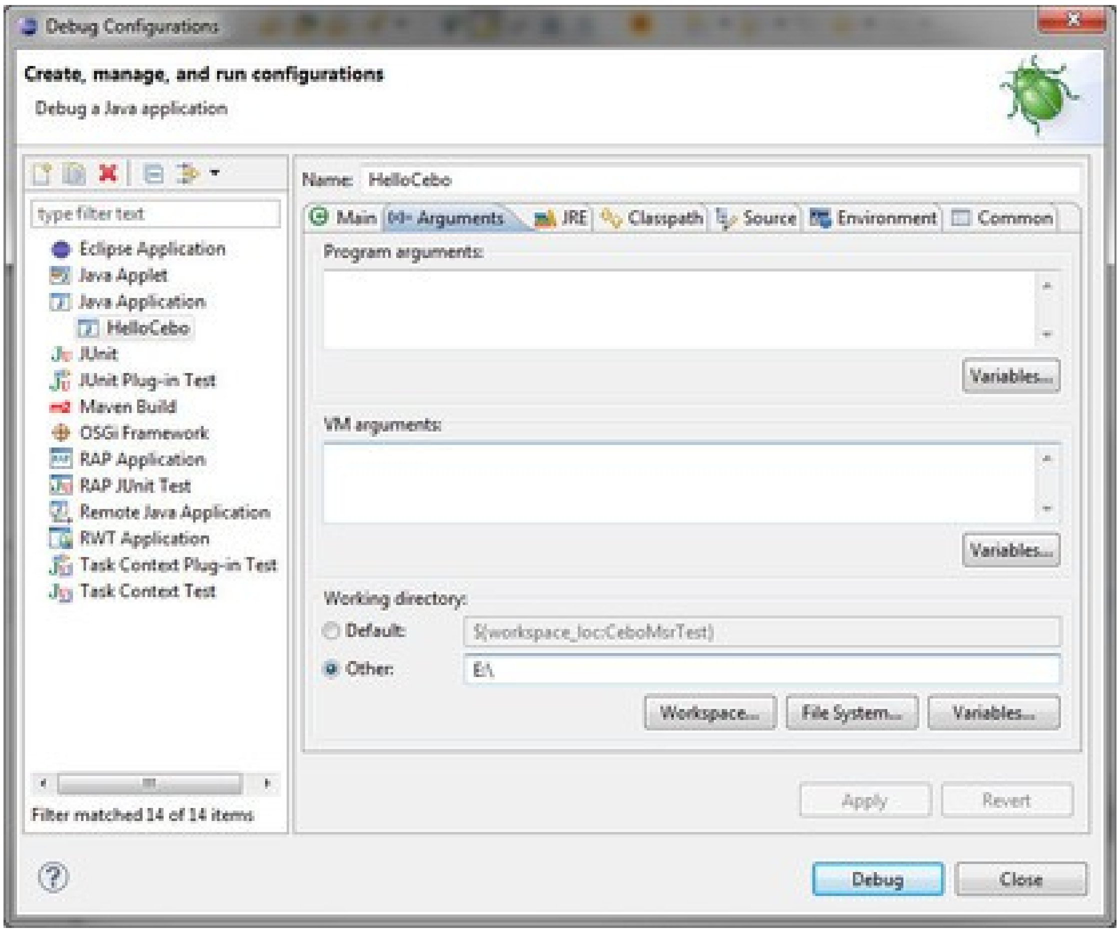
com.cesys.examples.HelloCebo

☐ Include system libraries when searching for a main class

☐ Include inherited mains when searching for a main class

☐ Stop in main

Switch to the Arguments tab for the **HelloCebo** configuration. In the lower part are settings for the working directory. Select a directory where the matching CeboMsr API libraries can be found, on Windows this is **cebomsr-{ver}-{arch}.dll** and **ceusbbase-{ver}-{arch}.dll**. **{ver}** of the cebomsr module must match the version of the used Java API. **{arch}** depends on the used JVM architecture, **not** the architecture of the operating system.



Name: HelloCebo

Working directory:
Default: ${workspace_loc:CeboMsrTest}
[✓ Other:] E:\

Clicking on Debug will close the dialog and start the debug session.

Basics

This section describes the first steps that are necessary to use the API. We start by showing a basic example that:

- searches for all known devices
- grabs the first available device
- opens the device
- resets its state
- closes the device

```
import com.cesys.cebo.cebomsr.*;

…

// Search for devices ...
List<Device> devices = LibraryInterface.enumerate(DeviceType.All);

// If at least one has been found, use the first one ...
if (!devices.isEmpty()) {
  Device device = devices.get(0);

  // Open device, nothing can be done without doing this.
  device.open();

 …

  // After some processing, the device can be resetted to its
  // power up condition.
  device.resetDevice();

 …

  // Finalize device usage, this free's up the device, so it can be used
  // again, including other applications.
  device.close();
}
```

Description

The first lines in the example import the complete namespace to the source unit.

In the application flow, enumeration must be always the first task when using the API. It searches the system for known, not yet used devices and offers the caller a list of device instances. The parameter that is used calling enumerate() of LibraryInterface specifies which devices should be searched:

- All known devices (DeviceType.All)
- Device classes (e.g. DeviceType.Usb)
- Specific types (e.g. DeviceType.CeboLC)

We search for all devices in the example above. The returned list contains candidates of type Device that can be used subsequently.
We use **isEmpty()** on the list to see if at least one device has been found. If a device has been found, the if branch is entered, were we continue by using the first device.

The call to open() flags the device for use. Any new enumeration (before calling on the same instance) will filter this instance. This is just a simple mechanism to prevent double access to the same device.

After the device has been opened, it can be used in any way. Method resetDevice() is shown here, because it has a superior role. It stops all active processing inside the device and resets all configuration values to their startup condition. It is not necessary to call it, but is a good way to reach a defined state i.e. in an error condition. It is implicitly invoked when calling open().

At the end of the current scope, method close() is called to signal API and system that the device is not used anymore. This is an important call, because leaving it in opened state will prevent it from getting enumerated again.

Single Value I/O

This section describes simple single input and output of values. It continues the example from section Basics, so an opened device instance is available. Single value I/O is described in the user guide CEBO-LC in detail.

```
// Read digital port #0 and write result to digital port #1 (mirror pins).
// At first, we need references to both ports.
DigitalPort dp0 = device.getDigitalPorts().get(0);
DigitalPort dp1 = device.getDigitalPorts().get(1);

// Configure port first, all bits of digital port #0 as input (default)
// and all bits of digital port #1 as output.
dp1.setOutputEnableMask(0xff);

// Read from #0 ...
int value = dp0.read();

// Write the value to #1 ...
dp1.write(value);

// Now some analog I/O, do it without any additional local references.
// Read single ended #0 input voltage ...
float voltageValue = device.getSingleEndedInputs().get(0).read();

// Write value to analog output #1
device.getAnalogOutputs().get(1).write(voltageValue);
```

Description

If you want work with device peripherals, get instances using the various component access methods of class Device. If more then one method must be called, create a local reference, this reduces the amount of code to write. After you have an instance, invocations to its methods affects this specific peripheral.

The example outlines how to read all I/O's of digital port #0 and mirror the result on port #1. The first requirement to accomplish this is to define the direction of the individual I/O's. By default, all I/O's are configured to be inputs, so the example shows how to set all I/O's on port #1 as output, calling setOutputEnableMask() on its reference. All bits that are '1' in the specified mask signal the I/O to be an output. This is all you have to do for configuration.

The mirroring step is quite easy, read() the value from the instance that represents port #0 and store it in a local integer. The result is subsequently used calling write() on the instance the represents port #1.

The second half of the example outlines how to access peripherals without local copies or references. This requires fewer but longer lines of code but has no behavioral difference. Choose which style you prefer on your own.

The example continues doing direct single value I/O using analog peripherals. Like the digital port part, an input is mirrored to an output.

The call to read() returns the calibrated voltage value on this input as float. Calling write() on the AnalogOutput instance that represents analog output #1 using the returned voltage value will directly modify the real output on the device.

Single Frame Input

This section presents an example that samples a selection of inputs in a single process. The difference to single value read is not that much on the coding side, but there's a large improvement when it comes to performance. Reading more than one input costs only an insignificant higher amount of time in comparison to single I/O.

Configuration of the individual peripherals is exactly the same, so setting the I/O direction if digital ports or set the input range on an analog input is almost identical.

```
// Prepare and fill the list of inputs to read.
Input[] inputs = new Input[] {
  device.getSingleEndedInputs().get(0),
  device.getSingleEndedInputs().get(1),
  device.getDifferentialInputs().get(1),
  device.getDigitalPorts().get(0),
  device.getDigitalPorts().get(1),
  device.getCounters().get(0)
};

// Setup device with this selection.
device.setupInputFrame(inputs);

// Read the values multiple times and write them to the console.
for (int i = 0; i < 100; ++i) {
  // Read all inputs into the instance of InputFrame.
  InputFrame inFrame = device.readFrame();

  // Write results to the console.
  System.out.println(
    "DigitalPort #0: " + inFrame.getDigitalPort(0) + ", " +
    "DigitalPort #1: " + inFrame.getDigitalPort(1) + ", " +
    "SingleEnded #0: " + inFrame.getSingleEnded(0) + " V, " +
    "SingleEnded #1: " + inFrame.getSingleEnded(1) + " V, " +
    "Differential #1: " + inFrame.getDifferential(1) + " V, " +
    "Counter #0: " + inFrame.getCounter(0));
}
```

Description

The primary work for frame input I/O is to setup the device with the list of inputs that are involved. An array which contains the inputs to sample is required.
The example does this in the upper part. It creates the array and add several inputs to it:

- Single ended analog input #0 and #1
- Differential analog input #1
- Digital ports #0 and #1
- Counter #0
-

This list is than used calling setupInputFrame(), which prepares the device with this setup. This is active until:

- A call to resetDevice()
- The device is closed
- Or a new list is specified
-

In the subsequent loop, the specified inputs are sampled in every loop cycle using method readFrame(). The call lasts until all inputs are sampled and returns the values immediately in an instance of class InputFrame.

