User Manual

日本語

翻訳: アドバン オートメーション株式会社

HBM Common API 概要マニュアル

HBM Common API

Hottinger Brüel & Kjaer GmbH
Im Tiefen See 45
D-64239 Darmstadt
Tel. +49 6151 803-0
Fax +49 6151 803-9100
Email: info@hbm.com
Internet: www.hbm.com

アドバン オートメーション株式会社
101-0047
東京都千代田区内神田1-9-5 SF内神田ビル 5 F
Tel. 03-5282-7047
Fax. 03-5282-0808
Email: info-advan@adv-auto.co.jp
Internet: https://adv-auto.co.jp/

DVS: A04174-6.1
HBM public
01.2021
2023年10月 初版

目次

1 概要

1.1 本書について

本書は、HBM Common APIの概要を説明する簡単なユーザマニュアルです。主な機能とワークフローについて説明します。すべての関数とプロパティの詳細については、このAPIとともに提供しているCHM形式のヘルプファイルを参照してください。コードのサンプルとAPIの簡単なデモについては、提供しているHBM.API.DemoProjectを参照してください。

このドキュメントのすべてのコードのサンプルはC#で記述されています。これらはデモンストレーションのみを目的としており、いかなる保証もありません。

1.2 .NET Framework

HBM Common APIはC#で記述されており、.NET Framework 4.0が必要です。

1.3 インストール

HBM Common APIは、単一のセットアップパッケージとして出荷されます。

このセットアップは任意のユーザが実行できますが、管理者権限で実行すると、HBM Device Scanが機能するよう、Windowsファイアウォールもセットアップされます(このセクションの以下を参照)。

デフォルトでは、セットアップはすべてのファイルをパブリックのドキュメントディレクトリ内のサブディレクトリ"HBM\HBM Common API"にコピーし(つまり、すべてのローカルユーザ用)、スタートメニュー(Windows 7)またはWindows 8.x/ Windows 10のタイル表示(すべてのアプリセクション、HBMグループ)からアクセスできるようになります。HBM.API.DemoProject内のプロジェクトをビルド可能にするには書き込みのアクセス権が必要であることに注意してください。

ここには、ドキュメント、APIバイナリ(一部のDLLはx86またはx64専用であることに注意してください)、およびC#デモプロジェクト(x86または任意のCPUとして構築可能)があります。

バイナリディレクトリには、HBM Device Scanのメッセージの到着を許可するようにWindowsファイアウォールを設定するためのバッチファイルがあります。開発者PCでは、管理者権限でセットアップを実行時にこれらのポートを開きます。それ以外の場合、スキャンが機能する前に、このバッチファイルを管理者権限で一度実行する必要があります。
別のファイアウォールを使用しているシステムでは、バッチファイルで行ったようにポートを開く必要があります。

MS Visual Studio™でデモプロジェクトをビルドすると、ビルド後イベントを使用して、必要なすべてのDLLがbinディレクトリにコピーされます(64ビットWindowsで任意のCPU構成を使用している場合はx64 DLL、それ以外の場合はx86 DLL)。APIアセンブリはすべて、Any CPUアーキテクチャに組み込まれています。

2 Common APIを使用した開発

2.1 説明

Common APIは、統合された拡張性かつ高速な方法でさまざまなデバイスファミリに計測機能を提供するモジュラDAQフレームワークです。また、カスタムセンサやHBMセンサデータベースのセンサを操作するためのsensor APIも含まれています。

Common APIは、デバイスファミリの違いを共通の同種インタフェースの背後に隠すため、あるハードウェアプラットフォームから別のハードウェアプラットフォームに切り替えるときにコードを書き直す必要性が軽減します。同じコードベースを再利用できます。

2.2 デバイスファミリ

Common APIの実際のバージョンは、次の3つの異なるHBMデバイスファミリをサポートしています。

デバイスファミリ名	説明
QuantumX / SomatXR	ユニバーサルデータ収集システム
PMX	工程管理/制御用計測アンプシステム
MGCplus	構造試験、耐久性試験、実験計測用システム。

2.3 コンポーネント

APIは、すべてのデバイスで共有される共通の関数を含む共通部分とデバイス固有の部分で構成され、これらは特定のデバイスファミリに固有であるため、特別なのデバイスファミリドライバが実装されます。

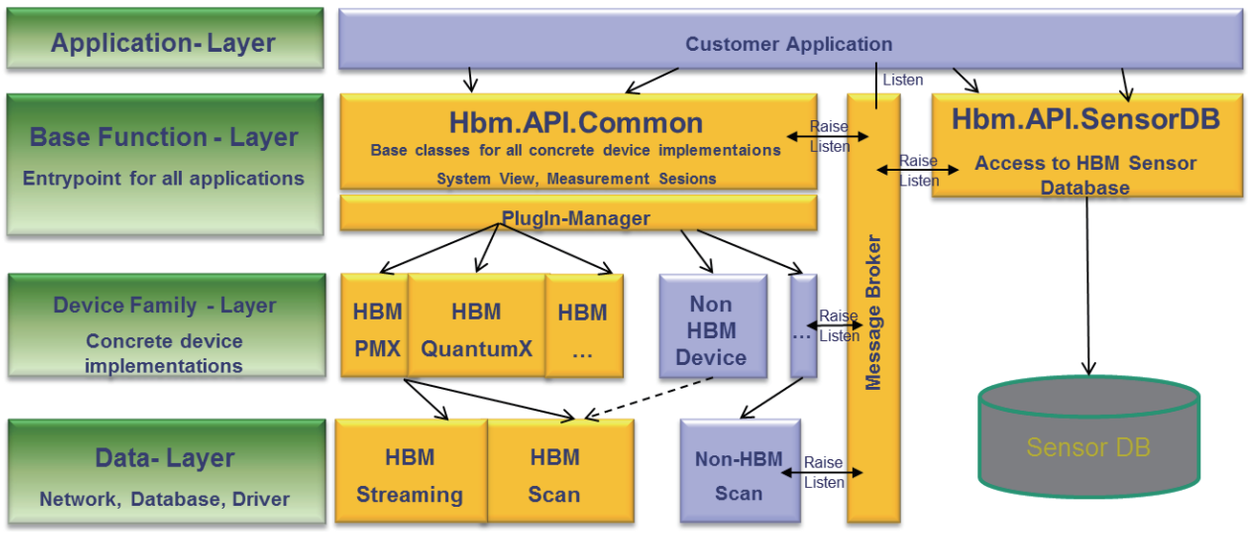
すべてのデバイスファミリドライバは、Common APIへのプラグインとして実装されます。したがって、Common APIは、追加のHBMデバイスファミリドライバや、サードパーティのデバイスファミリドライバによって簡単に拡張できます。

APIの主なアセンブリは次のとおりです:

コンポーネント	説明
Hbm.Api.Common	顧客向けアプリケーションの主要なエントリポイント。Common関数、オブジェクトモデル、基本クラス。
Hbm.Api.Scan	ネットワーク内のデバイスを検出するためのHBMデバイススキャンメカニズムが含まれています。内部でのみ使用し、顧客向けアプリケーションで直接使用しないでください。
Hbm.Api.Utils	ユーティリティとヘルパ関数。
Hbm.Api.SensorDB	センサデータベースAPI。
Hbm.Api.Logging	ロギングAPI。
Hbm.Api.QuantumX	QuantumX/ SomatXRデバイスファミリの実装。
Hbm.Api.Pmx	PMXデバイスファミリの実装。
Hbm.Api.Mgc	MGCデバイスファミリの実装。

高レベルの観点から見ると、Common APIは図1に示すように3つの異なるレイヤで構築されています。黄色の部分はHBM Common APIコンポーネントを表し、青色の部分はAPI上のアプリケーションなどの非HBMコンポーネントです。

図1: 高レベルのシステムアーキテクチャ



アセンブリHbm.Api.Commonには、最も重要なクラスDaqEnvironmentおよびDaqMeasurementと、名前空間Hbm.Api.Common.Entitiesの一般オブジェクトモデルが含まれています。

クラス	説明
DaqEnvironment	含まれるすべてのデバイスを含むシステム全体を表します。 シングルトンとして実装されているため、DaqEnvironment.GetInstance()を呼び出してオブジェクト参照を取得する必要があります。初めてインスタンスが作成されると、利用可能なすべての具体的なデバイス ファミリ実装(プラグイン)が列挙され、スキャンが開始されます。 Scan()を呼び出すときに、満たされた結果リストを確実に取得できるように、アプリケーション内でできるだけ早く DaqEnvironment.GetInstance()を呼び出します。 次のタスクを許可します: ネットワーク内のデバイスをスキャン デバイスに接続 デバイスから切断
DaqMeasurement	データ取得セッションを管理および表現します。 このクラスを使用して、連続計測を実行します。 次のタスクを許可します: 計測セッション用の信号を登録 計測セッションから信号を削除 DAQセッションを開始 DAQセッションを停止 登録した信号に計測値を分配

2.4 スレッドの安全性

APIのすべてのコンポーネントは並列アクセス用に設計されているため、スレッドの安全性が必要なすべての関数は、異なるスレッドからの同時アクセスに対して保護されます。

