Getting started with PalmSens SDK for WPF

Based on PalmSens SDK v5.9

Last revision: October 7, 2021

www.palmsens.com

PalmSens

Contents

1 Contents of the PalmSens SDK 3
1.1 Example programs 4
1.2 Compatible devices and firmware 6
2 Using the SDK in your Visual Studio.NET project 7
2.1 Add SDK References to the project 7
2.2 Add PSCommSimpleWPF or PS user control to your user interface (simplified wrapper only) 7
3 PalmSens.Core.dll 9
4 Working with files 11
4.1 Loading a method file (.psmethod) 11
4.2 Setting up a method 11
4.3 Saving a method 12
4.4 Loading and saving data 12
4.5 Loading data from PalmSens 4 internal storage ..... **Error! Bookmark not defined.**
5 Connecting and Measuring 15
5.1 Connecting to a device 15
5.2 Receive idle status readings 16
5.3 Manually controlling the device 18
5.4 Measuring 18
5.5 Disconnecting and disposing the device 25
5.6 Possible causes of communication issues 25
6 Data analysis and manipulation using the simplified wrapper 27
6.1 Obtaining the measured values 27
6.2 Smoothing/Filtering 27
6.3 Baseline Subtraction 27
6.4 Basic operations 28
6.5 Peak and level detection 28
6.6 Equivalent circuit fitting 29
7 Appendix A: Parameters for each technique 30
7.1 Common properties 30
7.2 Pretreatment settings 31
7.3 Linear Sweep Voltammetry (LSV) [0] 31
7.4 Differential Pulse Voltammetry (DPV) [1] 31
7.5 Square Wave Voltammetry (SWV) [2] 31
7.6 Normal Pulse Voltammetry (NPV) [3] 32
7.7 AC Voltammetry (ACV) [4] 32
7.8 Cyclic Voltammetry (CV) [5] 32
7.8.1 Fast Cyclic Voltammetry Scans 32
7.9 Chronopotentiometric Stripping (SCP) [6] 33
7.10 Chronoamperometry (CA) [7] 33
7.11 Pulsed Amperometric Detection (PAD) [8] 33

7.12 Fast Amperometry (FAM) [9] 33
7.13 Chronopotentiometry (CP) [10] 34
7.13.1 Open Circuit Potentiometry (OCP) 34
7.14 Multiple Pulse Amperometry (MPAD) [11] 34
7.15 Electrochemical Impedance Spectroscopy (EIS) 34
7.15.1 Time Scan 35
7.15.2 Potential Scan 36
7.16 Recording extra values (BiPot, Aux, CE Potential…) 36
7.17 Multiplexer 37
7.17.1 Multiplexer settings 38
7.18 Versus OCP 38
7.19 Properties for EmStat Pico 39

1 Contents of the PalmSens SDK

The PalmSens SDK contains the following libraries and projects:

PalmSens.Core.dll & PalmSens.Core.Windows.dll:
These libraries contain the namespaces with all the necessary files for using PalmSens/EmStat devices in your software.

- PalmSens — All necessary classes and functions for performing measurements and doing analysis with PalmSens or EmStat.
- PalmSens.Comm — For Serial, USB or TCP communication with PalmSens or EmStat
- PalmSens.DataFiles — For saving and loading method and data files
- PalmSens.Devices — For handling communications and device capabilities
- PalmSens.Techniques — Contains all measurement techniques for PalmSens and EmStat
- PalmSens.Units — Contains a collection of units used by these libraries

PalmSens.Core.Windows.BLE.dll
This is an optional library to add Bluetooth Low Energy support to your project (Windows 10 only).

PalmSens.Core.Simplified.csproj & PalmSens.Core.Simplified.WPF.csproj:
These projects contain an open source wrapper for the PalmSens.Core.dll & PalmSens.Core.Windows.dll. This wrapper gives you quick and easy access to all the basic functions of the PalmSens/EmStat devices and automatically handles most potential threading issues for you:

- Connecting
- Manual control of the cell
- Running measurements
- Accessing and processing measured data
- Analyzing and manipulating data

SDKPlot.csproj & SDKPlot.WPF.csproj
These projects contain a simple open source plot control that you can use to plot your measurements in real-time.

OxyPlot.dll & OxyPlot.Wpf.dll:
These libraries required when using the SDK's plot based on the open source OxyPlot library, http://www.oxyplot.org/.

1.1 Example programs

The following examples are included. All examples use the simplified wrapper for the PalmSens.Core libraries, except the Console Example.

Example – Console Example (C#):
Shows how to discover devices, establish a connection and run a measurement using only the PalmSens.Core.dll & PalmSens.Core.Windows.dll

Example – Basic Example (C#):
Shows how to make a connection and run a measurement, with a minimum amount of code.

Example – Plot Example (C#):
Shows how to make a connection, run a measurement and plot the results.

Example – Data Example (C#):
Shows how to load and measurements and methods from and to *.pssession and *.psmethod files, and how to smooth data, manipulate data and detect peaks.

Example – Multiplexer Example (C#):
Shows how to make a connection and run a measurement on different multiplexer channels.

Example – Auxilliary/BiPot Example (C#):
Shows how to measure additional data from the auxiliary port or a bipot.

Example – Peak Browser Example (C#):
Shows how to load files and access the parameters of peaks found via the automated peak detection.

Example – Electrochemical Impedance Spectroscopy Fit Example (C#):
Shows how perform an equivalent circuit analysis on an EIS frequency scan measurement.

Example – Peak Detection Example (C#):
Shows how to perform an advanced peak detection for Linear Sweep Voltammetry and Cyclic Voltammetry measurements.

Example – Internal Storage (C#):
Shows how to browse and load contents from the internal storage of a EmStat Pico Development Board, Sensit BT, EmStat4 and PalmSens4.

The following examples utilize asynchronous programming to prevent the PalmSens SDK libraries from blocking the user interface. Please note that when using the async functionalities of the core not to mix in any synchronous functions that communicate with the PalmSens/EmStat instrument.

Example – Basic Example Async (C#):
Shows how to make a connection and run a measurement, with a minimum amount of code.

Example – Plot Example Async (C#):
Shows how to make a connection, run a measurement and plot the results.

Example – Multi Channel Example (C#):
Shows how to make a connection to multiple channels, run different types of measurements and plot the results.

Example – Electrochemical Impedance Spectroscopy Fit Example Async (C#):
Shows how perform an equivalent circuit analysis on an EIS frequency scan measurement.

Example – Multi Electrochemical Impedance Spectroscopy Fit Example Async (C#):
Shows how perform multiple equivalent circuit analysis on an EIS frequency scan measurement simultaneously.

Example – Peak Detection Example Async (C#):

Shows how to perform an advanced peak detection for Linear Sweep Voltammetry and Cyclic Voltammetry measurements.

Example – Peak Detection Example Async (C#):

Shows how to perform an advanced peak detection for Linear Sweep Voltammetry and Cyclic Voltammetry measurements.

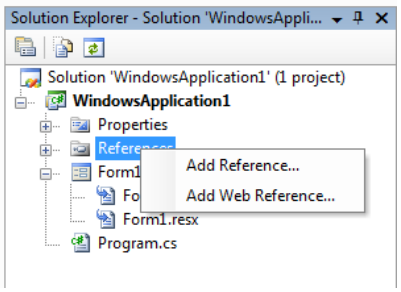
1.2 Compatible devices and firmware

	Minimum required firmware version
EmStat	3.7
EmStat2	7.7
EmStat3	7.7
EmStat3+	7.7
EmStat4	1.000
EmStat Go	7.7
EmStat Pico	1.301
Sensit Smart	1.301
Sensit BT	1.301
MultiEmStat	7.7
PalmSens3	2.8
PalmSens4	1.7
MultiPalmSens4	1.7

2 Using the SDK in your Visual Studio.NET project

2.1 Add SDK References to the project

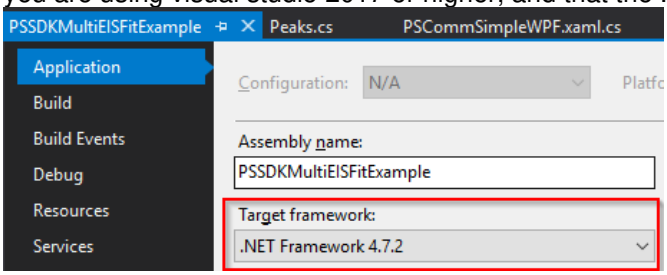
Right-click on the References map in your project and click Add Reference:



Reference **external libraries (*.dll files)** by selecting them via the 'Browse' button and reference **project files (*.csproj)** by adding them to your solution and referencing the project. (**Important!** When using the PalmSens SDK without the simplified wrapper the **CoreDependencies** must be initiated before you run a measurement, to do this call the **PalmSens.Windows.CoreDependencies.Init()** method one time beforehand)

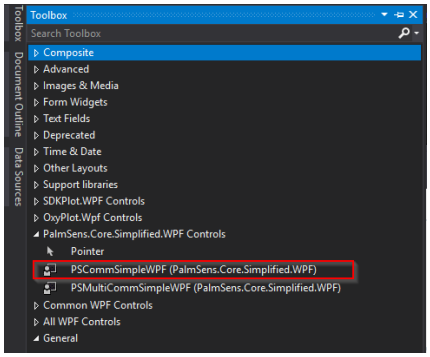
The PalmSens.Core.dll and PalmSens.Core.Windows.dll libraries should always be referenced. If you wish to use the simplified wrapper to control your devices the **PalmSens.Core.Simplified.csproj** and **PalmSens.Core.Simplified.WPF.csproj** should be referenced. To use the plot control the **SDKPlot.csproj**, **SDKPlot.WPF.csproj**, **OxyPlot.dll,** and **OxyPlot.WPF.dll** should be referenced.