This instance holds all sampled values and offers methods to request them, which is shown in the lower part of the example.

Single Frame Output

In this section, concurrent modification of outputs will be outlined. In comparison to single value I/O, this technique is very efficient when working with more than one single output. As you can see in the following example, using it is quite straightforward.

```
// Write to analog out #1 and digital out #2 in one call.
// Prepare and set output selection.
Output[] outputs = new Output[] {
  device.getDigitalPorts().get(2),
  device.getAnalogOutputs().get(1)
};

// Prepare device ...
device.setupOutputFrame(outputs);

// Create instance of OutputFrame. There's no direct construction, as this
// instance may vary between different device types.
OutputFrame outFrame = device.createOutputFrame();

// Write it to hardware, modify contents in every looping.
for (int i = 0; i < 100; ++i) {
  outFrame.setAnalogOutput(1, (float)(3 * Math.sin((float)i / 100.f)));
  outFrame.setDigitalPort(2, i);

  device.writeFrame(outFrame);
}
```

Description

First of all, the example assumes that an opened instance of class Device is available, as well as that DigitalPort #2 is configured to be output.
Similar to the other direction, the device must be set up with a list of outputs. This setup is active until:

- A call to resetDevice()
- The device is closed
- Or a new list is set up

An array that contains the outputs to set is required. In the upper part of the example, you can see the creation of this array, adding outputs DigitalPort #2 and AnalogOutput #1 to it and activate the setup using setupOutputFrame().

The subsequent call to createOutputFrame() creates an instance of type OutputFrame, which fits to the device metrics. There's no other way to create this type.

In the loop below, every cycle modifies the values of the outputs specified during setup. This does not do anything other than storing the values inside the frame. The active modification of the outputs is done using writeFrame(), which transfers the values to the respective peripherals.

Multi Frame Input

This section describes how to read many frames at once, a very useful feature when you want sample inputs at a constant frequency or using an external trigger. The API offers very handy methods. The most problematic thing is to choose the right start method including its parameters.

The example below samples five different inputs at a constant rate of 300 Hz, 20 x 25 frames, completely cached inside the device buffer.

```
// Construct selection that contains inputs to read from.
Input[] inputs = new Input[] {
  device.getSingleEndedInputs().get(0),
  device.getSingleEndedInputs().get(1),
  device.getDifferentialInputs().get(1),
  device.getDigitalPorts().get(0),
  device.getDigitalPorts().get(1)
};

// Prepare device with this collection ...
device.setupInputFrame(inputs);

// Start sampling ...
device.startBufferedDataAcquisition(300, 20 * 25, false);

// Read 20 x 25 frames using blocked read,
// this function returns after *all* (25) requested frames are collected.
for (int i = 0; i < 20; ++i) {
  // Read 25 frames ...
  List<InputFrame> frames = device.readBlocking(25);

  // Write out the 1st one.
  InputFrame inFrame = frames.get(0);
  System.out.println(
    "DigitalPort #0: " + inFrame.getDigitalPort(0) + ", " +
    "DigitalPort #1: " + inFrame.getDigitalPort(1) + ", " +
    "SingleEnded #0: " + inFrame.getSingleEnded(0) + " V, " +
    "SingleEnded #1: " + inFrame.getSingleEnded(1) + " V, " +
    "Differential #1: " + inFrame.getDifferential(1) + " V");
}

// Stop the DAQ.
device.stopDataAcquisition();
```

Description

The first lines in this example are similar to single frame example. The device is set up to sample the specified inputs into a single call. Single ended input #0 and #1, differential input #1 and digital ports #0 and #1 are used here.

The real data acquisition is than started calling startBufferedDataAcquition(). Choosing this method to start up, combined with the used parameters means the following:

- Data has to be completely stored in the device memory (buffered).
- Sampling is done using hardware timed capture at 300 Hz.
- The number of frames to capture is 20 x 25 = 500.
- No external trigger is necessary to initiate the process.

This method does not only configure the DAQ process, but start it as well. In the case of buffered mode, this is really uncritical. If you require continuous mode, a buffer overrun may occur shortly after starting the DAQ, so it is very important to either:

- Start to read the sampled frames immediately, as shown in the example.
- Use a second thread to read the frames, start this thread before the DAQ is started.

The example continues with a for loop where every cycle reads 25 frames and outputs the sampled values of the first frame in this block. The used method Device| outline returns after the given amount of frames has been read and stalls the thread up to this point. Use readBlocking() with care, because it has two downsides:

- It blocks the access to the whole API due to internal thread locking mechanisms.
- If the specified amount of frames is too small, especially at higher frame rates, buffer overruns on the device side will occur, as the call and transfer overhead is too high.

It is a very convenient way to read a defined number of frames. The alternative is the use readNonBlocking(), which returns immediately with all captured frames at the moment of calling.

Both read methods return the list of captured frames. The example uses the first frame of this block (which is guaranteed to contain 25 frames in the example), and output the sampled values of all specified inputs to the console.

At the end of the example, DAQ is stopped using stopDataAcquisition().

Counter

The example below shows how to use a counter. To allow this using software, a wired connection between IO-0 and CNT is necessary. IO-0 is used to generate the events that the counter should count.

```
// Create local copies, this shortens the source.
DigitalPort dp0 = device.getDigitalPorts().get(0);
Counter cnt = device.getCounters().get(0);

// Set IO-0 as output.
dp0.setOutputEnableMask(0x01);

// Enable the counter.
cnt.setEnabled(true);

// Check the counters current value.
System.out.println("Counter before starting: " + cnt.read());

// Pulse IO-0 3 times.
for (int i = 0; i < 3; ++i) {
  dp0.write(0x01);
  dp0.write(0x00);
}

// Check the counters current value.
System.out.println("Counter should be 3: " + cnt.read());

// Reset and ...
cnt.reset();

// ... disable the counter.
cnt.setEnabled(false);

// Check the counters current value.
System.out.println("Counter should be 0: " + cnt.read());

// Pulse IO-0 3 times, the counter should ignore these pulses.
for (int i = 0; i < 3; ++i) {
  dp0.write(0x01);
  dp0.write(0x00);
}

// Check the counters current value.
System.out.println("Counter should still be 0: " + cnt.read());
```

Description

Like most of the previous examples, an instance of class Device is required. This can be retrieved using the procedure described here. To demonstrate the counter without external peripherals, the software controls the events for the counter as well. This extends the example to around twice its size, but its still easy to understand.

To reduce the amount of code, dp0 and cnt are constructed as local copies of both the first DigitalPort and the Counter.

The LSB of digital port #0 represents IO-0. To modify IO-0 via software, the example continues to set this I/O as output using method setOutputEnableMask().

Counters are disabled by default, so the next required task is to enable it, otherwise no event will be detected. The example does this by calling setEnabled() of the Counter instance.

The first counter value that is printed out will be zero, which is the default value after startup or reset.

The counter reacts on every rising edge. The example shows this by pulsing IO-0 three times, setting its level high and than low in every loop cycle. The result is than printed to the console. It is expected to be three, based on the pulses in the previous loop (If not, verify the wired connection between IO-0 and CNT).

The value of the counter is than set to zero calling its reset() method. In addition, deactivates the counter to any event.

The remaining example code outlines this, by pulsing IO-0 three times again, but the console output shows that now flanks have been counted in this state.

Trigger

The following example is much longer than the previous, but outlines various new things:

- Working with multiple devices.
- Use different multi frame modes.
- Show how to use triggers in input and output mode.

You will need two devices for this example to run, as both devices are chained together using triggers. The first device acts as master, while the second device is the slave. Every time, the master samples a frame, an output trigger is generated, while the slave captures its frame if this trigger has been raised.

Devices must be connected to each other using two wires. Ground (GND) to ground and trigger (TRG) to trigger.

TIP #1: This example can easily be extended to support more than one slave, you only have to set up each slave the same way as the slave in the example.

TIP #2: At high bandwidth, it may be necessary to put the individual readNonBlocking() calls to separate threads. Otherwise reading the data from one device may last as long as the device side buffer on the second device will need to overflow.