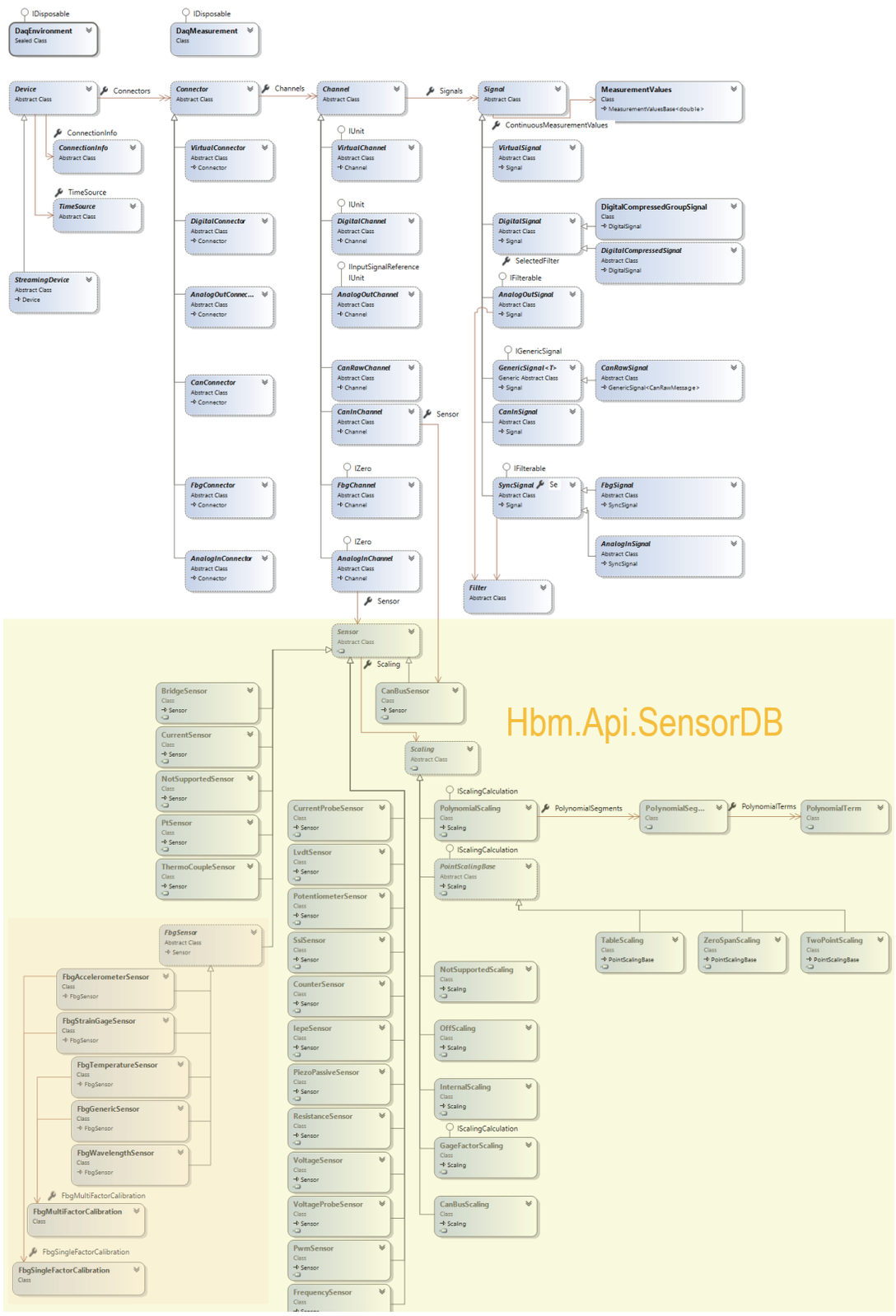
2.5 オブジェクトモデル

アセンブリ Hbm.Api.Common のオブジェクト モデルは、ストリーミングデバイスだけでなく通常のデバイスもサポートします。QuantumX/ SomatXR、MGC、またはPMXなどのデバイスの具体的な実装は、このオブジェクト モデルを実装して実現します。各プラグインデバイスドライバはこのモデルを拡張することもできますが、少なくともこのモデルはすべてのデバイスで共有されます。

ほとんどの場合、一般クラスDeviceの関数とプロパティを操作するだけで十分です。特別なデバイスファミリ固有の関数を使用する必要がある場合にのみ、この具体的なデバイス実装にキャストしてから、これらの追加機能を使用する必要があります。

デバイスのプロパティには、接続プロセス中に実際の値が入力されます。つまり、デバイスへの接続に成功すると、そのデバイスのすべてのプロパティを探索できるようになります。

図2: Commonオブジェクトモデル



2.6 メッセージシステム

Common APIには、中央メッセージングシステムが含まれています。このシステムは主に、利用可能なすべてのイベントを含む、静的な中央の到達可能なメッセージブローカに基づいています(図1を参照)。

関心のあるすべてのイベントのイベントハンドラをMessageBrokerクラスに登録します。

これは名前空間Hbm.Api.Common.Messagingにあり、内部および外部の通信に使用します。

例:

```
//MessageBroker handles all events of the common API.
MessageBroker.DeviceConnected += MessageBroker_DeviceConnected;

void MessageBroker_DeviceConnected( object sender, DeviceEventArgs e ) {
// Handle event here
}
```

メッセージシステムはすべてのイベントを非同期的に発生させます。つまり、各イベントハンドラは個別のスレッドで呼び出されます。イベントをGUIに反映する場合は、この事を考慮してください。そして、最初にGUIスレッドにマーシャリングする必要があります。

2.7 Visual Studioを使用

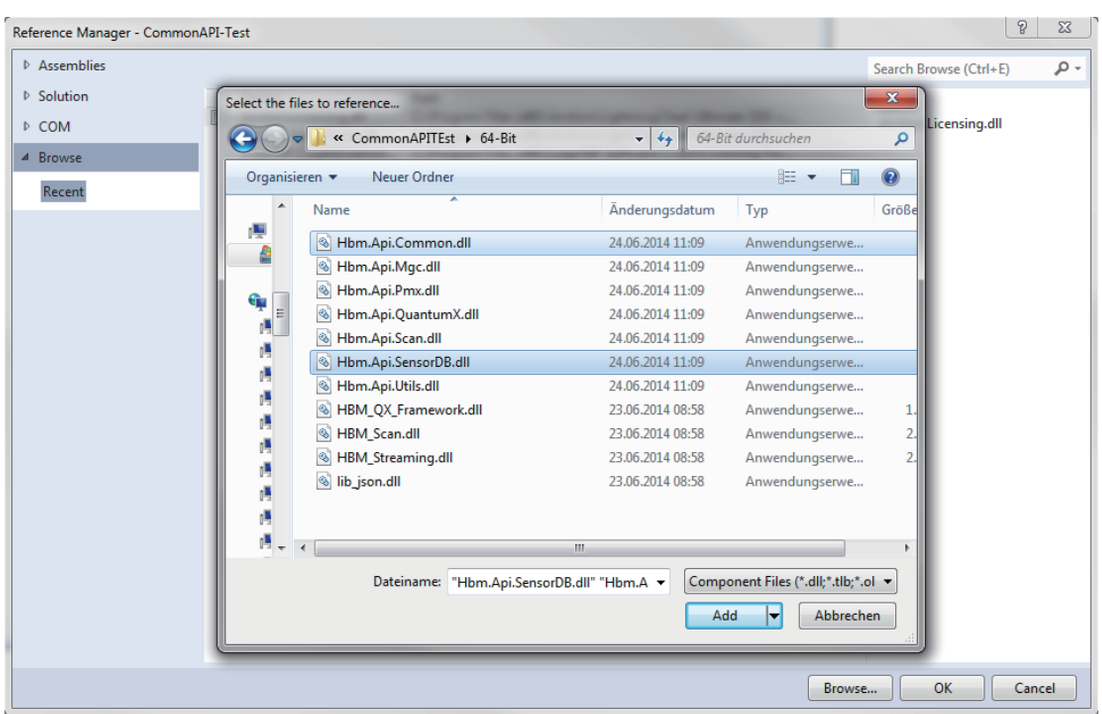
Common APIを使用するには、Visual Studioなどの開発環境と .NET Framework 4.0がインストールされている必要があります。

作業にはMicrosoft Visual Studio 2012以降を使用することをお勧めしますが、.NET 4をサポートする他のすべてのIDEでも問題なく動作するはずです。

2.7.1 参照

プロジェクトで*Reference Manager*を開き、以前にCommon APIアセンブリ(32/64ビット)をインストールしたフォルダを参照します。少なくとも、Hbm.Api.Common.dllおよびHbm.Api.SensorDB.dllへの参照が必要です。

図3: APIの参照



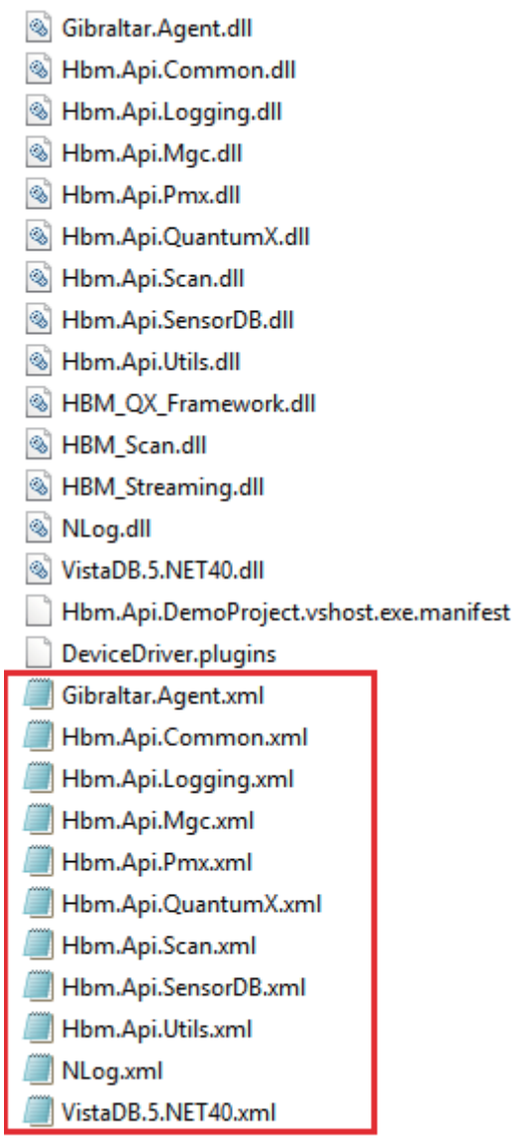
特定のデバイスの特別な関数を使用しない限り、必要なのはこれだけです。特定のデバイスに固有で、共通部分からアクセスできないプロパティや関数を使用したい場合は、それらのアセンブリ(Hbm.Api.QuantumX.dll など)への参照も必要になります。
通常の状況では、Hbm.Api.Scan.dllおよびHbm.Api.Utils.dllへの参照が必要になることはありません。