If your project does not build please make sure you are targeting the .NET Framework 4.7.2 or higher, you are using visual studio 2017 or higher, and that the .NET standard 2.0 is available.



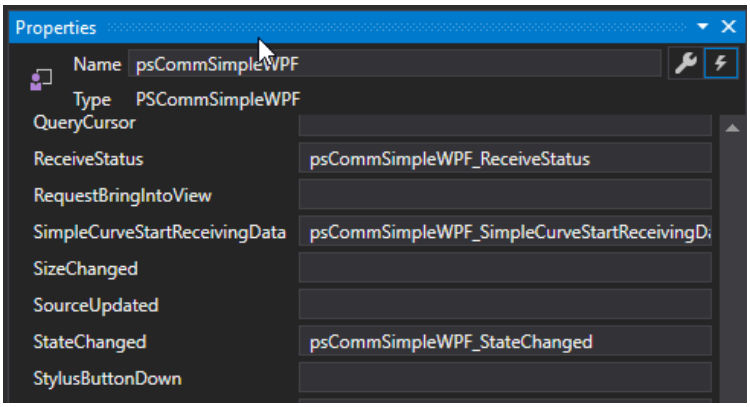
2.2 Add PSCommSimpleWPF or PS user control to your user interface (simplified wrapper only)

First import the **PalmSens.Core.Simplified.csproj** and **PalmSens.Core.Simplified.WPF.csproj** into your solution and build the **PalmSens.Core.Simplified.WPF** project. Then reference these projects in your project. Next, go to the designer of your **Main Form** and look for the **PSCommSimpleWPF** user control in your toolbox. Drag and drop this component on top of your **Window**. If you would like to develop for a multi channel instrument use the **PSMultiCommSimpleWPF** user control instead.



Finally, select the PSCommSimpleWPF/ PSMultiCommSimpleWPF user control in your XAML, and go to the events section of the properties to add the events you require.

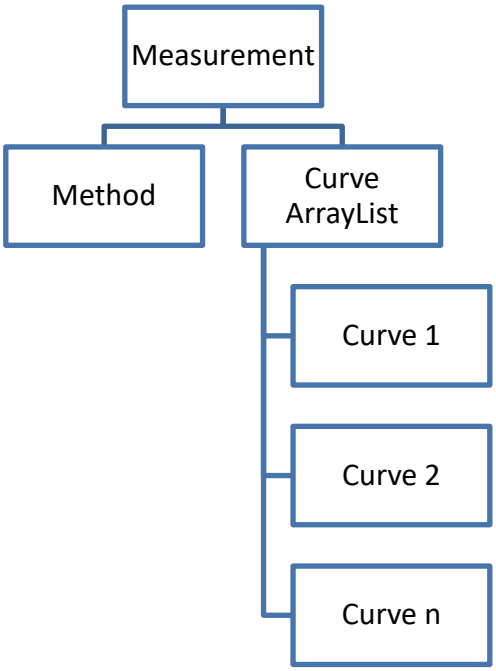
```
WPF:PSCommSimpleWPF (psCommSimpleWPF)                    WPF:PSCommSimpleWPF (psCommSimpleWPF)
46                </Grid>
47            </GroupBox>
48            <WPF:PSCommSimpleWPF x:Name="psCommSimpleWPF" HorizontalAlignment="Left" Height="100" VerticalAlignment="Top" Width="100"
49                                 ReceiveStatus="psCommSimpleWPF_ReceiveStatus"
50                                 StateChanged="psCommSimpleWPF_StateChanged"
51                                 MeasurementStarted="psCommSimpleWPF_MeasurementStarted"
52                                 MeasurementEnded="psCommSimpleWPF_MeasurementEnded"
53                                 SimpleCurveStartReceivingData="psCommSimpleWPF_SimpleCurveStartReceivingData"/>
54        </Grid>
55    </Window>
56
```



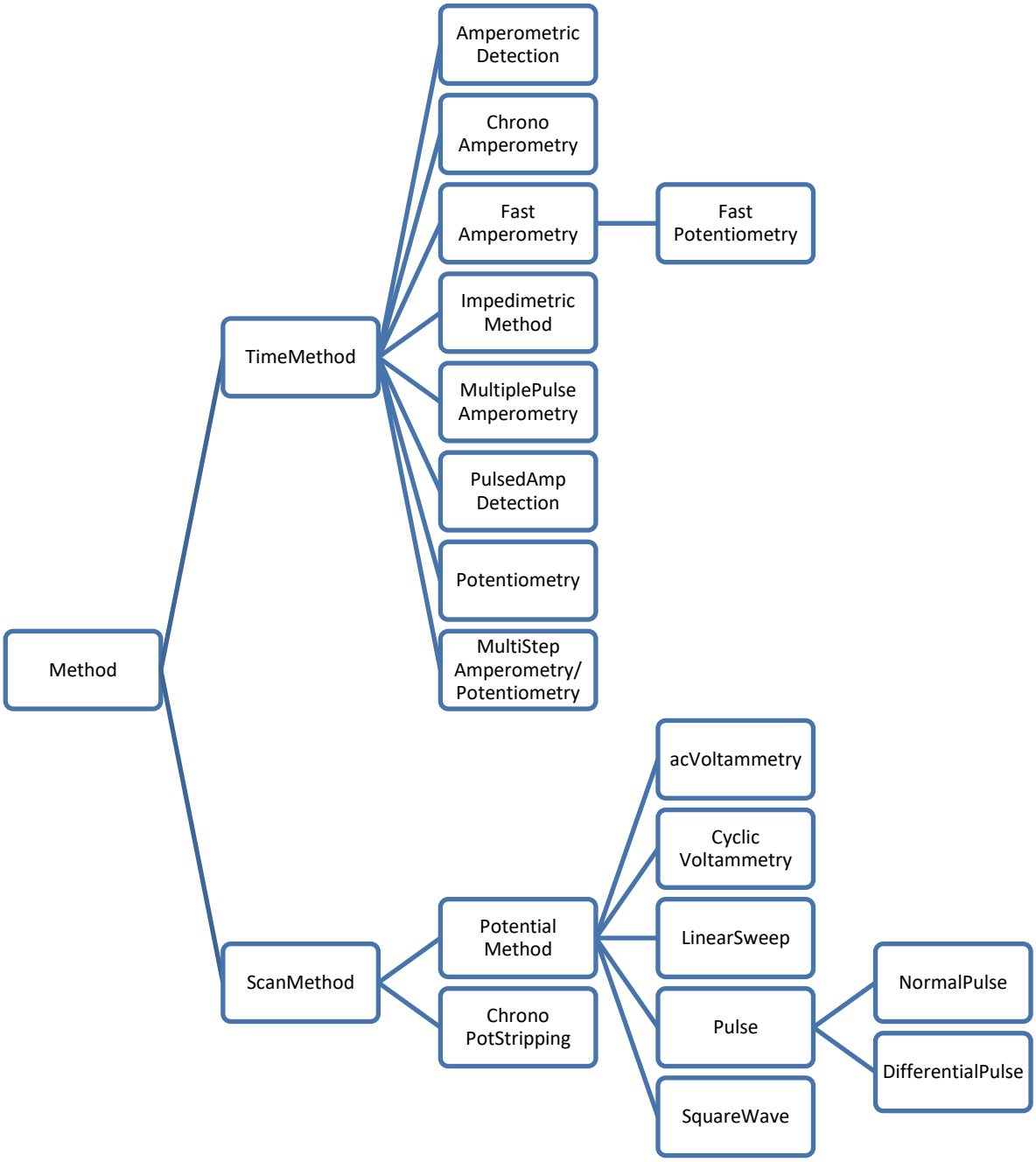
3 PalmSens.Core.dll

The basis for handling measurements is the **PalmSens.Measurement** class, or the **PalmSens.Core.Simplified.Data.SimpleMeasurement** class when using the simplified wrapper.