The description is located below the example.

```
public class TriggerExample {

  /**
   * Write frames to console.
   * @param device Device string.
   * @param frames Frame to dump.
   */
  private void dumpFrames(String device, List<InputFrame> frames) {
    for (InputFrame f : frames) {
      System.out.println(device + ": " +
          "se#0: " + f.getSingleEnded(0) + " V, " +
          "dp#0: " + f.getDigitalPort(0));
    }
  }

  /**
   * Start DAQ, read frames and output the results.
```

```
 * @param master Master device.
 * @param slave Slave device.
 * @throws IOException
 */
void runDataAcquisition(Device master, Device slave) throws IOException {
  // The slave must be started first, as it reacts on the master's
  // trigger. Continuous DAQ is used. Timed using an external trigger.
  slave.startContinuousExternalTimedDataAcquisition();

  // The master uses hardware timed DAQ, continuous at 50 Hz.
  // The 'false' here signals that the process should start immediately.
  master.startContinuousDataAcquisition(50, false);

  // The example reads at least 10 frames from
  // both devices and subsequently output the samples.
  int masterFrames = 0, slaveFrames = 0;
  while (masterFrames < 10 || slaveFrames < 10) {
    // Start by reading frames from master,
    // output it and increment counter.
    List<InputFrame> frames = master.readNonBlocking();
    dumpFrames("master", frames);
    masterFrames += (int)frames.size();

    // Do the same with the slave.
    frames = slave.readNonBlocking();
    dumpFrames("slave", frames);
    slaveFrames += (int)frames.size();

    // Don't poll to frequent, this would fully utilize one core.
    try {
      Thread.sleep(1);
    } catch (InterruptedException ex) {}
  }

  // Finished, gracefully stop DAQ.
  slave.stopDataAcquisition();
  master.stopDataAcquisition();
}

/**
 * Both devices are fully configured here.
 * @param master Master device.
 * @param slave Slave device.
 * @throws IOException
 */
void configure(Device master, Device slave) throws IOException {
  // The trigger for the master must be set to alternating output.
  master.getTriggers().get(0).setConfig(TriggerConfig.OutputAlternating);

  // The slave's trigger must be set to alternating as well.
  slave.getTriggers().get(0).setConfig(TriggerConfig.InputAlternating);
```

```
    // Both devices now gets configured to the same input frame layout.
    master.setupInputFrame(new Input[] {
      master.getSingleEndedInputs().get(0),
      master.getDigitalPorts().get(0)
    });

    // ... slave
    slave.setupInputFrame(new Input[] {
      slave.getSingleEndedInputs().get(0),
      slave.getDigitalPorts().get(0)
    });
  }

  /**
   * The examples main method.
   */
  private void runExample() {
    try {
      // Search for the devices, exactly two are required.
      List<Device> devices = LibraryInterface.enumerate(DeviceType.All);
      if (2 != devices.size()) {
        System.out.println("Exactly two devices are required.");
        return;
      }

      // As both devices can act as master,
      // we simply use the first one for this role.
      Device master = devices.get(0);
      master.open();
      System.out.println("Master: " + master.getSerialNumber() +
          "@" + master.getIdentifier());

      // ... and the second as slave.
      Device slave = devices.get(1);
      slave.open();
      System.out.println("Slave: " + slave.getSerialNumber() +
          "@" + slave.getIdentifier());

      // Configure both master and slave ...
      configure(master, slave);

      // ... and do the DAQ.
      runDataAcquisition(master, slave);

      // Close both.
      master.close();
      slave.close();
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
  }
  public static void main(String[] args) {
    new TriggerExample().runExample();
  }
}
```

Description

Read the example bottom up, starting in the **runExample()** method. Nothing really new in comparison to the previous examples, except that two devices are used concurrently. Both devices gets opened and a short information about master and slave is printed to the console.

What follows is the configuration, which is shown in method **configure()**. Trigger #0 of the master device is set to output, alternating. This means, every time, the master captures a frame, its first trigger toggles the level.

Trigger #0 of the slave is configured to work as input, alternating as well. So the slave captures a frame if its first trigger detects that the level has been toggled.
That's all for the trigger configuration. Method **configure()** completes the process by setting up a frame using single ended input #0 and digital port #0 as described in detail here.

Returned to **runExample()**, method **runDataAcquisition()** is invoked, which shows how data from both master and slave is read. The most important part here is, how both DAQ processes are started. The slave is started first, otherwise he may miss one or more events. The slave is set to sample frames every time a trigger has been detected, without any count limits. This is done calling startContinuousExternalTimedDataAcquisition().

The master's DAQ is than started calling startContinuousDataAcquisition(), which means, no frame count limitation at a specific frequency. The example uses a very low frequency, 50 Hz, and starts immediately (By using **false** for parameter **externalStarted**).

At runtime, the master will than start to sample frames at the given 50 Hz, toggle its Trigger every time, while the slave captures a frame every time this event is received at his input trigger pin. Both capture frames synchronously.

The example continues by reading frames from both devices one after another until both have returned at least 10 frames. Captured values are written to the console using method **dumpFrames()**. The loop contains a 1 ms sleep, otherwise the usage of one CPU core would go to 100%, which is never a good idea.

After the frames have been read, DAQ is stopped on both devices, calling stopDataAcquisition(). The program returns to **runExample()** and close() both devices.

Info

This section simply outlines what information you can get from the API about the API itself, the device and its peripherals.

```
// Put out some device specific information.
System.out.println("Device type: " + device.getDeviceType().getName());
System.out.println("USB base ver: " + LibraryInterface.getUsbBaseVersion());
System.out.println("API vers: " + LibraryInterface.getApiVersion());
System.out.println("Firmware ver: " + device.getFirmwareVersion());
System.out.println("Identifier: " + device.getIdentifier());
System.out.println("Serial number: " + device.getSerialNumber());
System.out.println("Temperature: " + device.getTemperature());

// Get the real current of the reference current sources.
for (CurrentSource source : device.getCurrentSources()) {
  System.out.println("Reference current of "
    + source.getName() + ": "
    + source.getReferenceCurrent() + " uA");
}

// Retrieve information about single ended input #0.
// This works on all analog inputs and outputs.
AnalogInput se0 = device.getSingleEndedInputs().get(0);
System.out.println("Info for single ended input " + se0.getName() + ":");
System.out.println("Min. ICD: " + se0.getMinInterChannelDelay() + " us");
System.out.println("  Current range: "
  + se0.getRange().getMinValue() + " V to "
  + se0.getRange().getMaxValue() + " V");
  System.out.println("Supported ranges:");
for (int i = 0; i < se0.getSupportedRanges().size(); ++i) {
  Range range = se0.getSupportedRanges().get(i);
  System.out.println("    Range #" + i + " range: "
    + range.getMinValue() + " V to "
    + range.getMaxValue() + " V, "
    + "Def. ICD: "
    + se0.getDefaultInterChannelDelay(range) + " us");
}

// Get info for digital port #1.
DigitalPort dp1 = device.getDigitalPorts().get(1);
System.out.println("Info for digital port " + dp1.getName() + ":");
System.out.println("  Count of I/O's: " + dp1.getIoCount());
for (int i = 0; i < dp1.getIoCount(); ++i) {
  System.out.println("    I/O #" + i + ":" + dp1.getIoName(i));
}
```

Description

As most of the previous examples, this assumes an opened device as well. How to do this is described here.

The block in the upper part of the example writes any API or device specific information to the console. The printed information is self-explanatory.

The first loop iterates over all CurrentSources of the connected device. For every source, its real current value is printed out. These values were determined during device calibration.

In the following block, information about single ended input #0 is printed out, which is the active range setting, as well as all ranges that are valid for this port. All ranges report their lower and upper bound using methods getMinValue() and getMaxValue(). This can be done for every AnalogInput and AnalogOutput.

The last part of the example prints information about digital port #1. The value retrieved by getIoCount() is the number of I/O's that can be accessed by this DigitalPort. This may vary between the individual ports a device has. The names of all its single I/O's are printed as well.

Class Reference

The class reference is a complete but short overview of all classes and their methods used in the Java API. It does not outline how the components have to be used.

The sections parallel to this topic show many practical examples and should be the first you have to read to understand and use the API. A good starting point is this topic.

Interfaces

Both interfaces below are used to group input- and output peripherals.

interface Input
Used to group peripherals which can act as input.

interface Output
Used to group peripherals which can act as output.

DeviceType

This class is an enumeration of device types and device classes. Its static instances can be used to control the enumeration process. Besides that, each device reports its class using this type in method getDeviceType().

final static DeviceType All
Includes all known devices, independent which bus type is used. In the current stage, this equals the following instance, Usb.

final static DeviceType Usb
By using this instance in the enumeration process, all known devices connected via USB are reported.

final static DeviceType CeboLC
This instance must be specified if CEBO LC devices should be searched. In addition, getDeviceType() of Device returns this if the device is of this type.

final static DeviceType CeboStick
This instance must be specified if CEBO STICK devices should be searched. In addition, getDeviceType() of Device returns this if the device is of this type.

String getName()
Returns the name, e.g. "CeboLC" for the CeboLC instance.

AnalogInput

This class is the host side interface to any analog input, so you have to use its methods to request or modify the peripheral that this instance is assigned to. Get an instance of this class by calling either getSingleEndedInputs() or getDifferentialInputs() of the corresponding Device instance.

List<Range> getSupportedRanges()
Returns the list of Ranges this input supports. You can use each of these calling setParameters().

int getDefaultInterChannelDelay(Range range)
Return the default interchannel delay at the specified Range in microseconds.

int getMinInterChannelDelay()
Return the minimal interchannel delay for this input in microseconds.

void setParameters(Range range)
Sets the Range level on the analog input. Overwrites any previously adjusted interchannel delay with the default value.

int setParameters(Range range, int interChannelDelay)
Set Range for this input. In addition, the interchannel delay in microseconds is adjusted manually. The returned value is the interchannel delay that is really used (as not all specified values can be handled by the hardware).

Range getRange()
Returns active range setting.

int getInterChannelDelay()
Returns active interchannel delay.

float read()
Returns voltage value from input directly (more: user guide Cebo LC; data acquisition; single value I/O)

String getName()
Returns the name of the input.

AnalogOutput

This class represents an analog output. You can get an instance of this class calling getAnalogOutputs() from the corresponding Device.

List<Range> getSupportedRanges()
Returns the list of Ranges this input supports. You can use each of these calling setParameters().

void setParameters(Range range)
Sets the Range on the analog output.

Range getRange()
Returns active range setting.

void write(float value)
Directly set the specified voltage value on the output.

String getName()
Returns the name of the output.