情報:
プロジェクトでどのような参照が必要かに関係なく、すべてのDLLファイル、DeviceDriver.pluginsファイル、すべてのスキルファイル、および完全なTEDSDefinitionディレクトリがメインのEXEファイルと同じ場所に必要であることに注意してください。

2.7.2 インテリセンス

Common API を使用した開発中の利便性を最大限に高めるには、インテリセンスの支援が不可欠です。インテリセンスを適切に動作させるには、Intellisense ヘルプを含む XMLファイル(図4を参照)が参照アセンブリと同じディレクトリに配置されていることを確認する必要があります。APIをインストールした直後、これらのファイルはすでに正しいフォルダに配置されていることがわかります。したがって、インテリセンスは追加設定なしに機能するはずです。

図4: インテリセンスファイル



2.8 メインワークフローとコンセプト

2.8.1 スキャン

現在のネットワーク内でデバイスを検索するために、一部のデバイスファミリはHBMスキャンプロトコルを実装しています。デバイスファミリがスキャンをサポートしているかどうかを確認するには、DeviceFamilyクラスのプロパティIsScanSupportedを調べることができます。

例:

```
MgcDeviceFamily mgcFamily = new MgcDeviceFamily();
bool hasScan = mgcFamily.IsScanSupported;
```

ネットワーク内のすべてのデバイスのスキャンを実行するには、DaqEnvironmentのScan()メソッドの一つを使用する必要があります。スキャンメカニズムは、初期化中にプラグインとして検出されるすべてのデバイスファミリを検索しますが、もちろん、スキャンメカニズムをサポートするデバイスファミリのみが対象となります。

例:

```
// Init (all families are enumerated, scan is initialized)
DaqEnvironment env = DaqEnvironment.GetInstance();

// Scan for all available devices
List<Device> foundDevices = env.Scan();
```

スキャン中に、ConnectionInfo、SerialNo、Model、Name、FirmwareVersionなどのデバイスの一部のプロパティが入力され、後で調べることができます。
スキャンメカニズムの性質上、- デバイスは数秒ごとにのみマルチキャストメッセージを送信します。- Scan()の呼び出し後、すべてのデバイスがまだ収集されていないことが発生する可能性があります。したがって、すべてのデバイスを確実に見つけるには、GetInstance()の最初の呼び出し後、しばらく(最大5秒)待つか、定期的に再スキャンする必要があります。

2.8.1 計測

APIは、計測値を取得する2つの方法を提供します。1つ目の方法は、単一のスナップショット値のみを取得する方法であり、2つ目の方法は、完全な連続計測セッションを設定することです。

連続計測セッションを設定するには、次の手順に従います:

1. 測定に使用するデバイスを選択します。
2. これらの選択したデバイスに接続します。スキャン結果からデバイスを選択することも、デバイスを手動で接続することもできます。デバイスに手動で接続する方法の詳細については、5.2を参照してください。
3. デバイスの同期に使用するタイムソースを設定します。詳細については、2.8.5を参照してください。
4. すべてのデバイスをパラメータ化します。センサを割り当てて設定します。少なくとも、計測する各信号のサンプルレートを設定し、信号をデバイスに割り当てます。
5. 新しいDaqMeasurementセッションを作成します。
6. 計測したい信号を計測セッションに追加します。
7. DAQセッションの準備を行います。
8. DAQセッションを開始します。
9. 計測値を循環バッファから各信号の計測値バッファに定期的に転送します。
10. DAQセッションを停止します。

手順7の準備アクションは、バッファ、スレッド、およびタイムスタンプの処理方法を初期化するために必要です。デフォルトでは、等間隔信号にはフェッチ間隔ごとに単一のタイムスタンプが1つだけ含まれます。不等間隔信号の場合、ステップ7で何を指定したかに関係なく、常に各値のタイムスタンプを取得します。通常、上記のパラメータをいじる必要はなく、パラメータなしのメソッドPrepareDaq()を使用するだけで、すべてがデフォルト値に設定されます。必要に応じてパラメータ、タイムスタンプ、取得スレッドの数、取得間隔などを変更する必要がある場合があります。ただし、デフォルト値を変更する場合は注意してください。関数PrepareDaq()のコード ドキュメント(CHM ファイル)を注意深く確認してください!

情報:
タイムスタンプは常に、計測開始以降の秒単位の倍精度値として提供します(DaqMeasuremnet.MeasurementStartTime、MeasurementStartUTCTime、およ MeasurmentStartSystemTimeを参照)。

ステップ9でデータを転送するには、適切な間隔内で何らかのタイマ制御ループでDaqMeasurement.FillMeasurementValuesを呼び出す必要があります。間隔の長さは、サンプルレートの速さ、使用可能なメモリの量、および測定データの処理方法によって異なります。
フェッチ間隔の選択が遅すぎると、基礎となる循環バッファがオーバーフローし、最も古い値が上書きされます。したがって、実行中の計測に関係するすべての信号について、Signal.MeasurementValues.BufferOverrunOccurredフラグとSignal.IsExcludedFromDaqフラグを常に確認する必要があります。
参加デバイスからデータを取得するための内部APIデータ取得間隔は、デフォルトで50ミリ秒に設定されています。この間隔は、データがデバイスから循環バッファに転送される頻度を決定します。したがって、データ取得タイマ間隔をその値未満に設定することは意味がありません。より高速なデータ取得間隔が必要な場合は、まずDaqMeasurementクラスのコンストラクタに低い値を指定して、内部APIデータ取得間隔を変更する必要があります(ステップ5)。

完成した例については、5.5を参照するか、提供されているデモプロジェクトを確認してください。

2.8.3 例外、エラー、警告

Common APIは、何か問題が発生したことを報告する3つの異なるレベルをサポートしています。致命的なコードエラーを示す旧知の例外、警告、エラー。

例外は、何か問題があり、目的のアクションを中止する必要がある場合に使用します。たとえば、DaqMeasurementクラスでPrepareDaq()を呼び出したときに、計測用に登録された信号がなかった場合などです。その場合、DaqNoSignalsAddedExceptionが発生し、PrepareDaq()は中止します。例外が発生するその他のケースとしては、メソッドのパラメータを何らかの方法で間違えた場合(ArgumentNullException)などがあります

例:

```
try {
        // Create new DAQ session
        DaqMeasurement daqSession = new DaqMeasurement();

        // Prepare DAQ session
        daqSession.PrepareDaq(); // Throws exception. No signals registered

} catch(DaqNoSignalsAddedException noSignalsEx) {
        // No signals registered for measurement
}
```

エラーと警告は主に、さらに問題が発生する可能性があり、一度に複数のデバイスに接続する場合など、すべての問題を一度に報告したい状況で使用されます。このシナリオでは、1つのデバイスに到達できない場合に例外をスローすることはあまり意味がありません。メソッドが中止され、その時点以降はデバイスが接続されなくなるからです。このような問題は、*List of Problems.*タイプの出力パラメータの形式で、エラーと警告としてまとめて報告されます。ErrorとWarningの両方のクラスは、Problemクラスから派生しているため、問題のリストにまとめることができます。

さらに、これらのメソッドは通常、問題のリストにエラーが含まれているかどうかを示すブール値の戻り値を持ちます。リストにエラーがない場合は、アクションが正常に実行されたことを確認できますが、リストにはまだ警告が含まれている可能性があります。警告は、たとえば、センサをデバイスに割り当てるときに、このデバイスがセンサの1つ以上のパラメータ(励起電圧など)を調整している可能性があることを示す可能性があります。
すべての接続および割り当て関数(AssignSensor、AssignConnector、AssignChannel、AssignSignalなど) は、この原則に従って動作します。

例: 複数のデバイスに接続

```
// Get instance of DaqEnvironment
DaqEnvironment env = DaqEnvironment.GetInstance();
// Scan for all available devices
List<Device> foundDevices = env.Scan();
// Connect to all devices
List<Problem> connectProblems;
bool isOk = env.Connect(foundDevices, out connectProblems);
if(isOk) {
       // No error occurred and all devices are connected,
       // check list for Warnings
} else {
       // Some or all devices are not connected
}
```

例: 信号の割り当て

```
// Change the name and sample rate of the first signal
// of the first channel of the first connector
Signal firstSignal = myDevice.Connectors[0].Channels[0].Signals[0];
// Change name
firstSignal.Name = "My signal name";
// Change sample rate
firstSignal.SampleRate = 19200;
// Assign changes back to device
List<Problem> assignProblems;
bool isOk = myDevice.AssignSignal(firstSignal, out assignProblems);
if(isOk) {
       // No error occurred, signal could be assigned
       // Check for possible warnings
       if(assignProblems.HasWarning()) {
              // Check each warning
       }
} else {
       // Signal could not be assigned to device
}
```

2.8.4 デバイスのパラメータ化

デバイスをパラメータ化するには、たとえば、センサを割り当てるには、次の手順に従う必要があります:
・デバイスに接続
・デバイスのオブジェクトモデル内の必要なプロパティを変更
・変更をデバイスに再度割り当て

情報: オブジェクトモデル内のプロパティを変更しても、ハードウェアデバイスの値は変更されません。変更を有効にするには、対応するAssignXYZ関数を呼び出す必要があります。

例:

```
// Change the name and sample rate of the first signal of the first channel of the
// first connector
Signal firstSignal = myDevice.Connectors[0].Channels[0].Signals[0];
// Change name
firstSignal.Name = "My signal name";
// Change sample rate
firstSignal.SampleRate = 19200;
// Assign changes back to device
List<Problem> problemsDuringAssign;
myDevice.AssignSignal(firstSignal, out problemsDuringAssign);
```