The measurement class contains all classes, functions, and parameters necessary for performing a measurement with a PalmSens or EmStat instrument. It has one method and can contain multiple curves. Curves are a representation of the data in the measurement used for plotting and analysis.



The following diagram shows the inheritance structure of the Method classes:



4 Working with files

As of version 5 of the PalmSens SDK and PSTrace measurements and their corresponding methods are stored in ***.pssession** files. Methods can be stored separately in ***.psmethod** files.

The PalmSens SDK is backward compatible with following filetypes:

	vs potential (scan method)	**Measurement vs time**
Method file	.pms (before 2012)	.pmt (before 2012)
Method file	.psmethod (default)	.psmethod (default)
Data (single curve) file	.pss	.pst
Analysis curves file	.psd	
Multiplexer curves file		.mux

4.1 Loading a method file (.psmethod)

In order to use the following examples, make sure the **PalmSens.Core.dll** and **PalmSens.Core.Windows.dll** are added as references in your project. For the examples using the Simplified Core wrapper the **PalmSens.Core.Simplified.csproj** and **PalmSens.Core.Simplified.WPF.csproj** must also be referenced.

Simplified PalmSens.Core:

```
using PalmSens;
using PalmSens.Core.Simplified.WPF;
```

Add these namespaces at the top of the document.

```
Method method = SimpleLoadSaveFunctions.LoadMethod(filepath);
```

This method loads a ***.psmethod** file from the specified file path (i.e. C:\\YourMethod.psmethod).

PalmSens.Core:

```
using PalmSens;
using PalmSens.Windows;
```

Add these namespaces at the top of the document.

```
Method method = LoadSaveHelperFunctions.LoadMethod(filepath);
```

This method loads a ***.psmethod** file from the specified file path (i.e. C:\\YourMethod.psmethod).

4.2 Setting up a method

The next example defines a Linear Sweep Voltammetry **method** with the same parameters shown in the table on the previous page.

Example:

```
using PalmSens;
using PalmSens.Techniques;
```

Add these namespaces at the top of the document.

```
LinearSweep lsv = new LinearSweep();
```

Instantiate a new Linear Sweep Voltammetry method.

```
lsv.BeginPotential = -1f;
lsv.EndPotential = 1f;
lsv.StepPotential = 0.01f;
lsv.Scanrate = 1f;
```

Define the method's parameters.

```
lsv.Ranging.StartCurrentRange = new CurrentRange(CurrentRanges.cr1uA);
lsv.Ranging.MaximumCurrentRange = new CurrentRange(CurrentRanges.cr10uA);
lsv.Ranging.MaximumCurrentRange = new CurrentRange(CurrentRanges.cr100pA);
```

Define the current range settings. The CurrentRange constructor uses the enum PalmSens.CurrentRanges to specify its range. Older versions of the SDK used an integer to specify its range:

```
-1 = 100 pA
 0 = 1 nA
 1 = 10 nA
 2 = 100 nA
 3 = 1 uA
 4 = 10 uA
 5 = 100 uA
 6 = 1 mA
 7 = 10 mA
 8 = 100 mA
```

4.3 Saving a method

To save the Linear Sweep Voltammetry **method** with the parameters as defined in the previous example, the following examples can be used:

Simplified PalmSens.Core:

```
using PalmSens;
using PalmSens.Core.Simplified.WPF;
```

Add these namespaces at the top of the document.

```
SimpleLoadSaveFunctions.SaveMethod(lsv, filepath);
```

This saves the previously defined Linear Sweep Voltammetry **method** (lsv) to a ***.psmethod** file specified in the file path (i.e. C:\\YourMethod.psmethod).

PalmSens.Core:

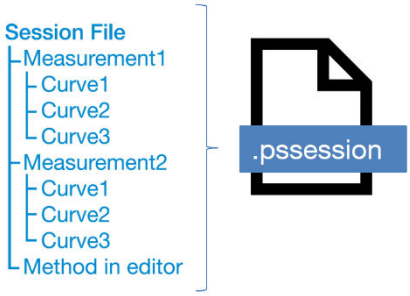
```
using PalmSens;
using PalmSens.Windows;
```

Add these namespaces at the top of the document.

```
LoadSaveHelperFunctions.SaveMethod(lsv, filepath);
```

This saves the previously defined Linear Sweep Voltammetry **method** (lsv) to a ***.psmethod** file specified in the file path (i.e. C:\\YourMethod.psmethod).

4.4 Loading and saving data



Data from measurements can be loaded from and stored to ***.pssession** files. This contains a session with one or more measurements containing its respective method and curves.

Loading measurements from ***.pssession** file a is done by the following code:

Simplified PalmSens.Core:
The simplified wrapper for the **PalmSens.Core** libraries uses the **SimpleMeasurement** and **SimpleCurve** classes from the **PalmSens.Core.Simplified.Data** namespace instead of the **PalmSens.Measurement**, **PalmSens.Plottables.Curve** and **PalmSens.Plottables.EISData** classes. The **SimpleMeasurement** and **SimpleCurve** classes make it easier to perform basic functions such as:

- Creating a curve with different units from a measurement (for example a curve with charge over time).
- Finding peaks in a curve.
- Determining the moving average baseline of a curve.
- Performing basic operations on a curve (Addition, Subtraction, Multiplication, Log10, Differentiation, Integration).

```
using PalmSens;
using PalmSens.Core.Simplified.Data;
using PalmSens.Core.Simplified.WPF;
```

Add these namespaces at the top of the document.

```
SimpleLoadSaveFunctions.SaveMeasurement(lsv, filepath);
```

This saves a **SimpleMeasurement** to a ***.pssession** file specified in the file path (i.e. C:\\YourData.pssession).

PalmSens.Core:

```
using PalmSens;
using PalmSens.Data;
using PalmSens.Windows;
```

Add these namespaces at the top of the document.

```
SessionManager session = new SessionManager();
session.AddMeasurement(measurement);
LoadSaveHelperFunctions.SaveSessionFile(filepath, session);
```

This saves a measurement (**PalmSens.Measurement** class) to a ***.pssession** file specified in the filepath (i.e. C:\\YourData.pssession).

4.5 Loading data from internal storage

When connected to a PalmSens4, EmStat4, Sensit BT or EmStat Pico Development Board it is possible to list and retrieve the measurements stored on its internal storage. The documentation on Connecting explains how to connect to a device and create an instance of a CommManager.

Simplified PalmSens.Core:

```
using PalmSens;
using PalmSens.Data;
using PalmSens.Core.Simplified.InternalStorage;

IInternalStorageFolder folder = psCommSimpleWPF.GetInternalStorageBrowser().GetRoot();
```

This line will return the root folder of the device's internal storage. The IInternalStorageFolder interface allows you to list subfolders and any files (measurements) located in that folder.

```
IReadOnlyList<IInternalStorageFolder> subFolders = folder.GetSubFolders();
IReadOnlyList<IInternalStorageFile> files = folder.GetFiles();
```

To load a measurement from a IInternalStorageFile use the GetMeasurement method.

```
SimpleMeasurement measurement = files[0].GetMeasurement();
```

PalmSens.Core:

```
using PalmSens;
using PalmSens.Data;

List<DeviceFile> DeviceFiles = comm.ClientConnection.GetDeviceFiles(“”); //Get the
contents from the root directory
```

The code above lists all the files / folder in the root (“”) of the internal storage. The DeviceFile class contains information on the Type (File/Folder), Name, Dir (Path) and Size. The EmStat4, Sensit BT and EmStat Pico Development Board will list the contents of all subfolders. The PalmSens4 can list the contents of a certain folder pass on the following argument to the GetDeviceFiles method.

```
List<DeviceFile> DeviceFiles = comm.ClientConnection.GetDeviceFiles(file.Dir + "\\" +
file.Name);
```

Where the file object refers to a DeviceFile of the type folder.

To get the contents of a DeviceFile of the type measurement use the GetDeviceFile method.

```
DeviceFile rawData = comm.ClientConnection.GetDeviceFile(file.Dir + "/" + file.Name);
```

This returns a DeviceFile which has a the unparsed measurement stored in its Content property. This can be parsed by creating a new instance of the Measurement Class and parsing the data, for more info please refer to the Internal Storage Example.

5 Connecting and Measuring

The following chapter details how to connect to a device, read data from the device, manually controlling the potential, run measurements on the device and finally how to properly close a connection to a device.

5.1 Connecting to a device

The following example shows how to get a list of all available devices and available serial com ports, and how to connect to one of the discovered devices that.