DigitalPort

This class is the interface to work with digital ports. Retrieve instances calling getDigitalPorts() of the respective Device.

void setOutputEnableMask(int mask)
Set bitwise mask that defines which of the I/O's on the specified port are input and output. A bit of value 1 defines the specific I/O as output, e.g. mask = 0x03 means that I/O 0 and 1 are set to output, while I/O 2 to n are inputs.

int getIoCount()
Returns the count of I/O's of the specific port.

int read()
Read the state of the I/O's of the port.

void write(int value)
Modify the output I/O's of the specified port.

String getName()
Returns the name of the port.

String getIoName(int io)
Returns the name of the I/O as specified by parameter io. The range of io is 0 <= io < getIoCount().

Counter

Interface class to counters inside the device. Class instances can be retrieved calling getCounters() of the specific Device.

void reset()
Reset the counter to value 0.

void setEnabled(boolean enabled)
Enable or disable the counter. A disabled counter stays at the last value.

boolean isEnabled()
Return whether the counter is enabled or not.

void setConfig(Counter.CounterConfig counterConfig)
Defines the counter behavior by using one of the constants specified in **Counter.CounterConfig**, listed in the following enumeration.

Counter.CounterConfig getConfig()
Returns the current setup. One of the constants in enumerator **Counter.CounterConfig**.

enum CounterConfig

- RisingEdge: Counter event is rising edge.
- FallingEdge: Counter event is falling edge.
- Alternating: Counter event are both rising and falling edges.

long read()
Read the current value of the counter.

String getName()
Returns the name of the counter.

Trigger

Class that represents a trigger inside the device. Instances can be retrieved by calling getTriggers() of the respective Device.

void setEnabled(booean enabled)
Enable or disable the trigger.

boolean isEnabled()
Return whether the trigger is enabled or not.

void setConfig(Trigger.TriggerConfig triggerConfig)
Defines the trigger behavior by using one of the constants specified in **Trigger.TriggerConfig**, listed in the following enumeration.

Trigger.TriggerConfig getConfig()
Returns the current setup. One of the constants in enumerator **Trigger.TriggerConfig**.

enum TriggerConfig

- OutputPulse: Trigger is output. Every trigger-event generates a positive pulse
- OutputAlternating: Trigger is output. Every trigger-event toggles the level
- InputRisingEdge: Trigger is input, reacts on a rising edge
- InputFallingEdge: Trigger is input, reacts on falling edge
- InputAlternating: Trigger is input, reacts on rising and falling edge.

String getName()
Returns the name of the trigger.

Range

Use when handling range settings. Valid setting for an AnalogInput or AnalogOutput can be retrieved using their getSupportedRanges() method.

float getMinValue()
Returns the lower voltage of the specific range.

float getMaxValue()
Returns the upper voltage of the specific range.

Led

Interface to the LED's on the board. Instances are accessed calling getLeds() of the corresponding Device.

void setEnabled(boolean enabled)
Enable or disable the LED.

String getName()
Returns the name of the LED.

CurrentSource

Class that represents the interface to the Fixed Current Outputs. Instances can be retrieved using getCurrentSources() of the respective Device.

float getReferenceCurrent()
Returns the actual value of the Fixed Current Output, which is determined during manufacturing process and stored in onboard flash. The returned value is given in micro ampere.

String getName()
Returns the name of the current source.

InputFrame

The input frame is a data class which stores measured samples from all inputs a device has (and which meet the frame concept). All samples inside a single frame are captured in a very short time span (which depends on the underlying device). Only values that have been selected using setupInputFrame() before sampling the frame are valid, the other are set to 0 by default.

float getSingleEnded(int index)
Return the voltage value of single ended analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

float getDifferential(int index)
Return the voltage value of differential analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

int getDigitalPort(int index)
Return the I/O state of the digital port indicated by the given **index** at the moment the frame has been sampled.

boolean getTrigger(int index)
Return the state of the Trigger as specified by the given **index** in the moment the frame has been sampled.

long getCounter(int index)
Return the value of the Counter as specified by the given **index**.

OutputFrame

This class stores all values the should be set to the outputs as specified using setupOutputFrame(). Calling one of its methods does not affect the hardware. This is done when calling writeFrame(). An OutputFrame instance can't be created directly, as its metrics depends on the underlying device. The instance can retrieved calling createOutputFrame() of the device instance that should be used.

void setDigitalPort(int index, int value)
Set the bit mask for the DigitalPort as specified using **index**. Only bits set to output using setOutputEnableMask() are affected.

void setAnalogOutput(int index, float value)
Define the voltage value that should be set to the AnalogOutput as specified by the given **index**. The value will be calibrated before it is active.

Device

This is the primary class of the framework. Each instance corresponds to one physical device. The class is not intended to be instanced directly. Use the method enumerate() of the class LibraryInterface to access instances. Many of the methods are related to the device itself. Besides that, instances to all device peripherals can be accessed.

void open()
This method must be called before doing anything with the device. Internally a communication to the device is constructed and several initialization is done. Various device and firmware specific constants are read and method resetDevice() is called. To guarantee compatibility between API and device, if the major number of the device firmware is higher than the major number of the API, open() will fail, a newer version of the API must be used with this device.

void close()
This closes the connection to the device, frees all internal resources and allows the device to be used by others (in the application instance and in different applications). Calling close will never throw an exception, it will always fail silently.

void resetDevice()
Calling this will stop any hardware controlled processing and reset the device to its power up settings. This is usually necessary if errors occur (Except logic or communication errors). Invoking this method in a multithreaded context should be done with extra care.

void resetPeripherals(int mask)
Resets specific peripherals on the device side. Parameter mask specifies which elements using a bit mask. The bit mask must be constructed using the following flags, which will be enhanced in future versions:

int Device.FlagResetInputFifo	Clears the FIFO used during data acquisition and its error flags.

String getIdentifier()
This returns a unique identifier to the device which is constructed using the physical connection properties. It is guaranteed to be unique at runtime, as well as constant between reboots of the operating system (except updates to the operating system change the behavior how physical properties are enumerated).

Be aware: Plugging a device to a different location (e.g. different USB port) will change this specifier, it is not device dependent. Use getSerialNumber() in this case.
This method can be called in front of invoking open().

DeviceType getDeviceType()
Returns the specific type of the device that is bound to the instance. This is one of class DeviceType static members.
This method can be called before invoking open().

String getFirmwareVersion()
Returns the firmware version encoded as string, e.g. "1.0".

String getSerialNumber()
Returns the device serial number as string. This string is unique for each device.

float getTemperature()
Returns current temperature in °C of the device.

void setWatchdogTimeout(long &timeout)
Sets the watchdog timeout in 250ms steps. If enabled, the device reboots if no data transfer has been done in the given time frame. Default value is 0xffffffff, which disables this feature.

long getWatchdogTimeout() const
Read back current watchdog timeout value.

int calculateMaxBufferedInputFrames()
Depending on the specified inputs the device site buffer can store a limited amount of frames. This method calculates the count of frames that will fit into this buffer using the current frame setup, which is last set using setupInputFrame(). Modifying the frame setup invalidates this value and must be updated invoking this method again. This value is primary intended to be used in context with startBufferedDataAcquisition().

void setupInputFrame(Input[] inputs)
When doing any form of frame based data acquisition, which is described in the user guide CEBO-LC, this method must be used to select the inputs to be read. There's no limit which inputs can be selected, but every input can only be specified once. This method can only be called if no multi frame based data acquisition is active.

Any subsequent frame based call is affected by this setup.

The input array can contain any instance that implements Input, AnalogInput, DigitalPort, Counter or Trigger.

float startBufferedDataAcquisition(float frameRate, int frameCount, boolean externalStarted)
Starts a buffered data acquisition at the specified **frameRate** for exactly **frameCount** frames. If **externalStarted** is true, the acquisition is started at the moment an external trigger event has been detected, immediately otherwise. Parameter **frameRate** must be equal or smaller than the value reported by calculateMaxBufferedInputFrames(). The returned value is the frame rate that is really used, as not every possible frame rate can

be handled by the device. This value is as near as possible to the specified frame rate. Detailed description for this can be found in the user manual for CEBO LC (data acquisition; multi frame DAQ.

void startBufferedExternalTimedDataAcquisition(int frameCount)
Very similar to the method above, except that no frame rate is used the sample frame, but every time an external trigger has been detected one single frame is sampled. Acquisition is automatically stopped after the specified amount of frames has been sampled. Detailed description for this can be found n the user manual for CEBO LC (data acquisition; multi frame DAQ).

float startContinuousDataAcquisition(float frameRate, boolean externalStarted)
Starts a data acquisition without frame limit. The host must read the sampled data as fast as possible, otherwise buffer overflow is signaled and the process has been failed. Frames are sampled at the specified **frameRate**. If **externalStarted** is true, sampling starts at the first detected trigger event, immediately otherwise. The return value is the frame rate that is really used, as not every possible rates are possible.

In a multithreaded context, it is advised to start reading frames before calling this method.

More details about data acquisition can be foundn the user manual for CEBO LC (data acquisition; mult frame DAQ).

void startContinuousExternalTimedDataAcquisition()
Similar to the method above, but instead of a fixed frame rate, frames are sampled for every detected trigger. Detailed information about this can be found n the user manual for CEBO LC (data acquisition; mult frame DAQ).

void stopDataAcquisition()
Stops a currently active data acquisition. The device buffer is not modified calling this, so unread data can be fetched subsequently.

List<InputFrame> readBlocking(int frameCount)
When doing multi frame data acquisition, this is one method to read the sampled frames from device to host. This version will block the current thread until the specified amount

of frames has been read, there's no timeout. After the call returns without an exception, the returned list contains exactly the specified amount of InputFrames. The alternative to this call is described below.