2.8.5 デバイスの同期

デバイスの同期により、デバイスのクロックが相互に可能な限り同期して実行されることが保証されます。CommonAPIは通常、タイムスタンプを使用して複数のデバイス間で同期した計測開始を実現するため、この場合は同じクロックのデバイスが必須です。

同じファミリの複数のデバイスを計測する場合、すべてのファミリがケーブルベースの同期方法(MGCとPMXの場合はそれぞれの同期ケーブル、QuantumX/ SomatXRの場合はFireWire ケーブルなど)をサポートしているため、通常はデバイスの同期に関する問題はありません。ただし、使用するデバイスファミリによっては、複数のサンプルレートで1つのデバイスのみを計測する場合でも、異なるサンプルレートの信号の最初のタイムスタンプが異なる場合があります(PMXなど)。

異なるデバイスファミリの複数のデバイスを計測する場合(例: PMXとQuantumX/ SomatXRを組み合わせた場合)、または同期ケーブルで配線できない(例: 距離が原因で)1つのファミリの複数のデバイスを計測する場合、次の問題が関連します:

- 同じサンプルレートであっても、異なるデバイスの計測値のタイムスタンプが異なる。
- 同じサンプルレートであっても、デバイスが異なれば計測値の数も異なります。

これは、デバイスのクロック/クォーツが100%同期して動作しないためです。タイムスタンプ同期測定を実行する場合、APIは常に、同じサンプルレートを使用する信号に対して同じ数の計測値を提供します。サンプルグループ内の"最も遅い"信号(例: 19200Hz)がサンプルグループ全体に配信される計測値の数を定義するため、一定の時間が経過すると、API内で内部バッファオーバーランが発生する可能性があります。適切なタイムソースを使用するか、非同期で計測を開始することにより、影響を最小限に抑えたり排除したりできます。

非同期で計測する場合、APIはサンプルレートグループ内の"最も遅い"信号の計測値の数だけではなく、常にすべての信号のすべての計測値を返すため、バッファオーバーランは発生しません。

デバイスを同期するために、APIはいくつかの"TimeSource"を提供しますが、これらは必ずしもすべてのデバイスでサポートされているわけではありません。

- AutoTimeSource: デバイスは内部同期方式を使用します。たとえば、FireWireケーブルを介して接続されている複数のQuantumX / SomatXR デバイスを計測する場合、または同期ケーブルを使用して多数のMGCまたはPMXデバイスを計測する場合に、このタイムソースが使用できます。
- NtpTimeSource: デバイスは、Network Time Protocol(NTP)を使用して、コンピュータネットワーク全体でクロックを特定のタイムサーバと同期します。ローカルエリアネットワークでは、1ミリ秒より優れたクロック精度を実現します。
  たとえば、QuantumX/ SomatXRとPMXまたはMGCデバイスを並行して測定する場合に、このタイムソースが使用できます。
- PtpTimeSource: デバイスは、Precision Time Protocol(PTP)を使用して、コンピュータネットワーク全体でクロックを同期します。ローカルエリアネットワークでは、マイクロ秒未満の範囲のクロック精度を実現します。
  たとえば、FireWire経由で接続されていないさまざまなQuantumX/ SomatXRデバイスを測定する場合に、このタイムソースが使用できます。
- EtherCatTimeSource: デバイスは、Ethernet for Controller and Automation Technology(EtherCat)を使用して、他のデバイスと同期します。サブマイクロ秒の範囲のクロック精度を実現します。
  たとえば、FireWire経由で接続されていないさまざまなQuantumX/ SomatXRデバイスを測定する場合に、このタイムソースが使用できます。

同期の品質を確認するには、Device.GetTimeSourceQuality(...)を呼び出します。これにより、現在のタイムソースへのオフセットと、使用しているTimeSourceの種類に応じた詳細情報を含む文字列が返されます。

次の例は、PMXおよびQuantumX/ SomatXRデバイスを特定のNTPサーバと同期する方法と、クロック間の同期の精度を確認する方法を示しています。

```
	// Assume we have a list of connected devices:
	// First one is e.g. a QuantumXDevice (_deviceList[0])
	// Second device is e.g. a PmxDevice (_deviceList[1])
	List<Problem> problemsDuringAssign;
	// Setup time source...
	foreach(Device device in _deviceList)
	{
		// Create an NTP time source with a valid NTP-Server
		device.TimeSource = new NtpTimeSource("172.19.160.111");
		// and assign it
		if (!device.AssignTimeSource(out problemsDuringAssign))
		{
			// error handling: e.g. check the problems here...
		}
	}
	// wait a while (let the devices synchronize this may take a moment...)
	System.Threading.Thread.Sleep(5000);
	foreach(Device device in _deviceList)
	{
		// check offset to NTP server in ms and further quality parameters
		double offset;
		string quality;
		device.GetTimeSourceQuality(out offset, out quality);
		Console.WriteLine(string.Format("Time offset in ms between NTP server and
device {0} = {1} ", device.Name, offset));
	}
	// Following similar output will be generated:
	// Time offset in ms between NTP server and device MX840_RT = -1,072
	// Time offset in ms between NTP server and device PMX_01 = -2,141
```

2.8.6 その他の機能

各デバイス実装は、そのデバイスに固有の特別な機能をサポートしているため、APIの共通部分には実装されていません。これらの特別な機能は、各デバイスのプロパティAdditionalFunctionの下にあります。

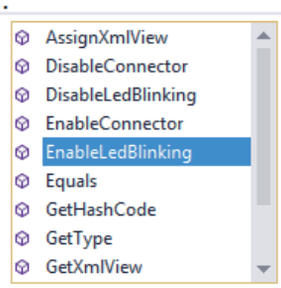
例:

```
// Init (all families are enumerated, scan is initialized)
DaqEnvironment env = DaqEnvironment.GetInstance();

// Scan for QuantumX devices only
List<Device> foundDevices = env.Scan(new List<string> { "QuantumX" });

// Cast to concrete QuantumX device
QuantumXDevice qxDevice = (QuantumXDevice) foundDevices[0];

qxDevice.AdditionalFeatures.
```



すべてのデバイス ファミリ(QuantumX /SomatXR、PMX、MGC)でサポートされている非常に重要な特殊機能の1つは、デバイスへのいわゆるダイレクトコマンドトンネルです。各デバイスファミリは、具体的な通信プロトコルに応じてわずかに異なる方法でこのメカニズムを実装しますが、デバイスに直接"話す"ことができるという共通点があります。例については、5.6を参照してください。

この通信パスを使用すると、現在Common APIで直接サポートされていないデバイスの関数を使用できるようになります。

情報: この機能を使用する場合、特定のデバイスの通信プロトコルを正確に知る必要があります。したがって、この機能を使用する場合は十分に注意してください。

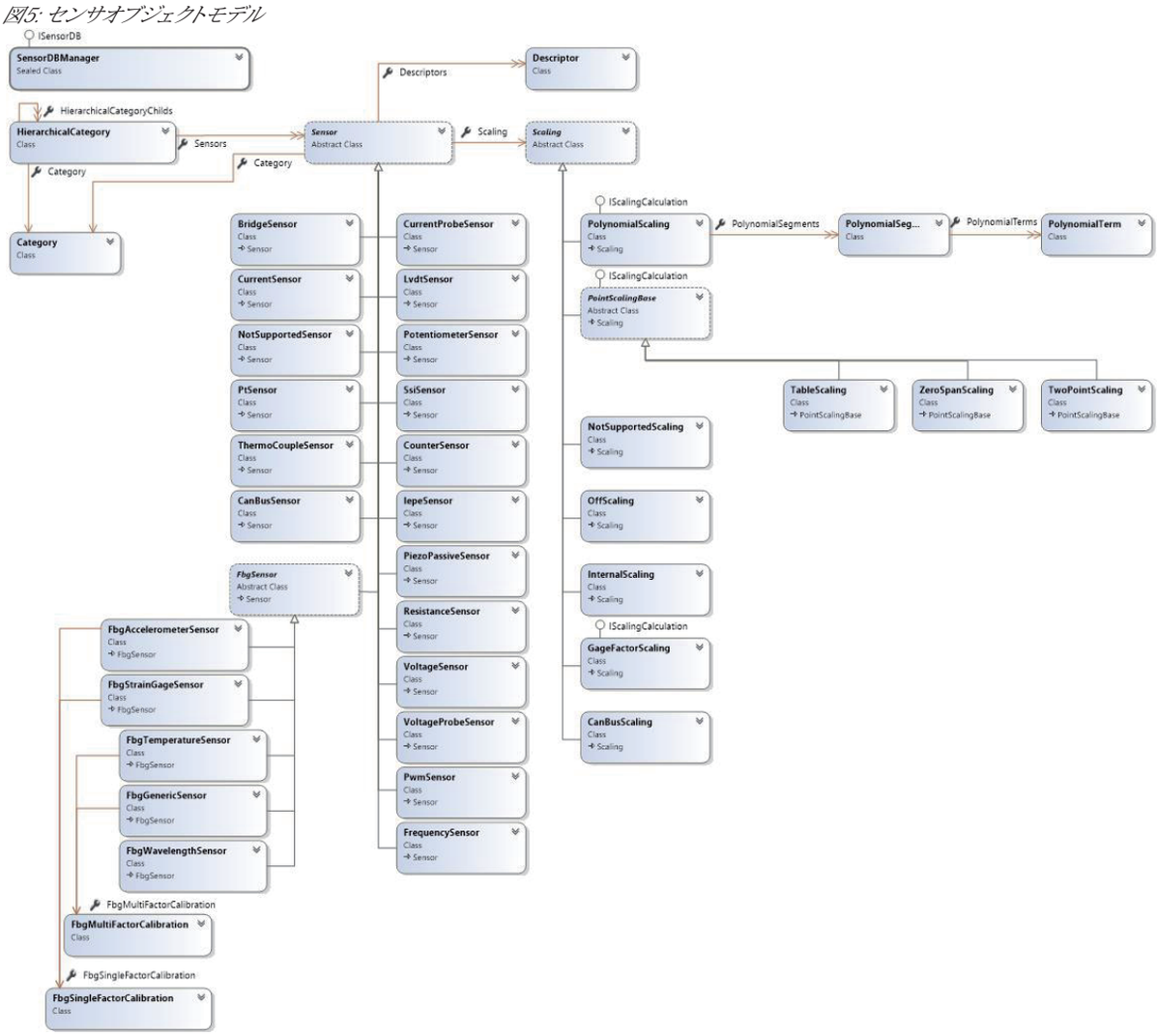
2.9 センサデータベースAPI

Common APIには、"読み取り専用"HBMセンサデータベース(ほとんどのHBMセンサが含まれます)と、独自のセンサの管理に使用できるユーザセンサデータベースの空のテンプレートが付属しています。
独自のセンサを作成してパラメータ化するために、APIは使用可能なすべてのセンサタイプとそのプロパティを定義するセンサオブジェクトモデルを提供します。SensorDBManagerを使用すると、APIで既存のセンサデータベースのセンサを管理できます。