Simplified PalmSens.Core:

```
using PalmSens.Devices;
```

Add this namespace at the top of the document.

```
Device[] devices = psCommSimpleWPF.ConnectedDevices;
psCommSimpleWPF.Connect(devices[0]);
```

The first line returns an array of all the connected devices, and the second connects to the first device in the array of connected devices. When Bluetooth devices should also be discovered set **psCommSimpleWPF.EnableBluetooth = true** first.

PalmSens.Core:

```
using PalmSens.Comm;
using PalmSens.Devices;
using PalmSens.Windows.Devices;
```

Add these namespaces at the top of the document.

```
//List of discover functions
List<DeviceList.DiscoverDevicesFunc> discFuncs = new
List<DeviceList.DiscoverDevicesFunc>();
```

Create an empty list of device discovery functions.

```
discFuncs.Add(USBCDCDevice.DiscoverDevices); //Default for PS4
discFuncs.Add(FTDIDevice.DiscoverDevices); //Default for Emstat + PS3
discFuncs.Add(SerialPortDevice.DiscoverDevices); //Devices connected via serial port
discFuncs.Add(BluetoothDevice.DiscoverDevices); //Bluetooth devices (PS4, PS3, Emstat
Blue)
```

Add the discovery functions for the types of devices you would like to discover.

```
string errors;
Device[] devices = new DeviceList(discFuncs).GetAvailableDevices(out errors);
```

The GetAvailableDevices() method has an additional optional parameter which must be set to true when Bluetooth devices should be included in the search.

```
try
{
       device.Open(); //Open the device to allow a connection
       comm = new CommManager(devices[0]); //Connect to the device
}
catch (Exception ex)
{
       device.Close();
}
```

To prevent your program from crashing it is recommended to use a try catch sequence when connecting to a device, this way a device will be closed again when an exception occurs. This code will connect to the first device in the array of discovered devices.

Async functionality

When using the async commands without the simplified core wrappers you will need to initiate the synchronization context remover, it is recommended to set the argument one lower than the amount of logical processor cores the CPU has unless it has a single core, for example:

```
var nCores = Environment.ProcessorCount;
SynchronizationContextRemover.Init(nCores > 1 ? nCores - 1 : 1);
```

Bluetooth

Discovery of Bluetooth devices is disabled by default, it can be enabled by setting the EnableBluetooth proprety to true.

```
psCommSimpleWPF.EnableBluetooth = true;
```

To be able to use Bluetooth in your project you should reference the 32.Feet.NET NuGet package in your project (https://www.nuget.org/packages/32feet.NET/). Bluetooth Classic works on all supported platforms (Windows 7 SP1, Windows 8.1 and Windows 10).

Bluetooth Low energy only works on Windows 10. To enable Bluetooth Low Energy you should add references to the **PalmSens.Core.Windows.BLE.dll** library (included in the PSLibraries folder) in **your** project and the **PalmSens.Core.Simplified.WPF** project. Finally, you will also need to respectively uncomment the following lines in the **ScanDevices** and the **ScanDevicesAsync** methods of the **DeviceHandler** class in the **PalmSens.Core.Simplified.WPF** project.

```
discFuncs.Add(BLEDevice.DiscoverDevices);
discFuncs.Add(BLEDevice.DiscoverDevicesAsync());
```

5.2 Receive idle status readings

The readings of PalmSens can be read continuously using the **ReceiveStatus** event. The following information can be found in the **status** object that is received using this event:

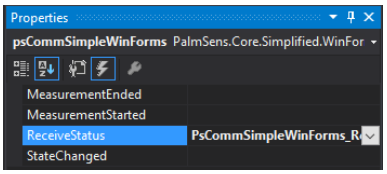
- AuxInput (auxiliary input in V, **Status.GetExtraValueAsAuxVoltage()**)
- Current (in uA, **Status.CurrentReading.Value** or **Status.CurrentReading.ValueInRange**)
- Current2 (in uA, in case a BiPot is used, **Status.GetExtraValueAsBiPotCurrent()**)
- Noise (**Status.Noise**)
- CurrentRange (the current range in use at the moment, **Status.CurrentReading.CurrentRange**)
- CurrentStatus (as **PalmSens.Comm.ReadingStatus** is ok, underload or overload, **Status.CurrentReading.ReadingStatus**)
- Potential (measured potential, **Status.PotentialReading.Value**)
- ReverseCurrent (the reverse current for SquareWave, **Status.ExtraValue**)

- PretreatmentPhaseStatus (None, Conditioning, Depositing or Equilibrating, **Status.PretreatmentPhase**)
- VoltageStatus (as **PalmSens.Comm.ReadingStatus** is ok, underload or overload, **Status.PotentialReading.ReadingStatus**)

Simplified PalmSens.Core:

```
psCommSimpleWPF.ReceiveStatus += PsCommSimpleWPF_ReceiveStatus;
```

Either subscribe to the ReceiveStatus event of the psCommSimpleWPF component via the designer or programmatically. It is not required to be connected to a device first.




```
private void psCommSimpleWPF_ReceiveStatus(object sender,
PalmSens.Comm.StatusEventArgs e)
{
	Status status = e.GetStatus();
}
```

The status is obtained from the event's **StatusEventArgs**.

PalmSens.Core:

```
comm.ReceiveStatus += Comm_ReceiveStatus;
```

To get the device's status updates subscribe to the **CommManager's ReceiveStatus** event after connecting to a device. (**comm** is a reference to the instance of the **CommManager** created when connecting to a device).

```
private void Comm_ReceiveStatus(object sender, StatusEventArgs e)
{
	Status status = e.GetStatus();
}
```

The status is obtained from the event's **StatusEventArgs**.

5.3 Manually controlling the device

Depending on your device's capabilities it can be used to set a potential/current and to switch current ranges. The potential can be set manually in potentiostatic mode and the current can be set in galvanostatic mode. The following examples show how to manually set a potential.

Simplfied PalmSens.Core:

```
psCommSimpleWPF.SetCellPotential(1f);
psCommSimpleWPF.TurnCellOn();
```

The psCommSimpleWPF component must be connected to a device before you can set its potential and control the cell. To turn the cell off call **psCommSimpleWPF.TurnCellOff()**.

PalmSens.Core:

```
comm.Potential = 1f;
comm.CellOn = true;
```

The device can be controlled using the CommManager that was created when connecting to the device. When the cell is off no potential will be set. (**comm** is a reference to the instance of the **CommManager** created when connecting to a device).

> **Device Capabilities**
>
> The capabilities of a connected device can either accessed via the **CommManager.Capabilities** or the **psCommSimpleWPF.Capabilities** property. The **DeviceCapabilities** contains properties such as its maximum potential, supported current ranges and support for specific features (galvanostat/impedance/bipot). The **DeviceCapabilities** can also be used to determine whether a certain method is compatible with a device using either **method.Validate(DeviceCapabilities)** or **psCommSimpleWPF.ValidateMethod(method)**.

5.4 Measuring

Starting a measurement is done by sending method parameters to a PalmSens/EmStat device. Events are raised when a measurement has been started/ended, when a new curve/scan is started/finished, and when new data is received during a measurement.

Simplified PalmSens.Core:

```
using PalmSens.Core.Simplified.Data;
```

Add these namespaces at the top of the document.

```
psCommSimpleWPF.MeasurementStarted += PsCommSimpleWPF_MeasurementStarted; //Raised
when a measurement begins

psCommSimpleWPF.MeasurementEnded += PsCommSimpleWPF_MeasurementEnded; //Raised when a
measurement is ended

psCommSimpleWPF.SimpleCurveStartReceivingData +=
PsCommSimpleWPF_SimpleCurveStartReceivingData; //Raised when a new SimpleCurve
instance starts receiving datapoints, returns a reference to the active SimpleCurve
instance
```

Subscribing to these events informs you on the status of a measurement and gives you references to the active **SimpleCurve** instances. (**psCommSimpleWPF** is a reference to the instance of the **psCommSimpleWPF** component in the Form).

```
SimpleMeasurement activeSimpleMeasurement = psCommSimpleWPF.Measure(method);
```

This line starts the measurement described in the instance of the method class. It returns a reference to the instance of the SimpleMeasurement, in the case of a connection error or invalid method parameters it returns null. Optionally, when using a multiplexer the channel can be specified as an integer, for example **psCommSimpleWPF.Measure (method,2)**. (**method** is a reference to an instance of the **PalmSens.Method** class, methods can be found in the namespace **PalmSens.Tecnhniques** more information on methods and their parameters is available in chapter 7).

```
SimpleCurve _activeSimpleCurve;

private void PsCommSimpleWPF_SimpleCurveStartReceivingData(object sender, SimpleCurve
activeSimpleCurve)
{
	_activeSimpleCurve = activeSimpleCurve;
	_activeSimpleCurve.NewDataAdded += _activeSimpleCurve_NewDataAdded;
	_activeSimpleCurve.CurveFinished += _activeSimpleCurve_CurveFinished;
}

private void _activeSimpleCurve_NewDataAdded(object sender,
PalmSens.Data.ArrayDataAddedEventArgs e)
{
	int startIndex = e.StartIndex;
	int count = e.Count;
	double[] newData = new double[count];
	(sender as SimpleCurve).YAxisValues.CopyTo(newData, startIndex);
}

private void _activeCurve_Finished(object sender, EventArgs e)
{
	_activeSimpleCurve.NewDataAdded -= _activeSimpleCurve_NewDataAdded;
	_activeSimpleCurve.Finished -= _activeSimpleCurve_Finished;
}
```

This code shows you how to obtain a reference to the instance of the active SimpleCurve currently receiving data from the **SimpleCurveStartReceivingData** event. It also shows how to subscribe this SimpleCurve's **NewDataAdded** and **CurveFinished** events and how these events can be used to retrieve the values of new data points from the Simple Curve as soon as they are available.