List<InputFrame> readNonBlocking()
This method is similar to the one above, except that it always returns immediately, while trying to read as much frames as possible. The returned list contains all sampled InputFrames since the start of the data acquisition or the last frame read. Especially for high data amounts, this method should be called cyclically without to high delays.

InputFrame readFrame()
Read single frame as set up calling setupInputFrame() and return immediately. This cannot be called if multi frame data acquisition is active.

void setupOutputFrame(Output[] outputs)
Specifies which Outputs are involved in the next frame output process. There's no limit of the specified outputs, but no single instance can be used more than once. Valid outputs that can be added to the list are AnalogOutput and DigitalPort.

void writeFrame(OutputFrame frame)
Set all outputs that have been selected using setupOutputFrame() in a single call. Data to set on the respective outputs must be set in the given OutputFrame. The frame must be constructed using method createOutputFrame() of the same device instance were it is used.

OutputFrame createOutputFrame()
Create an OutputFrame instance that fits to the device metrics from which instance the method is called. The frame can than be filled with data and used calling writeFrame() of the same device instance.

List<AnalogInput> getSingleEndedInputs()
Return list of all single ended inputs.

List<AnalogInput> getDifferentialInputs()
Return list of differential analog inputs.

List<AnalogOutput> getAnalogOutputs()
Return list of the analog outputs.

List<DigitalPort> getDigitalPorts()
Return list of digital ports.

List<Counter> getCounters()
Return list of counters.

List<Trigger> getTriggers()
Return list of triggers.

List<CurrentSource> getCurrentSources()
Return list of current sources.

List<Led> getLeds()
Return list of LED's.

LibraryInterface

This class contains static functionality only. Its responsibility is to serve as interface to methods that are not bound to any device.

String getApiVersion()
Report the version number of the underlying CeboMsr API string, i.e. "1.0".

String getUsbBaseVersion()
Return the version number of the system interface USB layer string, i.e. "1.0".

List<Device> enumerate(DeviceType type)
The specified **type** parameter must be one of class DeviceType static members. The system is than scanned for known devices and subsequently filtered to meet the specified **type**. Devices that are already opened were skipped too. The returned list contains all candidates that are ready for use.

For each instance that should be used, open() must be called.

Starting a new enumeration invalidates the list from the previous invocation.

Python

The Python API is a thin layer on top of the CeboMsr API. It uses ctypes to bind itself to the dynamic link library.

Compatibility

The API has been tested with Python 2.7 and 3.2, both branches are supported.

External Dependencies

The API is based on ctypes, which is part of the standard Python library, no additional libraries are required.

Error Handling

Any logical or runtime error that is reported by the CeboMsr API is transformed to an exception. The following table lists all possible exception thrown by the Python implementation of the CeboMsr API:

Exception	Circumstances
AttributeError	If any of the used parameters has an invalid value in the current context.
IndexError	When using an invalid index. (Most of these are prevented due to the API design).
SystemError	Thrown if something is called which is not allowed in this stage.
IOError	Indicates an unexpected behavior like communication problems with the device.

All exceptions contain a textual description of the problem.

Basics

This section describes the first steps that are necessary to use the API. We start by showing a basic example that:

- searches for all known devices
- grabs the first available device
- opens the device
- resets its state
- closes the device

```
from CeboMsrApiPython import LibraryInterface, DeviceType

...

# Search for devices ...
devices = LibraryInterface.enumerate(DeviceType.All)

# If at least one has been found, use the first one ...
if (len(devices) > 0):
  device = devices[0]

  # Open device, nothing can be done without doing this.
  device.open()

  # After some processing, the device can be resetted to its
  # power up condition.
  device.resetDevice()

  # Finalize device usage, this free's up the device, so it can be used
  # again, including other applications.
  device.close()
```

Description

The first lines in the example imports the two necessary classes the source unit.
In the application flow, enumeration must be always the first task when using the API. It searches the system for known, not yet used devices and offers the caller a tuple of device instances. The parameter that is used calling enumerate() of LibraryInterface specifies which devices should be searched:
All known devices (DeviceType.All)
Device classes (e.g. DeviceType.Usb)

Specific types (e.g. DeviceType.CeboLC)
We search for all devices in the example above. The returned list contains candidates of type Device that can be used subsequently.
We use **len()** on the tuple to see if at least one device has been found. If a device has been found, the if branch is entered, were we continue by using the first device.
The call to open() flags the device for use. Any new enumeration (before calling close() on the same instance) will filter this instance. This is just a simple mechanism to prevent double access to the same device.
After the device has been opened, it can be used in any way. Method resetDevice() is shown here, because it has a superior role. It stops all active processing inside the device and resets all configuration values to their startup condition. It is not necessary to call it, but is a good way to reach a defined state i.e. in an error condition. It is implicitly invoked when calling open().
At the end of the current scope, method close() is called to signal API and system that the device is not used anymore. This is an important call, because leaving it in opened state will prevent it from getting enumerated again.

Single Value I/O

This section describes simple single input and output of values. It continues the example from section Basics, so an opened device instance is available. Single value I/O is described in the user guide CEBO-LC in detail.

```python
# Read digital port #0 and write result to digital port #1 (mirror pins).
# At first, we need references to both ports.
dp0 = device.getDigitalPorts()[0]
dp1 = device.getDigitalPorts()[1]

# Configure port first, all bits of digital port #0 as input (default)
# and all bits of digital port #1 as output.
dp1.setOutputEnableMask(0xff)

# Read from #0 ...
value = dp0.read()

# Write the value to #1 ...
dp1.write(value)

# Now some analog I/O, do it without any additional local references.
# Read single ended #0 input voltage ...
voltageValue = device.getSingleEndedInputs()[0].read()
```

```
# Write value to analog output #1
device.getAnalogOutputs()[1].write(voltageValue)
```

Description

If you want work with device peripherals, get instances using the various component access methods of class Device. If more then one method must be called, create a local reference, this reduces the amount of code to write. After you have an instance, invocations to its methods affects this specific peripheral.

The example outlines how to read all I/O's of digital port #0 and mirror the result on port #1. The first requirement to accomplish this is to define the direction of the individual I/O's. By default, all I/O's are configured to be inputs, so the example shows how to set all I/O's on port #1 as output, calling setOutputEnableMask() on its reference. All bits that are '1' in the specified mask signal the I/O to be an output. This is all you have to do for configuration.

The mirroring step is quite easy, read() the value from the instance that represents port #0 and store it in a local variable. The result is subsequently used calling write() on the instance the represents port #1.

The second half of the example outlines how to access peripherals without local copies or references. This requires fewer but longer lines of code but has no behavioral difference. Choose which style you prefer on your own.

The example continues doing direct single value I/O using analog peripherals. Like the digital port part, an input is mirrored to an output.

The call to read() returns the calibrated voltage value on this input as float. Calling write() on the AnalogOutput instance that represents analog output #1 using the returned voltage value will directly modify the real output on the device.

Single Frame Input

This section presents an example that samples a selection of inputs in a single process. The difference to single value read is not that much on the coding side, but there's a

large improvement when it comes to performance. Reading more than one input costs only an insignificant higher amount of time in comparison to single I/O.
Configuration of the individual peripherals is exactly the same, so setting the I/O direction if digital ports or set the input range on an analog input is almost identical.

```
# Prepare and fill the list of inputs to read.
inputs = [
  device.getSingleEndedInputs()[0],
  device.getSingleEndedInputs()[1],
  device.getDifferentialInputs()[1],
  device.getDigitalPorts()[0],
  device.getDigitalPorts()[1],
  device.getCounters()[0]
]

# Setup device with this selection.
device.setupInputFrame(inputs)

# Read the values multiple times and write them to the console.
for _ in range(100):
  # Read all inputs into the instance of InputFrame.
  inFrame = device.readFrame()

  # Write results to the console.
  print(
    ("DigitalPort #0: %d" % inFrame.getDigitalPort(0)) + ", " +
    ("DigitalPort #1: %d" % inFrame.getDigitalPort(1)) + ", " +
    ("SingleEnded #0: %.2f" % inFrame.getSingleEnded(0)) + " V, " +
    ("SingleEnded #1: %.2f" % inFrame.getSingleEnded(1)) + " V, " +
    ("Differential #1: %.2f" % inFrame.getDifferential(1)) + " V, " +
    ("Counter #0: %d" % inFrame.getCounter(0)))
```

Description

The primary work for frame input I/O is to setup the device with the list of inputs that are involved. An array which contains the inputs to sample is required.

The example does this in the upper part. It creates the array and add several inputs to it:

- Single ended analog input #0 and #1
- Differential analog input #1
- Digital ports #0 and #1
- Counter #0

This list is than used calling setupInputFrame(), which prepares the device with this setup. This is active until:

- A call to resetDevice()
- The device is closed
- Or a new list is specified

In the subsequent loop, the specified inputs are sampled in every loop cycle using method readFrame(). The call lasts until all inputs are sampled and returns the values immediately in an instance of class InputFrame.

This instance holds all sampled values and offers methods to request them, which is shown in the lower part of the example.