2.9.1 センサオブジェクトモデル

センサモデルは、名前空間Hbm.Api.SensorDB.Entitiesにあるエンティティクラスによって表されます。

図5: センサオブジェクトモデル



センサオブジェクトモデルは主に次のクラスで構成されます:

コンポーネント	説明
Sensor	すべての物理センサの要約した基本クラス。
BridgeSensor	ブリッジセンサ。タイプはBridgeTypeで指定(Quarter、Half、Full)。
CanBusSensor	CANバス経由で接続されたセンサ。
CounterSensor	カウンタセンサ。
CurrentProbeSensor	電流プローブセンサ。
CurrentSensor	電流センサ。
FbgSensor	すべての光学センサの要約した基本クラス。
FbgAcceleratorSensor	光学式加速度センサ。
FbgStraingageSensor	光学式ひずみゲージセンサ。
FbgTemperatureSensor	光学式温度センサ。
FbgGenericSensor	光学式汎用センサ。
FbgWavelengthSensor	光電センサ。
FbgMultiFactorCalibration	光学センサの複数要素の校正情報。
FbgSingleFactorCalibration	光学センサの単一要素の校正情報。
FrequencySensor	加速度センサ。
IepeSensor	IEPEセンサ。
LvdtSensor	LVDTセンサ。
NotSupportedSensor	Common APIがデバイスから設定を読み取り、APIでサポートされていないセンサタイプ(IRIGセンサなど)を見つけたときに使用するセンサオブジェクト。
PiezoPassiveSensor	パッシブピエゾセンサ。
PotentiometerSensor	ポテンショメータ。
PtSensor	プラチナ温度センサ。
PwmSensor	パルス幅変調センサ。
ResistanceSensor	抵抗センサ。
SsiSensor	SSIセンサ。
ThermoCoupleSensor	サーモカプル。
VoltageProbeSensor	電圧プローブセンサ。
VoltageSensor	電圧センサ。
Scaling	すべてのスケーリングの要約した基本クラス。
GageFactorScaling	ゲージファクタスケーリング(K-Factor、ブリッジFactor)。
InternalScaling	内部デバイスのスケーリング。デバイス固有のスケーリングを使用する場合に設定。
NotSupportedScaling	デバイスがその設定でAPIでサポートしていないスケーリングを使用している場合、このスケーリングタイプを使用。これは、新しいスケーリングをサポートする新しいファームウェアをデバイスで使用しており、APIが古い日付のものである場合に発生する可能性がある。
OffScaling	デバイスのスケーリングを無効。
PolynomialScaling	多項式スケーリング。多項式セグメントのリストが含まれる。
TableScaling	複数直線のスケーリング(テーブルの点で定義された線)。
TwoPointScaling	2ポインスケーリング。
ZeroSpanScaling	ゼロスパンスケーリング。
PointScalingBase	すべてのポイントベースのスケーリング(ゼロスパン、2 ポイントなど)の要約した基本クラス。
PolynomialSegment	多項式スケーリングの単一セグメントを開始x値で定義。 PolynomialTermのリストが含まれる。
PolynomialTerm	多項式セグメントの項(係数、指数)を定義、例えば$5x^2$。
ScalingPoint	ポイントベースのスケーリング(テーブル、ゼロスパンなど)の単一スケーリングポイントをx値とy値で表す。
Descriptor	データシートや写真などのセンサの記述子を表す。

例: 2ポインスケーリングを使用したセンサオブジェクトの作成

```
// Create new sensor object
BridgeSensor newSensor = new BridgeSensor();
newSensor.UniqueName = "Sensor 200";
newSensor.Wiring = BridgeSensorWiring.SixWire;
newSensor.MinExcitationVoltage = 0.1M;
newSensor.MaxExcitationVoltage = 15;
newSensor.PreferredExcitationVoltage = 5;     // 5 Volt
newSensor.Impedance = 350;                    // 350 Ohm

// Create 2 point scaling
TwoPointScaling scaling2P = new TwoPointScaling();
scaling2P.EngineeringUnit = "kg";
scaling2P.MaxEngineeringRange = 200;
scaling2P.MinEngineeringRange = -200;

scaling2P.ElectricalP1 = 0;      // x1
scaling2P.EngineeringP1 = 0;     // y1
scaling2P.ElectricalP2 = 2;      // x2
scaling2P.EngineeringP2 = 200;   // y2

// Add scaling to sensor
newSensor.Scaling = scaling2P;
```

2.9.2 SensorDBManager

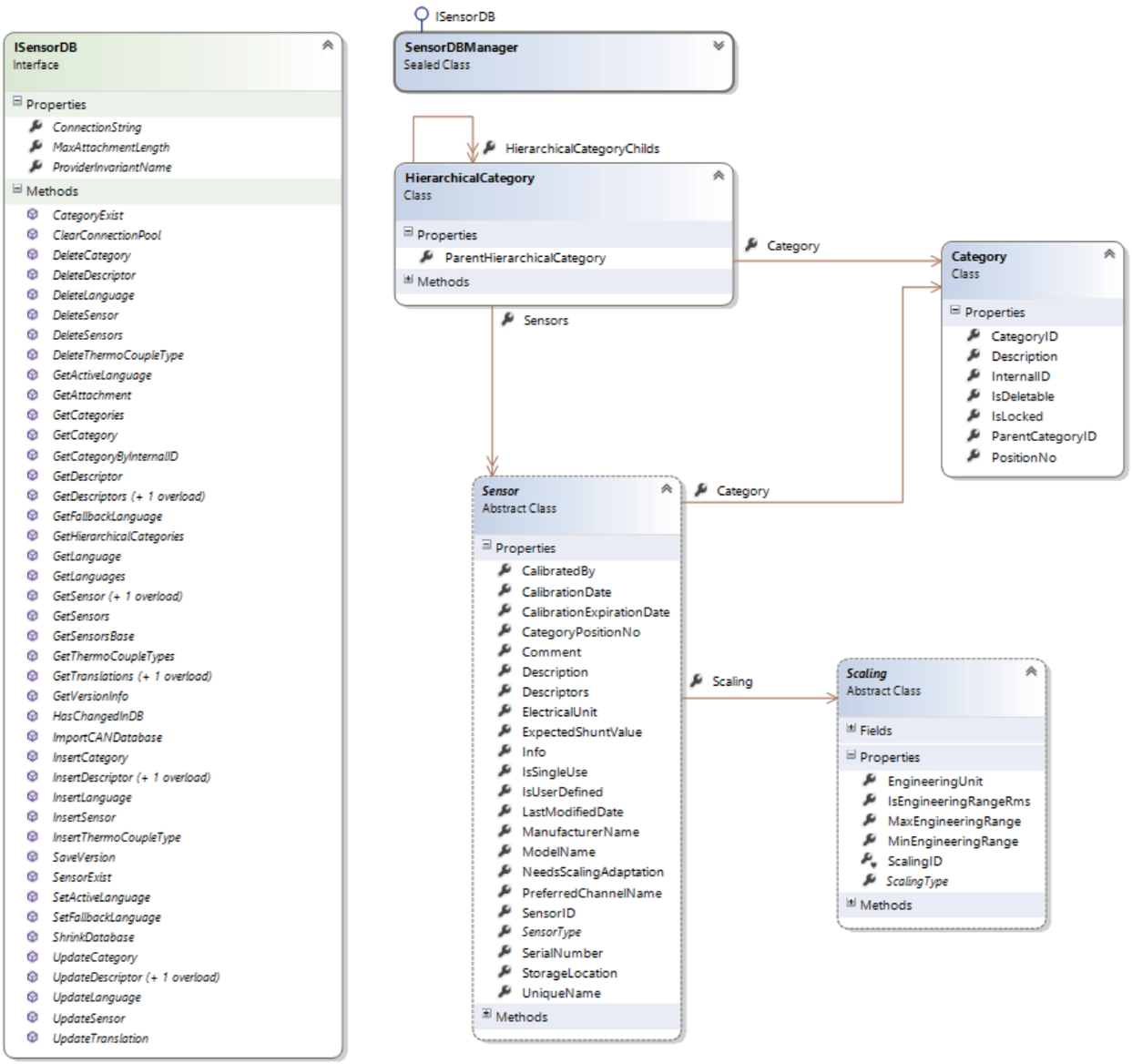
SensorDBManger(ISensorDBインタフェースを実装)は、センサを取得、挿入、更新、削除する機能と、センサデータベース内のセンサの階層グループ(カテゴリ)を管理する機能をさまざまな言語で提供します。

すべての光学センサ(FbgSensorから派生)をサポートしているわけではありません。

既存のセンサデータベースのすべてのカテゴリとセンサの完全なオブジェクト表現を取得するには、ISensorDB.GetHierarchicalCategoriesを呼び出します。関連するオブジェクトとの関係については、図6を参照してください。

センサデータベースにアクセスする方法を示す例は、第5章("実行手順")にあります。

図6 : 階層的なカテゴリ、センサの関係



SensorDBManagerを使用する場合は、次のヒントを考慮してください:

- すべてのセンサにはカテゴリが割り当てられている必要があります。センサを挿入する前に、カテゴリを保存する必要があります。
- すべてのセンサにスケーリングを割り当てる必要があります。具体的なスケーリング情報が利用できない場合は、ScalingTypeを"Internal"に設定する必要があります。
- 新しい言語を挿入した場合、既存の翻訳ごとに新しい言語の新しい翻訳レコードを作成する必要があります。API は、フォールバック言語の翻訳を使用して、新しい言語の翻訳レコードを作成します。

2.10 ロギング

Hbm.Api.Logging名前空間には、さまざまなログフレームワークを使用してメッセージをログに記録できるようにする関数が含まれています(これまではNLogとLoupeをサポートしていました)。

コンテキスト(データベース アクセス、デバイス通信など)に従ってログエントリをグループ化し、さまざまなLogContextを定義して、ログエントリが挿入される階層構造を構築できます。

ログを使用する場合は、基礎となるCommon APIがユーザのメッセージに加えてログのメッセージも記録することに注意してください。

無料のログファイル ビューアをダウンロードし、Loupeフレームワークを使用して作成されたバイナリ ログファイルを検査できます(http://www.gibraltarsoftware.comを参照)。NLogフレームワークを使用して作成されたログファイルは単なるテキストファイルです。

アプリケーションでのログインの使用方法を示す例は、第5章("実行手順")にあります。

図7: Logging名前空間の関連クラスと列挙

3 機能の概要

次の表は、Common APIのバージョンに関連するさまざまな機能の概要を示しています:

機能	QuantumX/ SomatXR	PMX	MGCplus
Device scan	✔	✔	✔(CP52のみ)
Measurement configuration	✔	✔	✔
Calibration certificates	✔	✔	利用不可
TimeSource configuration	✔	✔	✔
Sensor configuration	✔	✔	✔
Bridge	✔	✔	✔
CanBus	✔	利用不可	✔
Counter	✔	✔	✔
Current	✔	✔	✔
CurrentProbe	利用不可	利用不可	利用不可
Frequency	✔	✔	✔
FbgAccelerometerSensor	✔(MXFSのみ)	利用不可	利用不可
FbgGenericSensor	✔(MXFSのみ)	利用不可	利用不可
FbgStrainGageSensor	✔(MXFSのみ)	利用不可	利用不可
FbgTemperatureSensor	✔(MXFSのみ)	利用不可	利用不可
FbgWavelengthSensor	✔(MXFSのみ)	利用不可	利用不可
IEPE	✔	利用不可	✔
LVDT	✔	✔	✔
PiezoPassive	✔	利用不可	✔
Potentiometer	✔	✔	✔
Pt	✔	利用不可	✔
PWM	✔	✔	✔
Resistance	✔	利用不可	✔
SSI	✔	✔	利用不可
ThermoCouple	✔	利用不可	✔
Voltage	✔	✔	✔
VoltageProbe	利用不可	利用不可	利用不可
Analog In DAQ	✔	✔	✔
Analog Out (direct setting)	✔	✔	✔
Analog Out (channel routing)	✔	✔	✔
Digital In DAQ	✔	✔	✔
Digital Out DAQ	✔	✔	✔
Digital Out (direct setting)	✔	✔	✔
CAN DAQ	✔	✔(CODESYS/ calc.チャネル経由)	✔
CAN Raw-DAQ	✔	利用不可	利用不可
Optical DAQ	✔	利用不可	利用不可

✔ Common APIバージョン5.0でサポート
✔ Common APIバージョン6.0/6.1以降に対応

3.1 Common API V6.1の新機能

前回のリリース以降に追加された、最も関連性の高いプロパティと関数をここに示します:

全般:

- ファイバブラッググレーティング(Fiber Bragg Grating: FBG)に基づく光デバイスがサポートされるようになりました(MXFS)
- FBGのコンテキストにさまざまなオブジェクトと関数が追加されました(FbgConnector、FbgChannel、FbgSignal、FbgSpectrum、FbgBandsDetectionResultなど)。
- FBGをサポートするために追加された新しいセンサタイプ(センサデータベースには含まれていません)。
- CAN Rawデータ取得に関する各種オブジェクトを追加(CanRawChannel、CanRawSignalなど)。
- 非同期信号がサポートされるようになりました(現在、非同期信号は CanRawSignalのみです)。
- 任意のデータ型を計測値として(CanRawMessagesなど)を使用できます(これまではdoubleのみをサポート)。
- 同期信号と非同期信号が混在したデータ収集をサポートします。
- MGCは接続中に風袋値を0.0にリセットしなくなりました。MGCデバイスの風袋値を設定/取得するには、SetTareValue およびGetTareValue関数を使用してください。

4 制限事項

バージョン6.1のCommon APIは、すべての機能をサポートしているわけではありません。次の表は、現在サポートされていないものの概要を示しています:

4.1 QuantumX/ SomatXR

以下にAPIで現在サポートされていない、いくつかの機能を示します;

機能	説明
IRIG time service configuration	サポートしていません。
Test signals	サポートしていません。
Virtual Channels	計算を行う演算チャネルなどの仮想チャネルはサポートしていません。
Firmware	4.2.56以降でサポートし、4.36.12までテストしています。

情報:
サポートしていないモジュール: MX590

4.2 PMX

以下にPMX APで現在サポートされていない、いくつかの機能を示します;

機能	説明
Virtual Channels	仮想チャンネル/信号はパラメータ化できません。
Firmware	2.04以降でサポートし、4.40までテストしています。

情報:
サポートしていないモジュール: PX01EC、PX01PN、PX01EP

4.3 MGCplus

MGCデバイスファミリのCP52のみがHBMスキャンメカニズムをサポートしています。CP22またはCP42が付属するMGCデバイスはスキャンメカニズムをサポートしていないため、そのようなデバイスには“manually”で接続する必要があります。MGCデバイスに手動で接続する方法の例については、5.2を参照してください。

次の表に、MGCデバイス ファミリの追加の一般制限を示します:

機能	説明
Trigger	内部トリガマシンはサポートしていません。
Hard disk recording	内蔵ハードディスクへの録画には対応していません。
Serial port	シリアルインタフェースRS-232はサポートしていません。
Virtual Channels	計算を行う演算チャネルなどの仮想チャネルはサポートしていません。
Sensors	SSIは通常、MGCplusではサポートしていません。
Firmware	MGCプラス: CP52: 5.0.15 以降でサポートし、CP52 5.6.0までテストしています。 CP42: 4.84以降でサポートしています。 CP22: 4.44以降でサポートしています。

情報:
サポートしていないアンプモジュール: ML70、ML71S6、ML77 (ML74計測のみ – パラメータ化なし)

5 実行手順

5.1 スキャン

DaqEnvironment.GetInstance()への最初の呼び出しによって実行されるスキャンの初期化と、スキャンの実際の実行の間には、すべてのデバイスを収集するための時間がかかることに注意してください(最大5秒)。

例: スキャンの実行

```
// Init (all device family PLugins are enumerated, scan is initialized)
DaqEnvironment env = DaqEnvironment.GetInstance();

// This is only for demonstration purposes
Thread.Sleep(5000);

// --- Scan for all available devices
List<Device> allDevices = env.Scan();
```

特定のデバイスファミリのみをスキャンしたい場合は、次の行を使用します:

```
// --- Scan for QuantumX and PMX devices only
List<Device> someDevices = env.Scan(new List<string> { "QuantumX", "PMX" });
```

5.2 接続

デバイスに接続するには2つの方法があります。最初にスキャンを実行し、結果リストからデバイスの1つを使用するか、特定のデバイスの接続詳細がすでにわかっている場合は、そのデバイスに手動で接続します。後者は、デバイスファミリがスキャンを提供しない場合の唯一の方法でもあります。
複数のデバイスに同時に接続すると、時間がかかることがあります。したがって、Connect()メソッドはすべてのデバイスを非同期で接続します。

例: スキャンしてデバイスに接続する

```
// --- Execute scan
// .. see above

// Connect to first found device
List<Problem> connectProblems;
bool isOk = env.Connect(allDevices[0], out connectProblems);

// Check for any problems
if(isOk) {
   // No error occured and the device is connected

   if(connectProblems.HasWarning()) {
      // Check possible warnings
   }
} else {
   // Device could not be connected
}
```

例: MGCデバイスと手動で接続

```
// Create connection info
MgcDevice mgc = new MgcDevice("1.2.3.4", 7);
// Connect to device directly without scan
DaqEnvironment env = DaqEnvironment.GetInstance();
List<Problem> connectProblems;
bool isOk = env.Connect(mgc, out connectProblems);

// Check for any problems
if(isOk) {
// No error occured and the device is connected

if(connectProblems.HasWarning()) {
// Check possible warnings
}
} else {
// Device could not be connected
}
```

例: 複数のデバイスに同時に接続し、デバイスが接続されたときに通知を受け取る

```
// Get instance of DaqEnvironment
DaqEnvironment env = DaqEnvironment.GetInstance();

// Thread.Sleep (3000);
// Scan for all available devices
List<Device> foundDevices = env.Scan();

// Register event handler for Connected-Event
MessageBroker.DeviceConnected += MessageBroker_DeviceConnected;

// Connect to all devices asynchronously
List<Problem> connectProblems;
bool isOk = env.Connect(foundDevices, out connectProblems);

void MessageBroker_DeviceConnected( object sender, DeviceEventArgs e ) {
   Console.WriteLine("Device {0} connected", e.UniqueDeviceID);
}
```

5.3 センサの割り当て

センサの割り当ては共通レイヤ上で完全に行うことができ、具体的なデバイス実装にキャストする必要はありません。センサを割り当てるには、接続されたデバイスとセンサの2つだけが必要です。