During a measurement the property **psCommSimpleWPF.DeviceState** property equals either **CommManager.DeviceState.Pretreatment** or **CommManager.DeviceState.Measurement**.

PalmSens.Core:

```
using PalmSens;
using PalmSens.Comm;
using PalmSens.Plottables;
```

Add these namespaces at the top of the document.

```
comm.BeginMeasurement += Comm_BeginMeasurement; //Raised when a measurement begins,
returns a reference to its measurement instance

comm.EndMeasurement += Comm_EndMeasurement; //Raised when a measurement is ended

comm.BeginReceiveCurve += Comm_BeginReceiveCurve; //Raised when a curve instance
begins receiving datapoints, returns a reference to the active curve instance
```

```
comm.BeginReceiveEISData += Comm_BeginReceiveEISData; //Raised when a EISData instance
begins receiving datapoints, returns a reference to the active EISData instance
```

Subscribing to these events informs you on the status of a measurement and gives you the references to the active measurement and curve instances. (**comm** is a reference to the instance of the **CommManager** created when connecting to a device).

```
comm.Measure(method);
```

This line starts the measurement described in the instance of the method class. Optionally, when using a multiplexer the channel can be specified as an integer, for example **comm.Measure(method,2)**. (**method** is a reference to an instance of the **PalmSens.Method** class, methods can be found in the namespace **PalmSens.Techniques** more information on methods and their parameters is available in chapter 7).

```
Measurement measurement;

private void Comm_BeginMeasurement(object sender, ActiveMeasurement newMeasurement)
{
	measurement = newMeasurement;
}
```

When the **BeginMeasurement** event is raised it returns a reference to the instance of the current measurement. Alternatively, this reference can be obtained from the **CommManager.ActiveMeasurement** property after the measurement has been started.

```
Curve _activeCurve;

private void Comm_BeginReceiveCurve(object sender, PalmSens.Plottables.CurveEventArgs
e)
{
	_activeCurve = e.GetCurve();
	_activeCurve.NewDataAdded += _activeCurve_NewDataAdded;
	_activeCurve.Finished += _activeCurve_Finished;
}

private void _activeCurve_NewDataAdded(object sender,
PalmSens.Data.ArrayDataAddedEventArgs e)
{
	int startIndex = e.StartIndex;
	int count = e.Count;
	double[] newData = new double[count];
	(sender as Curve).GetYValues().CopyTo(newData, startIndex);
}

private void _activeCurve_Finished(object sender, EventArgs e)
{
	_activeCurve.NewDataAdded -= _activeCurve_NewDataAdded;
	_activeCurve.Finished -= _activeCurve_Finished;
}
```

This code shows you how to obtain a reference to the instance of the active curve currently receiving data from the **BeginReceiveCurve** event. It also shows how to use the active curve's **NewDataAdded** and **Finished** events to retrieve the values of new data points from the curve as soon as they are available.

```
EISData _activeEISData;

private void Comm_BeginReceiveEISData(object sender, PalmSens.Plottables.EISData
eisdata)
{
	_activeEISData = eisdata;
	_activeEISData.NewDataAdded += _activeEISData_NewDataAdded; //Raised when new
	data is added
	_activeEISData.NewSubScanAdded += _activeEISData_NewSubScanAdded; //Raised when
	a new frequency scan is started
	_activeEISData.Finished += _activeEISData_Finished; //Raised when EISData is
	finished
}
```

When performing Impedance Spectroscopy measurements data points are stored in an instance of the **EISData** class and these events should be used similarly to those used for other measurements.

During a measurement the property **comm.Busy** is true.

Mains Frequency
To eliminate noise induced by other electrical appliances it is highly recommended to set your regional mains frequency (50/60 Hz) in the static property PalmSens.Method.PowerFreq.

5.5 MethodSCRIPT™

The MethodSCRIPT™ scripting language is designed to integrate our OEM potentiostat (modules) effortlessly in your hardware setup or product.

MethodSCRIPT™ allows developers to program a human-readable script directly into the potentiostat module by means of a serial (TTL) connection. The simple script language allows for running all supported electrochemical techniques and makes it easy to combine different measurements and other tasks.

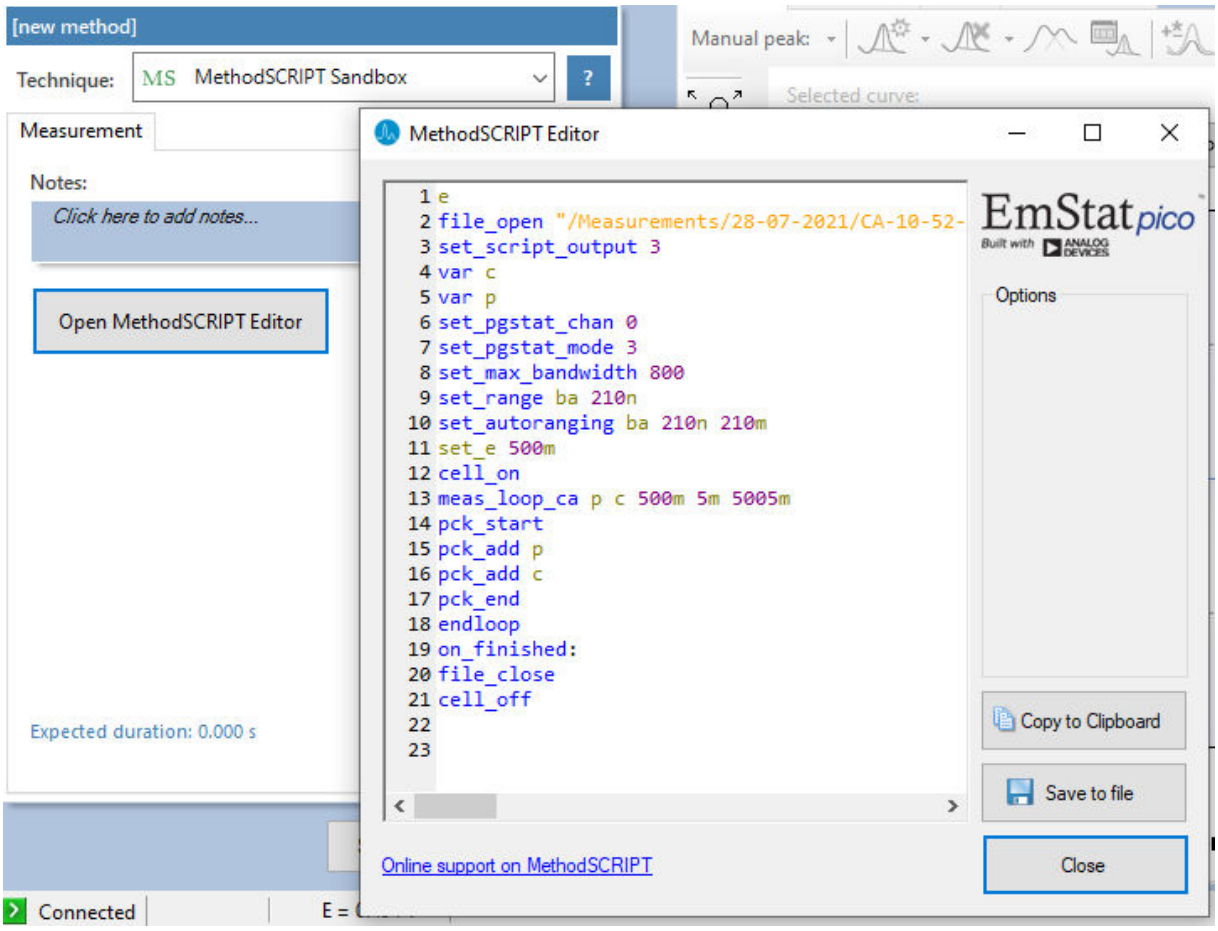
More script features include:

- Use of variables
- (Nested) loops
- Logging results to an SD card
- Digital I/O for example for waiting for an external trigger
- Reading auxiliary values like pH or temperature
- Going to sleep or hibernate mode

See for more information: www.palmsens.com/methodscript

5.5.1 Sandbox Measurements

PSTrace includes an option to make use MethodSCRIPT™ Sandbox to write and run scripts. This is a great place to test MethodSCRIPT™ measurements to see what the result would be. That script can then be used in the MethodScriptSandbox technique in the SDK as demonstrated below.