Single Frame Output

In this section, concurrent modification of outputs will be outlined. In comparison to single value I/O, this technique is very efficient when working with more than one single output. As you can see in the following example, using it is quite straightforward.

```python
# Write to analog out #1 and digital out #2 in one call.
# Prepare and set output selection.
outputs = [
  device.getDigitalPorts()[2],
  device.getAnalogOutputs()[1]
]

# Prepare device ...
device.setupOutputFrame(outputs)

# Create instance of OutputFrame. There's no direct construction, as this
# instance may vary between different device types.
outFrame = device.createOutputFrame()

# Write it to hardware, modify contents in every looping.
for i in range(100):
  outFrame.setAnalogOutput(1, float(3 * sin(float(i) / 100)))
  outFrame.setDigitalPort(2, i)

  device.writeFrame(outFrame)
```

Description

First of all, the example assumes that an opened instance of class Device is available, as well as that DigitalPort #2 is configured to be output.

Similar to the other direction, the device must be set up with a list of outputs. This setup is active until:

- A call to resetDevice()
- The device is closed
- Or a new list is set up

An array that contains the outputs to set is required. In the upper part of the example, you can see the creation of this array, adding outputs DigitalPort #2 and AnalogOutput #1 to it and activate the setup using setupOutputFrame().

The subsequent call to createOutputFrame() creates an instance of type OutputFrame, which fits to the device metrics. There's no other way to create this type.

In the loop below, every cycle modifies the values of the outputs specified during setup. This does not do anything other than storing the values inside the frame. The active modification of the outputs is done using writeFrame(), which transfers the values to the respective peripherals.

Multi Frame Input

This section describes how to read many frames at once, a very useful feature when you want sample inputs at a constant frequency or using an external trigger. The API offers very handy methods. The most problematic thing is to choose the right start method including its parameters.
The example below samples five different inputs at a constant rate of 300 Hz, 20 x 25 frames, completely cached inside the device buffer.

```
# Construct selection that contains inputs to read from.
inputs = [
  device.getSingleEndedInputs()[0],
  device.getSingleEndedInputs()[1],
  device.getDifferentialInputs()[1],
  device.getDigitalPorts()[0],
  device.getDigitalPorts()[1]
]

# Prepare device with this collection ...
device.setupInputFrame(inputs)

# Start sampling ...
device.startBufferedDataAcquisition(300, 20 * 25, False)

# Read 20 x 25 frames using blocked read,
# this function returns after *all* (25) requested frames are collected.
for _ in range(20):
  # Read 25 frames ...
  frames = device.readBlocking(25)

  # Write out the 1st one.
  inFrame = frames[0]
  print(
    ("DigitalPort #0: %d" % inFrame.getDigitalPort(0)) + ", " +
    ("DigitalPort #1: %d" % inFrame.getDigitalPort(1)) + ", " +
    ("SingleEnded #0: %.2f" % inFrame.getSingleEnded(0)) + " V, " +
    ("SingleEnded #1: %.2f" % inFrame.getSingleEnded(1)) + " V, " +
    ("Differential #1: %.2f" % inFrame.getDifferential(1)))

# Stop the DAQ.
device.stopDataAcquisition()
```

Description

The first lines in this example are similar to single frame example. The device is set up to sample the specified inputs into a single call. Single ended input #0 and #1, differential input #1 and digital ports #0 and #1 are used here.

The real data acquisition is than started calling startBufferedDataAcquition(). Choosing this method to start up, combined with the used parameters means the following:

- Data has to be completely stored in the device memory (buffered).
- Sampling is done using hardware timed capture at 300 Hz.
- The number of frames to capture is 20 x 25 = 500.
- No external trigger is necessary to initiate the process.

This method does not only configure the DAQ process, but start it as well. In the case of buffered mode, this is really uncritical. If you require continuous mode, a buffer overrun may occur shortly after starting the DAQ, so it is very important to either:

- Start to read the sampled frames immediately, as shown in the example.
- Use a second thread to read the frames, start this thread before the DAQ is started.

The example continues with a for loop where every cycle reads 25 frames and outputs the sampled values of the first frame in this block. The used method readBlocking() returns after the given amount of frames has been read and stalls the thread up to this point. Use readBlocking() with care, because it has two downsides:

- It blocks the access to the whole API due to internal thread locking mechanisms.
- If the specified amount of frames is too small, especially at higher frame rates, buffer overruns on the device side will occur, as the call and transfer overhead is too high.

It is a very convenient way to read a defined number of frames. The alternative is the use readNonBlocking(), which returns immediately with all captured frames at the moment of calling.

Both read methods return the tuple of captured frames. The example uses the first frame of this block (which is guaranteed to contain 25 frames in the example), and output the sampled values of all specified inputs to the console.
At the end of the example, DAQ is stopped using stopDataAcquisition().

Counter

The example below shows how to use a counter. To allow this using software, a wired connection between IO-0 and CNT is necessary. IO-0 is used to generate the events that the counter should count.

```python
# Create local copies, this shortens the source.
dp0 = device.getDigitalPorts()[0]
cnt = device.getCounters()[0]

# Set IO-0 as output.
dp0.setOutputEnableMask(0x01)

# Enable the counter.
cnt.setEnabled(True)
```

```
# Check the counters current value.
print("Counter before starting: %d" % cnt.read())

# Pulse IO-0 3 times.
for _ in range(3):
  dp0.write(0x01)
  dp0.write(0x00)

# Check the counters current value.
print("Counter should be 3: %d" % cnt.read())

# Reset and ...
cnt.reset()

# ... disable the counter.
cnt.setEnabled(False)

# Check the counters current value.
print("Counter should be 0: %d" % cnt.read())

# Pulse IO-0 3 times, the counter should ignore these pulses.
for _ in range(3):
  dp0.write(0x01)
  dp0.write(0x00)

# Check the counters current value.
print("Counter should still be 0: %d" % cnt.read())
```

Description

Like most of the previous examples, an instance of class Device is required. This can be retrieved using the procedure described here. To demonstrate the counter without external peripherals, the software controls the events for the counter as well. This extends the example to around twice its size, but its still easy to understand.

To reduce the amount of code, dp0 and cnt are constructed as local copies of both the first DigitalPort and the Counter.

The LSB of digital port #0 represents IO-0. To modify IO-0 via software, the example continues to set this I/O as output using method setOutputEnableMask().

Counters are disabled by default, so the next required task is to enable it, otherwise no event will be detected. The example does this by calling setEnabled() of the Counter instance.

The first counter value that is printed out will be zero, which is the default value after startup or reset.

The counter reacts on every rising edge. The example shows this by pulsing IO-0 three times, setting its level high and than low in every loop cycle. The result is than printed to the console. It is expected to be three, based on the pulses in the previous loop (If not, verify the wired connection between IO-0 and CNT).

The value of the counter is than set to zero calling its reset() method. In addition, setEnabled() deactivates the counter to any event.

The remaining example code outlines this, by pulsing IO-0 three times again, but the console output shows that now flanks have been counted in this state.

Trigger

The following example is much longer than the previous, but outlines various new things:

- Working with multiple devices.
- Use different multi frame modes.
- Show how to use triggers in input and output mode.

You will need two devices for this example to run, as both devices are chained together using triggers. The first device acts as master, while the second device is the slave. Every time, the master samples a frame, an output trigger is generated, while the slave captures its frame if this trigger has been raised.

Devices must be connected to each other using two wires. Ground (GND) to ground and trigger (TRG) to trigger.

TIP #1: This example can easily be extended to support more than one slave, you only have to set up each slave the same way as the slave in the example.

TIP #2: At high bandwidth, it may be necessary to put the individual readNonBlocking()

calls to separate threads. Otherwise reading the data from one device may last as long as the device side buffer on the second device will need to overflow.

The description is located below the example.

```
def dumpFrames(device, frames):
    """
    Write frames to console.
    device: Device string.
    frames: Frame to dump.
    """
    for f in frames:
        print("%s: se#0: %.2f V, dp#0: %d" % (
            device, f.getSingleEnded(0), f.getDigitalPort(0)))

def runDataAcquisition(master, slave):
    """
    Start DAQ, read frames and output the results.
    master: Master device.
    slave: Slave device.
    """
    # The slave must be started first, as it reacts on the master's
    # trigger. Continuous DAQ is used. Timed using an external trigger.
    slave.startContinuousExternalTimedDataAcquisition()

    # The master uses hardware timed DAQ, continuous at 50 Hz.
    # The 'false' here signals that the process should start immediately.
    master.startContinuousDataAcquisition(50, False)

    # The example reads at least 10 frames from
    # both devices and subsequently output the samples.
    masterFrames, slaveFrames = 0, 0
    while (masterFrames < 10 or slaveFrames < 10):
        # Start by reading frames from master,
        # output it and increment counter.
        frames = master.readNonBlocking()
        dumpFrames("master", frames)
        masterFrames += len(frames)

        # Do the same with the slave.
        frames = slave.readNonBlocking()
        dumpFrames("slave", frames)
        slaveFrames += len(frames)

        # Don't poll to frequent, this would fully utilize one core.
        time.sleep(0.001)

    # Finished, gracefully stop DAQ.
    slave.stopDataAcquisition()
```

```python
    master.stopDataAcquisition()

def configure(master, slave):
    """
    Both devices are fully configured here.
    master: Master device.
    slave: Slave device.
    """
    # The trigger for the master must be set to alternating output.
    master.getTriggers()[0].setConfig(
        Trigger.TriggerConfig.OutputAlternating)

    # The slave's trigger must be set to alternating as well.
    slave.getTriggers()[0].setConfig(
        Trigger.TriggerConfig.InputAlternating)

    # Both devices now gets configured to the same input frame layout.
    master.setupInputFrame([
        master.getSingleEndedInputs()[0],
        master.getDigitalPorts()[0]
    ])

    # ... slave
    slave.setupInputFrame([
        slave.getSingleEndedInputs()[0],
        slave.getDigitalPorts()[0]
    ])

def main():
    """
    The examples main method.
    """
    try:
        # Search for devices ...
        devices = LibraryInterface.enumerate(DeviceType.All)
        if (len(devices) != 2):
            print("Exactly two devices are required.")
            return

        # As both devices can act as master,
        # we simply use the first one for this role.
        master = devices[0]
        master.open()
        print("Master: " + master.getSerialNumber() +
                "@" + master.getIdentifier())

        # ... and the second as slave.
        slave = devices[1]
        slave.open()
        print("Slave: " + slave.getSerialNumber() +
                "@" + slave.getIdentifier())
        # Configure both master and slave ...
        configure(master, slave)

        # ... and do the DAQ.
        runDataAcquisition(master, slave)

        # Close both.
        master.close()
        slave.close()
    except Exception as e:
        print(e)

if (__name__ == "__main__"):
    main()
```

Description

Read the example bottom up, starting in the **runExample()** function. Nothing really new in comparison to the previous examples, except that two devices are used concurrently. Both devices gets opened and a short information about master and slave is printed to the console.