例: 2点スケーリングを使用した抵抗センサの割り当て

```
// Scan
DaqEnvironment env = DaqEnvironment.GetInstance();
List<Device> foundDevices = env.Scan();

// Use first found device
Device device = foundDevices[0];

// Connect to device --> fills object model
List<Problem> connectProblems;
bool isConnected = env.Connect(device, out connectProblems);

if(isConnected) {
   // Only analog in channels have a sensor
   AnalogInChannel ch = device.Connectors[0].Channels[0] as AnalogInChannel;

   // Was it really an analog in channel
   if(ch != null) {
      // Create sensor object, add to device
      ResistanceSensor resSensor = new ResistanceSensor();
      resSensor.NominalResistance = 500; //Ohm
      resSensor.UniqueName = "My Resistance 500 Ohm";

      TwoPointScaling scaling = new TwoPointScaling();
      scaling.MinEngineeringRange = -500;
      scaling.MaxEngineeringRange = 500;
      scaling.EngineeringUnit = "Ohm";

      scaling.ElectricalP1 = 0;
      scaling.EngineeringP1 = 0;
      scaling.ElectricalP2 = 500;
      scaling.EngineeringP2 = 500;

      // Set scaling
      resSensor.Scaling = scaling;
      // Set sensor
      ch.Sensor = resSensor;

      // --- Assign sensor to device ----------------------
      List<Problem> assignProblems;
      bool isOk = device.AssignSensor(ch, out assignProblems);

      // Check list of problems for errors and warning
      // ......
   }
}
```

5.4 スナップショットの取得

例: デバイスの信号(スナップショット)ごとに1つの値を取得

```
  // Scan
  DaqEnvironment env = DaqEnvironment.GetInstance();
  List<Device> foundDevices = env.Scan();

  // Use first found device
  Device device = foundDevices[0];

  // Connect to device --> fills object model
  List<Problem> connectProblems;
  bool isConnected = env.Connect(device, out connectProblems);

  if(isConnected) {
     // Read snapshot for each signal from device (fill buffers)
     device.ReadSingleMeasurementValueOfAllSignals();

     // Iterate through all signals to show the snapshot data
     List<Signal> allSignals = device.GetAllSignals ();
     foreach(Signal sig in allSignals) {
        // Read data from buffer
        MeasurementValue snapData = sig.GetSingleMeasurementValue();

        // Show the data
        Console.WriteLine("Signal : {0} - {1}", sig.Name, sig.GetUniqueID());
        Console.WriteLine("Timestamp : {0}", snapData.Timestamp);
        Console.WriteLine("Value : {0}", snapData.Value);
        Console.WriteLine("State : {0}", snapData.State);
     }
  }
```

5.5 連続計測

次の例は、連続計測セッションを設定して実行する方法を示しています。この例では、わかりやすいデモンストレーションを目的としており、エラー/問題の処理を省略しています。

例: 2つの信号による計測セッション

```
  private DaqMeasurement _daqSession = null; // DAQ session
  private DaqEnvironment _daqEnv = null; // DAQ environment
  private List<Signal> _sessionSignals = null; // Registered measurement signals
  private System.Threading.Timer _dataFetchTimer = null; // Timer to periodically
fetch data

  void ContinuousMeasurement() {

     // Init
     _daqEnv = DaqEnvironment.GetInstance();
     _daqSession = new DaqMeasurement();
     _sessionSignals = new List<Signal>(); // To store our registered signals
     // Data fetch timer, with callback. We start it later !!
     _dataFetchTimer = new System.Threading.Timer(FetchData, null, Timeout.Infinite,
0);

     // Please note: You normally need some time to gather devices
     // Thread.Sleep (3000);
     // --- Scan for all available devices
     List<Device> scanDevices = _daqEnv.Scan();
```

```
    if(scanDevices.Count > 0) {
        // Use first device, no matter what device type it really is
        Device measDevice = scanDevices[0];

        // Connect to device --> fills object model of device
        List<Problem> problems;
        bool isConnected = _daqEnv.Connect(measDevice, out problems);

        if(isConnected) {
            // Use signals from first 2 connectors
            Signal s1 = measDevice.Connectors[0].Channels[0].Signals[0];
            Signal s2 = measDevice.Connectors[1].Channels[0].Signals[0];

            // ------------------------------------------------------------
            // ... parameterize device --> sensor / filter etc.
            // Omitted for demonstration purposes, only set sample rate!!
            // ------------------------------------------------------------
            s1.SampleRate = 2400; // Hz
            s2.SampleRate = 2400;

            // We don't check problems list here
            measDevice.AssignSignal(s1, out problems);
            measDevice.AssignSignal(s2, out problems);

            // Register first 2 signals for measurement session
            _sessionSignals.Add(s1);
            _sessionSignals.Add(s2);
            _daqSession.AddSignals(measDevice, _sessionSignals);

            // Prepare DAQ session. We only need 1
            // single timestamp per block
            _daqSession.PrepareDaq();

            // ---- Start unsynchronized DAQ session ---------------
            _daqSession.StartDaq(DataAcquisitionMode.Unsynchronized);
            StartDataFetching(); // Start callback timer

            // ----------------------------------------
            // Stop Daq session after 5 seconds !!
            Thread.Sleep(5000);
            StopDataFetching();
            // ----------------------------------------

            // Disconnect device
            _daqEnv.Disconnect(measDevice);
        }
    }
    // Cleanup
    _daqSession.Dispose();
    _daqEnv.Dispose(); // Call this only once at the end of your application
}

/// <summary>
/// Periodically fetch data
/// </summary>
/// <param name="o">Unused. Only needed because of delegate signature</param>
void FetchData( object o ) {
    // Check if our session is running
    if(_daqSession.IsRunning) {

        // Transfer data into signal buffers, Keep in mind we only have 1 timestamp
        _daqSession.FillMeasurementValues();
```

```
        // Process data of each signal, e.g. store to file or show it in chart
        // For demonstration purposes we only show first value on console
        foreach(Signal s in _sessionSignals) {
            MeasurementValues measVals = s.ContinuousMeasurementValues;

            // Check for overrun
            if(measVals.BufferOverrunOccurred) {
                Console.WriteLine("Buffer overrun for signal {0}", s.Name);
            }

            // How many new values do we have ?
            int newValuesCount = measVals.UpdatedValueCount;
            Console.WriteLine("{0} updated values {1}", s.Name, newValuesCount);

            if(newValuesCount > 0) {
                // Show first data value
                // In real world application we would take
                // measVals.UpdatedValueCount from buffer
                // --> measVals.Values[measVals.UpdatedValueCount-1]
                double timestamp = measVals.Timestamps[0];
                double value = measVals.Values[0];

                Console.WriteLine("value: {0} : {1}", timestamp, value);
            } else {
                // We don't have values
                Console.WriteLine("No values for signal: {0}", s.Name);
            }
        }
    }
}
void StartDataFetching() {
    // Data fetching every 100ms
    _dataFetchTimer.Change(0, 100);
}

void StopDataFetching() {
    // Stop data fetching
    _dataFetchTimer.Change(Timeout.Infinite, Timeout.Infinite);
    _dataFetchTimer.Dispose();
    // Stop our DAQ session
    _daqSession.StopDaq();
}
```

5.6 特別なデバイス関数

すべてのデバイスファミリは特別な関数をサポートしています。このような関数を使用するには、具体的なデバイス実装にキャストする必要があります。

例: QuantumXデバイスのLEDの点滅

```
// --- Scan for QuantumX only
DaqEnvironment env = DaqEnvironment.GetInstance();
List<Device> qxDevices = env.Scan(new List<string> { "QuantumX" });

// Use first QuantumX device
QuantumXDevice qxDevice = (QuantumXDevice) qxDevices[0];
// Connect to device --> fills object model
List<Problem> connectProblems;
bool isConnected = env.Connect(qxDevice, out connectProblems);

if(isConnected) {
   // Flash LED
   qxDevice.AdditionalFeatures.EnableLedBlinking();

   // Let it blink for 2 seconds
   Thread.Sleep(2000);

   // Stop blinking
   qxDevice.AdditionalFeatures.DisableLedBlinking();

   // Disconnect from device
   env.Disconnect(qxDevice);
}
```

例: PMXデバイスへのコマンド トンネル

```
// --- Scan for PMX only
DaqEnvironment env = DaqEnvironment.GetInstance();
List<Device> pmxDevices = env.Scan(new List<string> { "PMX" });

// Use first PMX device
PmxDevice pmxDevice = (PmxDevice) pmxDevices[0];

// Connect to device
List<Problem> connectProblems;
bool isConnected = env.Connect(pmxDevice, out connectProblems);
if(isConnected) {

   // Use direct device communication tunnel. IDN? = ask device for its identity
   string response = pmxDevice.AdditionalFeatures.SendCommand("IDN?");

   // Show result
   Console.WriteLine(response);

   // Disconnect from device
   env.Disconnect(pmxDevice);
}
```

5.7 センサデータベースへのアクセス

センサとスケーリングタイプの定義に加えて、名前空間Hbm.Api.SensorDBにはSensorDBManagerオブジェクトが含まれています。このオブジェクトは、センサの取得、挿入、更新、削除と、センサの階層カテゴリ(グループ)をさまざまな言語で管理できるようにするISensorDBインタフェースを実装します。

例: 特定のセンサを取得するさまざまな方法と、HBMセンサデータベースからカテゴリとセンサー式の階層を取得する方法

```
// Open HBM sensor database that is located
// in the same directory as the executable
string sdbFilename=    System.IO.Path.Combine(
                       System.IO.Directory.GetCurrentDirectory(),
                       "HBMSensorDatabase.vdb5");
// Create a sensorDBManager to get access to the
// sensor database (readonly)
ISensorDB sdbManager = new SensorDBManager(
                       "en",
"Data Source=" + sdbFilename+
";Open Mode=NonExclusiveReadOnly;",
"System.Data.VistaDB5");
// Get the complete hierarchy of the sensor database including all
// categories/subcategories and their sensors
// just add a breakpoint here and explore the nodes:
List<HierarchicalCategory> hierarchicalCategoryNodes =
   sdbManager.GetHierarchicalCategories();
// within each hierarchicalCategoryNode you will find a list of
// sensor objects that belongs to this category

// there are various ways to get a sensor...
// e.g. by getting a list of all sensors of the sensor database
// (do that only once...) and search for a specific sensor on
// your own... (here: get a list of all sensors and search
// for the first voltage sensor)
Sensor sensor;
List<Sensor> listOfAllSensors = sdbManager.GetSensors();
sensor = listOfAllSensors.Where(
   p => p.SensorType== SensorDB.Enums.SensorType.Voltage).First();

// or e.g. let the sensorDBManager search for a sensor
// with a certain ID
sensor = sdbManager.GetSensor(112);

// or e.g. let the sensorDBManager search for a sensor
// with a certain unique name
sensor = sdbManager.GetSensor("HBM_T22_50Nm");
```