The following example contains 2 measurements, a LSV (meas_loop_lsv) and a CV (meas_loop_cv). Custom MethodSCRIPT™ can be run using the MethodScriptSandbox Method class.

```
var methodSCRIPT = @"e
var c
var p
set_pgstat_chan 1
set_pgstat_mode 0
set_pgstat_chan 0
set_pgstat_mode 3
set_max_bandwidth 400
set_range_minmax da -1 1
set_range ba 590u
set_autoranging ba 590n 590u
cell_on
meas_loop_lsv p c -500m 500m 10m 1
pck_start
pck_add p
pck_add c
pck_end
endloop
meas_loop_cv p c -500m -1 1 10m 1
pck_start
pck_add p
pck_add c
pck_end
endloop
on_finished:
cell_off
".Replace("\r", ""); // Remove all carriage return characters

var sandbox = new MethodScriptSandbox
{
    MethodScript = methodSCRIPT
};
```

As seen with the example above, MethodSCRIPT™ allows multiple measurements with a single script without having to send multiple scripts. The script string text must not contain the default newline characters (\r\n), these need to be replace just with the line feed or new line character (\n).

Please see section Measuring to see how to run methods. MethodSCRIPT™ must be run on the appropriate devices. You can check if a device is capable of running method script by casting the capabilities to MethodScriptDeviceCapabilities.

```
psCommSimpleWPF.Capabilities is MethodScriptDeviceCapabilities
```

SandboxMeasurements parse and store the variables sent in pcks. Curves are generated automatically for each meas_loop that defines a pck with two or more variables, scripts with multiple meas_loops will generate multiple curves. The first variable in the pck will be set as the x-axis and a curve is created for each subsequent variable in the pck. Please note that to plot data versus time you will need to a variable with the time to the pck.

5.5.2 Getter/Setter

The getter/setter allows you to control the IO pins of the devices that allow this, for example with the EmStat PICO. Here a simple getter/setter example:

Simplified PalmSens.Core:

```
Getter Example:
byte bitMask = 0b10101010; //A bitmask specifying which digital lines to read (0 =
 ignore, 1 = read).

//Synchronous
var result = psCommSimpleWPF.ReadDigitalLine(bitMask);
//Asynchronous
var result = await psCommSimpleWPF.ReadDigitalLineAsync(bitMask);

Setter Example:
byte bitMask = 0b11111111; //A bitmask specifying the output signal of the digital
 lines (0 = low, 1 = high).
var configGPIO = 0b10101010; //A bitmask specifying the the mode of digital lines
(0 = input, 1 = output).

//Synchronous
psCommSimpleWinForms.SetDigitalOutput(bitMask, configGPIO);
//Asynchronous
await psCommSimpleWinForms.SetDigitalOutputAsync(bitMask, configGPIO);
```

Using MethodSCRIPT™:
Setter Example:

```
string script = "e\nset_gpio_cfg 0b11111111 1\nset_gpio 0b10101010i\n\n";
//Synchronous
psCommSimpleWPF.StartSetterMethodScript(script);
//Synchronous
await psCommSimpleWPF.StartSetterMethodScriptAsync(script);
```

Getter Example:

```
string script = "e\nvar p\nset_gpio_cfg 0b11111111 0\nget_gpio p\npck_start\npck_a
dd p\npck_end\n\n";
//Synchronous
var result = psCommSimpleWPF.StartGetterMethodScript(script);
//Asynchronous
var result = await psCommSimpleWPF.StartGetterMethodScriptAsync(script);
```

PalmSens.Core:

Getter Example:

```
string script = "e\nvar p\nset_gpio_cfg 0b11111111 0\nget_gpio p\npck_start\npck_a
dd p\npck_end\n\n";

//Synchronous
if (Comm.ClientConnection is ClientConnectionMS connMS)
{
    string result = connMS.StartGetterMethodScript(script, timeout);
    return result;
}
//Asynchronous
if (comm.ClientConnection is ClientConnectionMS connMS)
{
    string result = await connMS.StartGetterMethodScriptAsync(script, timeout);
    return result;
}
```

Setter Example:

```
string script = "e\nset_gpio_cfg 0b11111111 1\nset_gpio 0b10101010i\n\n";
//Synchronous
if (comm.ClientConnection is ClientConnectionMS connMS)
    connMS.StartSetterMethodScript(script, timeout);
//Asynchronous
if (comm.ClientConnection is ClientConnectionMS connMS)
    await connMS.StartSetterMethodScriptAsync(script, timeout);
```

5.6 Disconnecting and disposing the device

The com port is **automatically closed** when the instance of the **CommManager** is disconnected or disposed.

Simplified PalmSens.Core:

```
psCommSimpleWPF.Disconnect(); or psCommSimpleWPF.Dispose();
```

PalmSens.Core:

```
comm.ClientConnection.Run(() => comm.Disconnect()).Wait() or comm.Disconnect(); or
comm.Dispose();
```

The **psCommSimpleWPF.Disconnected** event is raised when the device is disconnected, this can be particularly useful when the device was disconnected due to a communication error as the event also returns the respective exception as an argument in that case.

5.7 Possible causes of communication issues.

Communication issues can occur when certain commands are executed at the same time, i.e. starting a measurement and triggering a read potential at the same time will result in the device receiving commands in an incorrect order. These issues typically arise when a timer is used, when using multiple threads, and when invoking commands in a callback on one on the **psCommSimpleWPF/psMultiCommSimpleWPF** events.

When using the simplified core wrapper communication issues are prevented as much as possible. Using commands to control the device from your **psCommSimpleWPF/psMultiCommSimpleWPF** event callbacks is blocked, to prevent communication issues. With the asynchronous methods it is possible to control your device from one of these callbacks as the command will be delayed and run after completion of the previous command, however, as it can be run at a later point in time it is important to check whether all conditions for executing the command are still true. This can be adjusted in the **PSCommSimple.cs** or **PSMultiCommSimple.cs** files in the **PalmSens.Core.Simplified** project.

When using the **PalmSens.Core** directly useful aids to prevent threading issues are the **comm.ClientConnection.Run** and **comm.ClientConnection.Run<T>** methods. These assure the commands are run on the correct context which prevents communication errors due to multiple threads communicating with the device simultaneously. When using multiple threads it is highly recommended to use these helper methods when invoking methods that communicate with the device (i.e. Measure, Current, Potential, CurrentRange and CellOn) from a different thread.

Setting a value safely:

```
comm.ClientConnection.Run(() => { comm.CellOn = true; }).Wait();
```

or when connected to a device asynchronously

```
await comm.ClientConnection.RunAsync(() => comm.SetCellOnAsync(true));
```

Getting a value safely:

```
Task<float> GetPotentialTask = comm.ClientConnection.Run<float>(new Task<float>(() =>
{ return comm.Potential; }));
GetPotentialTask.Wait();
float potential = GetPotentialTask.Result;
```

or when connected to a device asynchronously

```
float potential = comm.ClientConnection.RunAsync<float>(() =>
comm.GetPotentialAsync());
```

6 Data analysis and manipulation using the simplified wrapper

This chapter covers how data can be retrieved from a measurement and how to manipulate and analyze the data. This is also detailed in the data, EIS fit and peak detect examples.