What follows is the configuration, which is shown in function **configure()**. Trigger #0 of the master device is set to output, alternating. This means, every time, the master captures a frame, its first trigger toggles the level.

Trigger #0 of the slave is configured to work as input, alternating as well. So the slave captures a frame if its first trigger detects that the level has been toggled.

That's all for the trigger configuration. Function **configure()** completes the process by setting up a frame using single ended input #0 and digital port #0 as described in detail the user guide CEBO-LC.

Returned to **runExample()**, method **runDataAcquisition()** is invoked, which shows how data from both master and slave is read. The most important part here is, how both DAQ processes are started. The slave is started first, otherwise he may miss one or more events. The slave is set to sample frames every time a trigger has been detected, without any count limits. This is done calling startContinuousExternalTimedDataAcquisition().

The master's DAQ is than started calling startContinuousDataAcquisition(), which means, no frame count limitation at a specific frequency. The example uses a very low frequency, 50 Hz, and starts immediately (By using **false** for parameter **externalStarted**).

At runtime, the master will than start to sample frames at the given 50 Hz, toggle its Trigger every time, while the slave captures a frame every time this event is received at his input trigger pin. Both capture frames synchronously.

The example continues by reading frames from both devices one after another until both have returned at least 10 frames. Captured values are written to the console using function **dumpFrames()**. The loop contains a 1 ms sleep, otherwise the usage of one CPU core would go to 100%, which is never a good idea.

After the frames have been read, DAQ is stopped on both devices, calling stopDataAcquisition(). The program returns to **runExample()** and close() both devices.

Info

This section simply outlines what information you can get from the API about the API itself, the device and its peripherals.

```python
# Put out some device specific information.
print("Device type: %s" % device.getDeviceType().getName())
print("USB base ver: %s" % LibraryInterface.getUsbBaseVersion())
print("API vers: %s" % LibraryInterface.getApiVersion())
print("Firmware ver: %s" % device.getFirmwareVersion())
print("Identifier: %s" % device.getIdentifier())
print("Serial number: %s" % device.getSerialNumber())
print("Temperature: %.2f" % device.getTemperature())

# Get the real current of the reference current sources.
for source in device.getCurrentSources():
  print("Reference current of %s: %.2f uA" % (
    source.getName(),
    source.getReferenceCurrent()))

# Retrieve information about single ended input #0.
# This works on all analog inputs and outputs.
se0 = device.getSingleEndedInputs()[0]
print("Info for single ended input %s:" % se0.getName())
print("  Min. ICD: %d us" % se0.getMinInterChannelDelay())
print("  Current range: %.2f V to %.2f V" % (
  se0.getRange().getMinValue(),
  se0.getRange().getMaxValue()))
print("  Supported ranges:")
for i in range(len(se0.getSupportedRanges())):
  rng = se0.getSupportedRanges()[i]
  print("    Range #%d range: %.2f V to %.2f V, Def. ICD: %d us" % (
    i,
    rng.getMinValue(),
    rng.getMaxValue(),
    se0.getDefaultInterChannelDelay(rng)))

# Get info for digital port #1.
dp1 = device.getDigitalPorts()[1]
print("Info for digital port %s:" % dp1.getName())
print("  Count of I/O's: %d" % dp1.getIoCount())
for i in range(dp1.getIoCount()):
  print("    I/O #%d: %s" % (i, dp1.getIoName(i)))
```

Description

As most of the previous examples, this assumes an opened device as well. How to do this is described here.

The block in the upper part of the example writes any API or device specific information to the console. The printed information is self-explanatory.

The first loop iterates over all CurrentSources of the connected device. For every source, its real current value is printed out. These values were determined during device calibration.

In the following block, information about single ended input #0 is printed out, which is the active range setting, as well as all ranges that are valid for this port. All ranges report their lower and upper bound using methods getMinValue() and getMaxValue(). This can be done for every AnalogInput and AnalogOutput.

The last part of the example prints information about digital port #1. The value retrieved by getIoCount() is the number of I/O's that can be accessed by this DigitalPort. This may vary between the individual ports a device has. The names of all its single I/O's are printed as well.

Class Reference

The class reference is a complete but short overview of all classes and their methods used in the Python API. It does not outline how the components have to be used.

The sections parallel to this topic show many practical examples and should be the first you have to read to understand and use the API. A good starting point is this topic.

DeviceType

This class is an enumeration of device types and device classes. Its static instances can be used to control the enumeration process. Besides that, each device reports its class using this type in method getDeviceType().

All
This DeviceType includes all known devices, independent which bus type is used. In the current stage, this equals the following instance, Usb.

Usb
By using this instance of type DeviceType in the enumeration process, all known devices connected via USB are reported.

CeboLC
This instance of type DeviceType must be specified if CEBO LC devices should be searched. In addition, getDeviceType() of Device returns this if the device is of this type.

CeboStick
This instance must be specified if CEBO STICK devices should be searched. In addition, getDeviceType() of Device returns this if the device is of this type.

getName()
Returns the name as string, e.g. "CeboLC" for the CeboLC instance.

AnalogInput

This class is the host side interface to any analog input, so you have to use its methods to request or modify the peripheral that this instance is assigned to. Get an instance of this class by calling either getSingleEndedInputs() or getDifferentialInputs() of the corresponding Device instance.

getSupportedRanges()
Returns the Ranges this input supports as tuple. You can use each of these calling setParameters().

getDefaultInterChannelDelay(range)
Return the default interchannel delay at the specified Range in microseconds.

getMinInterChannelDelay()
Return the minimal interchannel delay for this input in microseconds.

setParameters(range[, interChannelDelay])
Set Range for this input. If specified, the interchannel delay in microseconds is adjusted manually, otherwise the default for this range is used. The returned value is the interchannel delay that is really used (as not all specified values can be handled by the hardware).

getRange()
Returns active Range.

getInterChannelDelay()
Returns active interchannel delay in microseconds.

read()
Returns voltage as floating point value from input directly (more: user guide CEBO-LC, data acquisition, single value I/O).

getName()
Returns the name of the input.

AnalogOutput

This class represents an analog output. You can get an instance of this class calling getAnalogOutputs() from the corresponding Device.

getSupportedRanges()
Returns the list of Ranges this input supports as tuple. You can use each of these calling setParameters().

setParameters(range)
Sets the Range on the analog output.

getRange()
Returns active Range.

write(value)
Directly set the specified voltage value on the output.

getName()
Returns the name of the output.

DigitalPort

This class is the interface to work with digital ports. Retrieve instances calling getDigitalPorts() of the respective Device.

setOutputEnableMask(mask)
Set bitwise mask that defines which of the I/O's on the specified port are input and output. A bit of value 1 defines the specific I/O as output, e.g. mask = 0x03 means that I/O 0 and 1 are set to output, while I/O 2 to n are inputs.

getIoCount()
Returns the count of I/O's of the specific port.

read()
Read the state of the I/O's of the port.

write(value)
Modify the output I/O's of the specified port.

getName()
Returns the name of the port.

getIoName(io)
Returns the name of the I/O as specified by parameter io. The range of io is 0 <= io < getIoCount().

Counter

Interface class to counters inside the device. Class instances can be retrieved calling getCounters() of the specific Device.

reset()
Reset the counter to value 0.

setEnabled(enabled)
Enable or disable the counter using a boolean expression. A disabled counter stays at the last value.

isEnabled()
Return whether the counter is enabled or not.

setConfig(counterConfig)
Defines the counter behavior by using one of the constants specified in **Counter.CounterConfig**, listed in the following table.

getConfig()
Returns the current setup. One of the constants in enumerator

Counter.CounterConfig.
CounterConfig

Edge	Comment
RisingEdge	Counter event is rising edge.
FallingEdge	Counter event is falling edge
Alternating	Counter event is both rising and falling edges.

read()
Read the current value of the counter.

getName()
Returns the name of the counter.

Trigger

Class that represents a trigger inside the device. Instances can be retrieved by calling getTriggers() of the respective Device.

setEnabled(enabled)
Enable or disable the trigger using a boolean expression.

isEnabled()
Return whether the trigger is enabled or not.

setConfig(triggerConfig)
Defines the trigger behavior by using one of the constants specified in **Trigger.TriggerConfig**, listed in the following table

getConfig()
Returns the current setup. One of the constants in enumerator **Trigger.TriggerConfig**.

TriggerConfig

Pulse or Edge	Circumstances
OutputPulse	Trigger is output. Every trigger-event generates a positive pulse.
OutputAlternating	Trigger is output. Every trigger-event toggles the level.
InputRisingEdge	Trigger is input, reacts on a rising edge.
InputFallingEdge	Trigger is input, reacts on a falling edge.