例: ユーザセンサデータベースを処理するさまざまな方法 (接続、カテゴリとセンサの階層の取得、センサーの挿入、更新、削除)

```
// Open user sensor database that is located
// in the same directory as the executable
string sdbFilename =    System.IO.Path.Combine(
                        System.IO.Directory.GetCurrentDirectory(),
                        "UserSensorDatabase.vdb5");

// Create a sensorDBManager to get access to the
// sensor database (read/write access)
ISensorDB sdbManager = new SensorDBManager(
                        "en",
                        "Data Source=" +
                        sdbFilename +
                        ";Open Mode=ExclusiveReadWrite;",
                        "System.Data.VistaDB5");

// Get the complete hierarchy of the sensor database
// including all categories/subcategories and their sensors
// just add a breakpoint here and explore the nodes:
List<HierarchicalCategory> nodes =
   sdbManager.GetHierarchicalCategories();

// Insert a sensor into a certain category of the database

// Therefore we have to do 2 things here:
// 1. Create a sensor object (we use the SensorFactory here
// to get a certain sensortype with default scaling...)
VoltageSensor myVoltageSensor = SensorFactory.CreateSensor(
   SensorDB.Enums.SensorType.Voltage) as VoltageSensor;
// set the name of the sensor and further properties if you like
myVoltageSensor.Description = "my voltage sensor";

// 2. To insert a sensor into a sensor database, we have to
// choose an existing category (group) of the sensor database into
// which the sensor should be inserted.
// Here we insert the sensor into the "Imported" category of the
// database (this category exists in each user sensor database).
Category categoryIntoWhichToInsertTheSensor = sdbManager.GetCategories().
   Where(c => c.Description == "Imported").First();
// Set the Category property of the sensor we want to insert
myVoltageSensor.Category = categoryIntoWhichToInsertTheSensor;
// Insert the sensor into the user database
sdbManager.InsertSensor(myVoltageSensor);
// ATTENTION! After successfully inserting the sensor, some
// properties of the sensor have been changed!!!
// Take a look at following sensor properties:
// SensorID, UniqueName, IsUserDefined, LastModifiedDate...

// To update the sensor, we just change its description here
myVoltageSensor.Description = "my updated voltage sensor";
// and update the (now) already existing sensor in the
// sensor database... (Description AND LastModifiedDate will be changed!)
sdbManager.UpdateSensor(myVoltageSensor);

// To delete the sensor we just call
sdbManager.DeleteSensor(myVoltageSensor.SensorID);
```

5.8 ロギング

Hbm.Api.Logging名前空間には、さまざまなログフレームワークを使用してメッセージをログに記録できるようにする機能が含まれています(現在はNLogとLoupeをサポートしています)。

例: アプリケーション内でロギングを使用する方法

```
  // Start logging with a certain LoggingFramework (Loupe or NLog)
  // and a combination of "ORed" LogLevels:
  LogManager.Start (LoggingFramework.NLog,
     LogLevel.Error | LogLevel.Warn | LogLevel.Info);

  // Create a logger object that has to be used to log messages
  ILogger logger = LogManager.CreateLogger("ApiDemo");

  // We want to group our log entries into different parts.
  // Logging uses LogContext objects to define these groups.
  // Notice: The Hbm.CommonAPI will also log messages
  // additional to the messages that will be logged here!

  // E.g.: One group to collect entries concerning measurement tasks:
  LogContext logContextMeasuring = new LogContext("ApiDemo.Measurement");

  // E.g.: One group to collect entries concerning problems:
  LogContext logContextProblems = new LogContext("ApiDemo.Problems");

  // The resulting hierarchy into which the entries can be logged (using Loupe) is
now:
  // Categories
     // - ApiDemo
        // - Measurement
           // - Error: This is an error type log entry!
        // - Problems
           // - Info: This is an info type log entry!
           // ...
  // using Nlog, these hierarchy infos are included in the line
  // that will be appended to the logfile
  // e.g.: 2015-09-10 09:37:25.1234 [10] ERROR |
  // ApiDemo.Measurement| ApiDemo: Error: This is an error type log entry!

  // Generate some log entries (under Measurement)
  // this entry will be logged because we started logging with "LogLevel.Error"
  logger.Error(logContextMeasuring, "Error: This is an error type log entry!");

  // Following entry will *** N O T *** be logged because we
  // started logging without "LogLevel.Debug"
  logger.Debug(logContextMeasuring, "Debug: This is a debug type log entry!");

  // Generate log entry (under Problems)
  logger.Info(logContextProblems, "Info: This is an info type log entry!");

  // End logging.
  LogManager.Stop();

  // You can find the directory and the created logfile under
  Debug.Print("Directory of the logfile: "+LogManager.LogFolder);
  Debug.Print("Fullt path to the logfile: "+ LogManager.CurrentLogFile);
```

6 トラブルシューティングのヒント

一般的なエラーに対する次の解決策は、問題の解決に役立つ可能性があります。

一般的に: 問題のトラブルシューティングを行うには、API関数を使用する前にログ記録をアクティブ化し(たとえば、"LogManager.Start(LoggingFramework.NLog, LogLevel.All)"を追加する)、アプリケーションの実行中に生成されるログファイルを確認することをお勧めします 。実稼働コードでログレベルを無効にするか取り下げることを忘れないでください。

問題: スキャンしてもデバイスが見つかりません。
解決策: ファイアウォールの設定を確認してください。受信UDPポート31416および31417をアプリケーション用に開く必要があります。

デバイスとPCの間にルータがないことを確認します。ルータは、通常はPCが受信するマルチキャストメッセージ(スキャン可能なデバイスから送信された)をドロップします。また、一部の管理対象スイッチではこれらのメッセージがドロップされます。

デバイスがPCに直接接続されている場合は、デバイスがPCに物理的に接続された後にスキャンを呼び出す必要があります。そうしないと、スキャンはデバイスが接続されているネットワークアダプタを無視します。

また、DeviceDriver.pluginsファイルが実行可能ファイルと同じディレクトリにあることを再確認してください。

問題: 計測を開始しません("同期"計測を開始する時間が足りません)。
解決策: 特に非常に低い計測レート(0.1Hzなど)を使用する場合、デフォルトの同期タイムアウト(5000 ミリ秒)が短すぎるため、同期計測を開始できない可能性があります。その場合はタイムアウトを長くしてください。このエラーは、デバイスが相互に同期していない場合(同じNTPサーバを使用していない場合など)にも発生する可能性があります。

問題: 同期計測の実行中、突然一部の信号が新しい計測値を取得できなくなります。
解決策: 信号が属するデバイスへの接続が失われた可能性があります。Common APIのバージョン3.0以降、信号の計測レート (Signal.ExcludeFromDaqTimeout)に応じて、一定時間新しい計測値を受信しなかった場合、フラグSignal.IsExcludedFromDaqがtrueに設定されます。ネットワーク接続を確認してください。

問題: ContinuousMeasurementValues には、新しいタイムスタンプと新しいステータスが1つだけ含まれるはずですが、複数の新しい計測値が含まれています。
解決策: DaqMeasurement.PrepareDaq関数の別の多重定義を使用して、各計測値のタイムスタンプとステータスを取得してください。パフォーマンス上の理由により、デフォルトは1つのタイムスタンプ/ステータスのみです。

問題: ContinuousMeasurementValuesに、非現実的な計測値または過去のタイムスタンプが含まれています。
解決策: "DaqMeasurement.FillMeasurementValues"によって測定値を更新した後、ContinuousMeasurementValues.UpdatedValueCountを使用して新しい計測値の数を決定する必要があります。ContinuousMeasurementValues.Lengthは使用しないでください。

問題: センサが奇妙な計測値を出力したり、スケーリングが計測値に適合しません。
解決策: デバイスのオブジェクト表現内で何かを変更した後、適切な割り当て関数(AssignSignal、AssignSensor、AssignConnectorなど)を呼び出したことをアサートします。

問題: 同じファミリのデバイス複合体のハードウェア同期測定を開始したいと考えています。

解決策: QuantumXデバイスの場合は、すべてのデバイスをFireWire経由で接続し、"TimeStampSynchronized"計測を開始する必要があります。

MGCplusデバイスの場合は、同期ケーブルを各デバイスに接続し、1つのMGCplusをマスタとして設定し、他のすべてをスレーブとして設定し、"HardwareSynchronized"計測を開始する必要があります。

PMXデバイスの場合、ハードウェア同期計測はまだ可能ではありません。各デバイスに同じNTPタイムサーバを使用し、代わりにソフトウェア同期計測を開始してください。

問題: プログラムが起動しません (間違ったアセンブリ エラー)。

解決策: 間違ったバージョンのHBM_Scan.dll、HBM_QX_Framework.dll、またはHBM_Streaming.dllを使用している可能性があります。バージョンがマシン/プロジェクト設定のx86またはx64に適合していることを確認します。

www.hbm.com

HBM Test and Measurement
Tel. +49 6151 803-0
Fax +49 6151 803-9100
info@hbm.com

アドバン オートメーション株式会社
Tel. 03-5282-7047
Fax. 03-5282-0808
info-advan@adv-auto.co.jp

measure and predict with confidence

HBM