6.1 Obtaining the measured values

When a measurement is started by calling the **psCommSimpleWPF.Measure** method a reference to the instance of that **SimpleMeasurement** class is returned. The **SimpleMeasurement** class contains references to instances of the **SimpleCurve** class in the **SimpleMeasurement.SimpleCurveCollection** property, this class is used to analyze and manipulate the data. New instances of the **SimpleCurve** class can be generated by calling the **SimpleMeasurement.NewSimpleCurve** method, this method will generate new **SimpleCurves** based on the available **DataArrayTypes** in the instance **SimpleMeasurement** (In the case of a Cyclic Voltammetry or Mux measurements multiple SimpleCurves can be generated). The **SimpleMeasurement.AvailableDataTypes** property contains a list of the available **DataArrayTypes** in the measurement. In the case you would like to plot a different **SimpleCurve** than the default one it is possible to call the **SimpleMeasurement.NewSimpleCurve** method directly after it has started to generate another **SimpleCurve** with different units/values.

```
List<SimpleCurve> chargeCurves =
simpleMeasurement.NewSimpleCurve(PalmSens.Data.DataArrayType.Time,
PalmSens.Data.DataArrayType.Charge, "Charge/Time"); //Get Charge over Time curves
```

To access the raw values in the form of an array of doubles use either the **SimpleCurve.XAxisValues** or the **SimpleCurve.YAxisValues** properties.

```
double[] xValues = simpleCurve.XAxisValues;
double firstYValue = simpleCurve.YAxisValue(0); //Get value of the Y Axis a specified
index
```

6.2 Smoothing/Filtering

To smooth a **SimpleCurve** call one of the **SimpleCurve.Smooth** methods. These methods use the Savitsky-Golay filter. The arguments can be either the **SmoothLevel** enumerator or an int specifying the window size, i.e. a window of 4 will filter based on the 4 adjacent points in both directions.

```
SimpleCurve smoothedCurve = simpleCurve.Smooth(SmoothLevel.Medium);
SimpleCurve smoothedCurve2 = simpleCurve.Smooth(25); //Smooth using the specified
window size
```

6.3 Baseline Subtraction

A baseline correction can be performed by subtracting the SimpleCurve of a baseline measurement from your SimpleCurve or by determining the moving average baseline of the SimpleCurve by calling **SimpleCurve.MovingAverageBaseline**. To subtract one curve from another call **SimpleCurve.Subtract**.on the SimpleCurve you would like to subtract the other curve from which must be passed on as the argument.

```
SimpleCurve movingAverageBaseline = simpleCurve.MovingAverageBaseline(); //Get the
moving average baseline curve
SimpleCurve baselineSubtractedCurve = simpleCurve.Subtract(movingAverageBaseline);
//Get the simple curve with the subtracted baseline
```

6.4 Basic operations

The **SimpleCurve** class supports other basic operations such as:

- Addition; **SimpleCurve.Add()**
- Subtraction; **SimpleCurve.Subtract()**
- Multiplication; **SimpleCurve.Multiply()**
- Exponentiation; **SimpleCurve.Exponentiate()**
- Differentiation; **SimpleCurve.Differentiate()**
- Integration; **SimpleCurve.Integrate()**
- Base 10 Logarithm; **SimpleCurve.Log10()**
- Average; **SimpleCurve.Average()**
- Sum; **SimpleCurve.Sum()**
- Minimum; **SimpleCurve.Minimum()**
- Maximum; **SimpleCurve.Maximum()**

6.5 Peak and level detection

The **SimpleCurve** class contains functions for detecting peaks and levels using our algorithms.

There are three peak detection algorithms; the default algorithm detects peaks using the curve's derivative, the shoulder algorithm can detect peaks that are on a slope and missed by the default algorithm, and the LSV/CV algorithm is specifically designed for detecting peaks in Linear Sweep and Cyclic Voltammetry. The detected peaks are added to **SimpleCurve.Peaks**, an IEnumerable collection of the **Peak** interface. The **Peak** interface describes the properties of the peak (i.e. the peak potential, current, height, width, etc.). Examples of the peak detection are also provided in the PSSDKDataExample and the PSSDKPlotPeakDetectionExample projects.

```
activeSimpleCurve.DetectPeaks(0.01, 0.05, true, false); //Detect peaks with a minimum
width of 0.01V, a minimum height of 0.05µA, discard any existing peaks, using the
default algorithm
await activeSimpleCurve.DetectPeaksAsync(0.01, 0.05, true, PeakTypes.LSVCV); //Detect
peaks with a minimum width of 0.01V, a minimum height of 0.05µA, discard any existing
peaks, using the Linear Sweep / Cyclic Voltammetry algorithm

//Get peak properties from the first peak
double peakHeight = activeSimpleCurve.Peaks[0].PeakValue;
double peakPotential = activeSimpleCurve.Peaks[0].PeakX;
```

Level detection works similar to peak detection, except that the results are stored in **SimpleCurve.Levels**, an IEnumerable collection of the **Level** class.

```
await activeSimpleCurve.DetectLevelsAsync(0.5,0.05,true); //Detect levels with a
minimum width of 0.5s, a minimum height of 0.05µA, discard any existing levels

//Get level properties from the first level
double levelBegin = activeSimpleCurve.Levels[0].LeftX;
double levelEnd = activeSimpleCurve.Levels[0].RightX;
double levelCurrent = activeSimpleCurve.Levels[0].LevelY;
```

6.6 Equivalent circuit fitting

The **SimpleCurve.FitEquivalentCircuit** function allows you to fit an equivalent circuit model on your data. The simplest way to fit a circuit using the default fit settings is using the following version of the FitEquivalentCircuit function:

```
FitResult fitResult = await activeSimpleCurve.FitEquivalentCircuit("R(RC)", new
double[] {
       100, //The initial value for the solution resistance (series resistor)
       8000, //The initial value for the charge transfer resistance (parallel resistor)
       1e-8 //The initial value for the double layer capacitance (parallel capacitor)
}); //Fit a Randles circuit using the specified inital values and default fit options

//Get fit results
double solutionResistance = fitResult.FinalParameters[0];
double chargeTransferResistance = fitResult.FinalParameters[1];
double doubleLayerCapacitance = fitResult.FinalParameters[2];
```

To change the default fit options use the following function in combination with the **CircuitModel** and **FitOptionsCircuit** classes.

```
//Change model parameters
CircuitModel circuitModel = new CircuitModel();
circuitModel.SetEISdata(_activeMeasurement.Measurement.EISdata[0]); //Sets reference
to measured data
circuitModel.SetCircuit("R(RC)"); //Sets the circuit defined in the CDC code string,
in this case a Randles circuit

//Change bounds and initial value of the solution resistance in the Randles circuit
Parameter p = circuitModel.InitialParameters[0];
p.MaxValue = 1e6; //Set 1e6 Ω as the upper bound
p.MinValue = 1e4; //Set 1e4 Ω as the lower bound
p.Value = 1e5; //Set 1e5 Ω as the initial value

//Override default Fit Options
FitOptionsCircuit fitOptions = new FitOptionsCircuit();
fitOptions.Model = circuitModel; //Specift model to fit
fitOptions.RawData = _activeMeasurement.Measurement.EISdata[0]; //Sets reference to
measured data
fitOptions.MaxIterations = 1000; //The maximum number of iterations, 500 by default
fitOptions.MinimumDeltaErrorTerm = 1e-12; //The minimum delta in the error term (sum
of squares difference between model and data), default is 1e-9

FitResult fitResult = await activeSimpleCurve.FitEquivalentCircuit(circuitModel,
fitOptions); //Fit the circuit defined in the CircuitModel and the options specified
in the FitOptions

//Get fit results
double solutionResistance = fitResult.FinalParameters[0];
double chargeTransferResistance = fitResult.FinalParameters[1];
double doubleLayerCapacitance = fitResult.FinalParameters[2];
```

The PSSDKPlotEISFit example projects also demonstrate how to use the equivalent circuit fitting.

7 Appendix A: Parameters for each technique

All applicable parameters for each technique can be found here. For the inheritance hierarchy of the the techniques, see section 3 in this document. See section 'Available techniques' in the PSTrace manual for more information about the techniques.

Each technique is identified by a specific integer value. This integer value can be used to create a class derived from the corresponding technique, as follows:

```
PalmSens.Method.FromTechniqueNumber(integervalue)
```

The integer values are indicated in this appendix inside the brackets [ ] following each technique name.

The techniques are also directly available from the **PalmSens.Techniques** namespace.

Please refer to the PSTrace manual for explanations and expected values for each parameter.