Pulse or Edge	Circumstances
InputAlternating	Trigger is input, reacts on rising and falling edge.

getName()
Returns the name of the trigger.

Range

Use when handling range settings. Valid setting for an AnalogInput or AnalogOutput can be retrieved using their getSupportedRanges() method.

getMinValue()
Returns the lower voltage of the specific range as floating point.

getMaxValue()
Returns the upper voltage of the specific range as floating point.

Led

Interface to the LED's on the board. Instances are accessed calling getLeds() of the corresponding Device.

setEnabled(enabled)
Enable or disable the LED using a boolean expression.

getName()
Returns the name of the LED.

CurrentSource

Class that represents the interface to the Fixed Current Outputs. Instances can be retrieved using getCurrentSources() of the respective Device.

getReferenceCurrent()
Returns the actual value of the Fixed Current Output, which is determined during manufacturing process and stored in onboard flash. The returned value is given in micro ampere.

getName()

Returns the name of the current source.

InputFrame

The input frame is a data class which stores measured samples from all inputs a device has (and which meet the frame concept). All samples inside a single frame are captured in a very short time span (which depends on the underlying device). Only values that have been selected using setupInputFrame() before sampling the frame are valid, the other are set to 0 by default.

getSingleEnded(index)
Return the voltage value of single ended analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

getDifferential(index)
Return the voltage value of differential analog input using the specified **index** at the moment the frame was sampled. This value is calibrated.

getDigitalPort(index)
Return the I/O state of the digital port indicated by the given **index** at the moment the frame has been sampled.

getTrigger(index)
Return the state of the Trigger as specified by the given **index** in the moment the frame has been sampled.

getCounter(index)
Return the value of the Counter as specified by the given **index**.

OutputFrame

This class stores all values the should be set to the outputs as specified using setupOutputFrame(). Calling one of its methods does not affect the hardware. This is done when calling writeFrame(). An OutputFrame instance can't be created directly, as its metrics depends on the underlying device. The instance can retrieved calling createOutputFrame() of the device instance that should be used.

setDigitalPort(index, value)
Set the bit mask for the DigitalPort as specified using **index**. Only bits set to output using setOutputEnableMask() are affected.

setAnalogOutput(index, value)
Define the voltage value that should be set to the AnalogOutput as specified by the given **index**. The value will be calibrated before it is active.

Device

This is the primary class of the framework. Each instance corresponds to one physical device. The class is not intended to be instanced directly. Use the method enumerate() of the class LibraryInterface to access instances. Many of the methods are related to the device itself. Besides that, instances to all device peripherals can be accessed.

open()
This method must be called before doing anything with the device. Internally a communication to the device is constructed and several initialization is done. Various device and firmware specific constants are read and method resetDevice() is called. To guarantee compatibility between API and device, if the major number of the device firmware is higher than the major number of the API, open() will fail, a newer version of the API must be used with this device.

close()
This closes the connection to the device, frees all internal resources and allows the device to be used by others (in the application instance and in different applications). Calling close will never throw an exception, it will always fail silently.

resetDevice()
Calling this will stop any hardware controlled processing and reset the device to its power up settings. This is usually necessary if errors occur (Except logic or communication errors). Invoking this method in a multithreaded context should be done with extra care.

resetPeripherals(mask)

Resets specific peripherals on the device side. Parameter mask specifies which elements using a bit mask. The bit mask must be constructed using the following flags, which will be enhanced in future versions:

int Device.FlagResetInputFifo	Clears the FIFO used during data acquisition and its error flags.

getIdentifier()

This returns a unique identifier to the device which is constructed using the physical connection properties. It is guaranteed to be unique at runtime, as well as constant between reboots of the operating system (except updates to the operating system change the behavior how physical properties are enumerated).
Be aware: Plugging a device to a different location (e.g. different USB port) will change this specifier, it is not device dependent. Use getSerialNumber() in this case.
This method can be called in front of invoking open().

getDeviceType()

Returns the specific type of the device that is bound to the instance. This is one of class DeviceType static members.
This method can be called before invoking open().

getFirmwareVersion()

Returns the firmware version encoded as string, e.g. "1.0".

getSerialNumber()

Returns the device serial number as string. This string is unique for each device.

getTemperature()

Returns current temperature in °C of the device.

void setWatchdogTimeout(timeout)

Sets the watchdog timeout in 250ms steps. If enabled, the device reboots if no data transfer has been done in the given time frame. Default value is 0xffffffff, which disables this feature.

getWatchdogTimeout()

Read back current watchdog timeout value.

calculateMaxBufferedInputFrames()
Depending on the specified inputs the device site buffer can store a limited amount of frames. This method calculates the count of frames that will fit into this buffer using the current frame setup, which is last set using setupInputFrame(). Modifying the frame setup invalidates this value and must be updated invoking this method again. This value is primary intended to be used in context with startBufferedDataAcquisition().

setupInputFrame(inputs)
When doing any form of frame based data acquisition, which is described in the user guide CEBO-LC, this method must be used to select the inputs to be read. There's no limit which inputs can be selected, but every input can only be specified once. This method can only be called if no multi frame based data acquisition is active.
Any subsequent frame based call is affected by this setup.
The input list can contain any instance of AnalogInput, DigitalPort, Counter or Trigger.

startBufferedDataAcquisition(frameRate, frameCount, externalStarted)
Starts a buffered data acquisition at the specified **frameRate** for exactly **frameCount** frames. If **externalStarted** is true, the acquisition is started at the moment an external trigger event has been detected, immediately otherwise. Parameter **frameRate** must be equal or smaller than the value reported by calculateMaxBufferedInputFrames(). The returned value is the frame rate that is really used, as not every possible frame rate can be handled by the device. This value is as near as possible to the specified frame rate. Detailed description for this can be found in the user guide CEBO-LC.

startBufferedExternalTimedDataAcquisition(frameCount)
Very similar to the method above, except that no frame rate is used the sample frame, but every time an external trigger has been detected one single frame is sampled. Acquisition is automatically stopped after the specified amount of frames has been sampled. Detailed description for this can be found in the user guide CEBO-LC.

startContinuousDataAcquisition(frameRate, externalStarted)
Starts a data acquisition without frame limit. The host must read the sampled data as fast as possible, otherwise buffer overflow is signaled and the process has been failed.

Frames are sampled at the specified **frameRate**. If **externalStarted** is true, sampling starts at the first detected trigger event, immediately otherwise. The return value is the frame rate that is really used, as not every possible rates are possible.
In a multithreaded context, it is advised to start reading frames before calling this method.
More details about data acquisition can be found in the user guide CEBO-LC.

startContinuousExternalTimedDataAcquisition()
Similar to the method above, but instead of a fixed frame rate, frames are sampled for every detected trigger. Detailed information about this can be found in the user guide CEBO-LC.

stopDataAcquisition()
Stops a currently active data acquisition. The device buffer is not modified calling this, so unread data can be fetched subsequently.

readBlocking(frameCount)
When doing multi frame data acquisition, this is one method to read the sampled frames from device to host. This version will block the current thread until the specified amount of frames has been read, there's no timeout. After the call returns without an exception, the returned tuple contains exactly the specified amount of InputFrames. The alternative to this call is described below.

readNonBlocking()
This method is similar to the one above, except that it always returns immediately, while trying to read as much frames as possible. The returned tuple contains all sampled InputFrames since the start of the data acquisition or the last frame read. Especially for high data amounts, this method should be called cyclically without to high delays.

readFrame()
Read single frame as set up calling setupInputFrame() and return immediately. This cannot be called if multi frame data acquisition is active.

setupOutputFrame(outputs)
Specifies which outputs are involved in the next frame output process. There's no limit of the specified outputs, but no single instance can be used more than once. Valid

outputs that can be added to the list are AnalogOutput and DigitalPort.

writeFrame(frame)
Set all outputs that have been selected using setupOutputFrame() in a single call. Data to set on the respective outputs must be set in the given OutputFrame. The frame must be constructed using method createOutputFrame() of the same device instance were it is used.

createOutputFrame()
Create an OutputFrame instance that fits to the device metrics from which instance the method is called. The frame can than be filled with data and used calling writeFrame() of the same device instance.

getSingleEndedInputs()
Return tuple of all single ended inputs.

getDifferentialInputs()
Return tuple of differential analog inputs.
getAnalogOutputs()
Return tuple of the analog outputs

getDigitalPorts()
Return tuple of digital ports.

getCounters()
Return tuple of counters.

getTriggers()
Return tuple of triggers.

getCurrentSources()
Return tuple of current sources.

getLeds()
Return tuple of LED's.

LibraryInterface

This class contains static functionality only. Its responsibility is to serve as interface to methods that are not bound to any device.

getApiVersion()
Report the version number of the underlying CeboMsr API string, i.e. "1.0".

getUsbBaseVersion()
Return the version number of the system interface USB layer string, i.e. "1.0".

enumerate(type)
The specified **type** parameter must be one of class DeviceType members. The system is than scanned for known devices and subsequently filtered to meet the specified **type**. Devices that are already opened were skipped too. The returned list contains all candidates that are ready for use.
For each instance that should be used, open() must be called.
Starting a new enumeration invalidates the list from the previous invocation.

Copyright Notice

Disclaimer

CESYS Gesellschaft für angewandte Mikroelektronik mbH
Zeppelinstrasse 6a
D - 91074 Herzogenaurach
Germany

Revision history

V1.0		Initial release.
V1.1	April 28, 2014	Layout, Header, Footer modified
V1.2	Sep 26, 2017	LabView & ProfiLab removed. WatchDog functionality added

Table of contents