7.1 Common properties

Property	**Description**	**Type**
Technique	The technique number used in the firmware	System.Int
Notes	Some user notes for use with this method	System.String
StandbyPotential	Standby Potential (for use with cell on after measurement)	System.Float
StandbyTime	Standby time (for use with cell on after measurement)	System.Float
CellOnAfterMeasurement	Enable/disable cell after measurement	System.Boolean
MinPeakHeight	Determines the minimum peak height in µA. Peaks lower than this value are neglected.	System.Float
MinPeakWidth	The minimum peak width, in the unit of the curves X axis. Peaks narrower than this value are neglected.	System.Float
SmoothLevel	The smoothlevel to be used. -1 = none 0 = no smooth (spike rejection only) 1 = 5 points 2 = 9 points 3 = 15 points 4 = 25 points	System.Int
Ranging	Ranging information, settings defining the minimum/maximum/starting current range	PalmSens.Method.Ranging
PowerFreq	Adjusts sampling on instrument to account for mains frequency. It accepts two values: 50 for 50Hz 60 for 60Hz	System.Int

7.2 Pretreatment settings

The following properties specify the measurements pretreatment settings:

Property	Description	Type
ConditioningPotential	Conditioning potential in volt	System.Float
ConditioningTime	Conditioning duration in seconds	System.Float
DepositionPotential	Deposition potential in volt	System.Float
DepositionTime	Deposition duration in seconds	System.Float
EquilibrationTime	Equilibration duration in seconds. BeginPotential is applied during equilibration and the device switches to the appropriate current range	System.Float

7.3 Linear Sweep Voltammetry (LSV) [0]

Class: Palmsens.Techniques.LinearSweep

Property	Description	Type
BeginPotential	Potential where scan starts.	System.Float
EndPotential	Potential where measurement stops.	System.Float
StepPotential	Step potential	System.Float
Scanrate	The applied scan rate. The applicable range depends on the value of E step since the data acquisition rate is limited by the connected instrument.	System.Float

7.4 Differential Pulse Voltammetry (DPV) [1]

Class: Palmsens.Techniques.DifferentialPulse

Property	Description	Type
BeginPotential	Potential where scan starts.	System.Float
EndPotential	Potential where measurement stops.	System.Float
StepPotential	Step potential	System.Float
Scanrate	The applied scan rate. The applicable range depends on the value of E step since the data acquisition rate is limited by the connected instrument.	System.Float
PulsePotential	Pulse potential	System.Float
PulseTime	The pulse time	System.Float

7.5 Square Wave Voltammetry (SWV) [2]

Class: Palmsens.Techniques.SquareWave

Property	Description	Type
BeginPotential	Potential where scan starts.	System.Float
EndPotential	Potential where measurement stops.	System.Float
StepPotential	Step potential	System.Float
PulseAmplitude	Amplitude of square wave pulse. Values are half peak-to-peak.	System.Float
Frequency	The frequency of the square wave	System.Float

7.6 Normal Pulse Voltammetry (NPV) [3]

Class: Palmsens.Techniques.NormalPulse

Property	Description	Type
BeginPotential	Potential where scan starts.	System.Float
EndPotential	Potential where measurement stops.	System.Float
StepPotential	Step potential	System.Float
Scanrate	The applied scan rate. The applicable range depends on the value of E step since the data acquisition rate is limited by the connected instrument.	System.Float
PulseTime	The pulse time	System.Float

7.7 AC Voltammetry (ACV) [4]

Class: Palmsens.Techniques.ACVoltammetry

Property	Description	Type
BeginPotential	Potential where scan starts.	System.Float
EndPotential	Potential where measurement stops.	System.Float
StepPotential	Step potential	System.Float
SineWaveAmplitude	Amplitude of sine wave. Values are RMS	System.Float
Frequency	The frequency of the AC signal	System.Float

7.8 Cyclic Voltammetry (CV) [5]

Class: Palmsens.Techniques.CyclicVoltammetry

Property	Description	Type
BeginPotential	Potential where scan starts and stops.	System.Float
Vtx1Potential	First potential where direction reverses.	System.Float
Vtx2Potential	Second potential where direction reverses.	System.Float
StepPotential	Step potential	System.Float
Scanrate	The applied scan rate. The applicable range depends on the value of E step since the data acquisition rate is limited by the connected instrument.	System.Float
nScans	The number of repetitions for this scan	System.Float

7.8.1 Fast Cyclic Voltammetry Scans

Class: Palmsens.Techniques.FastCyclicVoltammetry

Outdated class. PalmSens 3 and 4 only. CV's with sampling over 5000 data points per second, use the regular **Palmsens.Techniques.CyclicVoltammetry()** constructor instead.

7.9 Chronopotentiometric Stripping (SCP) [6]

Class: PalmSens.Techniques.ChronoPotStripping

Property	Description	Type
EndPotential	Potential where measurement stops.	System.Float
MeasurementTime	The maximum measurement time. This value should always exceed the required measurement time. It only limits the time of the measurement. When the potential response is erroneously and E end is not found within this time, the measurement is aborted.	System.Float
AppliedCurrentRange	The applied current range	PalmSens.CurrentRange
Istrip	If specified as 0, the method is called chemical stripping otherwise it is constant current stripping. The current is expressed in the applied current range.	System.Float

7.10 Chronoamperometry (CA) [7]

Class: PalmSens.Techniques.AmperometricDetection

Property	Description	Type
Potential	Potential during measurement.	System.Float
IntervalTime	Time between two current samples.	System.Float
RunTime	Total run time of scan.	System.Float

7.11 Pulsed Amperometric Detection (PAD) [8]

Class: PalmSens.Techniques.PulsedAmpDetection

Property	Description	Type
Potential	The dc or base potential.	System.Float
PulsePotentialAD	Potential in pulse. Note that this value is not relative to dc/base potential, given above.	System.Float
PulseTime	The pulse time.	System.Float
tMode	DC: I(dc) measurement is performed at potential E pulse: I(pulse) measurement is performed at potential E pulse differential: I(dif) measurement is I(pulse) - I(dc)	PalmSens.Techniques.PulsedAmpDetection.enumMode
IntervalTime	Time between two current samples.	System.Float
RunTime	Total run time of scan.	System.Float

7.12 Fast Amperometry (FAM) [9]

Class: PalmSens.Techniques.FastAmperometry

Property	Description	Type
EqPotentialFA	Equilibration potential at which the measurement starts.	System.Float
Potential	Potential during measurement.	System.Float
IntervalTimeF	Time between two current samples.	System.Float
RunTime	Total run time of scan.	System.Float

7.13 Chronopotentiometry (CP) [10]

Class: PalmSens.Techniques.Potentiometry

Property	Description	Type
Current	The current to apply. The unit of the value is the applied current range. So if 10 uA is the applied current range and 1.5 is given as value, the applied current will be 15 uA.	System.Float
AppliedCurrentRange	The applied current range.	PalmSens. CurrentRange
RunTime	Total run time of scan.	System.Float
IntervalTime	Time between two potential samples.	System.Float

7.13.1 Open Circuit Potentiometry (OCP)

Class: PalmSens.Techniques.OpenCircuitPotentiometry

The same as setting the Current to 0.

Property	Description	Type
RunTime	Total run time of scan.	System.Float
IntervalTime	Time between two potential samples.	System.Float

7.14 Multiple Pulse Amperometry (MPAD) [11]

Class: PalmSens.Techniques.MultiplePulseAmperometry

Property	Description	Type
E1	First potential level in which the current is recorded	System.Float
E2	Second applied potential level	System.Float
E3	Third applied potential level	System.Float
t1	The duration of the first applied potential	System.Float
t2	The duration of the second applied potential	System.Float
t3	The duration of the third applied potential	System.Float
RunTime	Total run time of scan.	System.Float

7.15 Electrochemical Impedance Spectroscopy (EIS)

Class: PalmSens.Techniques.ImpedimetricMethod

The most common properties are described first. These are used for a typical EIS measurement, a scan over a specified range of frequencies (i.e. using the default properties **ScanType** = **ImpedimetricMethod**.
enumScanType.FixedPotential and **FreqType** = **ImpedimetricMethod.enumFrequencyType.Scan**). The additional properties used for a **TimeScan** and a **PotentialScan** are detailed separately in next sections.

Property	Description	Type
ScanType	Scan type specifies whether a single or multiple frequency scans are performed. When set to FixedPotential a single scan will be performed, this is the recommended setting. **The TimeScan and PotentialScan are not fully supported in the SDK**, we highly recommend you to implement yourself. A TimeScan performs repeated scans at a given time interval within a specified time range. A PotentialScan performs scans where the DC Potential of the applied sine is incremented within a specified range. A PotentialScan should not be performed versus the OCP.	ImpedimetricMethod. enumScanType

Potential	The DC potential of the applied sine	System.Float
Eac	The amplitude of the applied sine in RMS (Root Mean Square)	System.Float
FreqType	Frequency type specifies whether to perform a scan on a range of frequencies or to measure a single frequency. The latter option can be used in combination with a TimeScan or a PotentialScan.	ImpedimetricMethod.enumFrequencyType
MaxFrequency	The highest frequency in the scan, also the frequency at which the measurement is started	System.Float
MinFrequency	The lowest frequency in the scan	System.Float
nFrequencies	The number of frequencies included in the scan	System.Int
SamplingTime	Each measurement point of the impedance spectrum is performed during the period specified by SamplingTime. This means that the number of measured sine waves is equal to SamplingTime * frequency. If this value is less than 1 sine wave, the sampling is extended to 1 / frequency. So for a measurement at a frequency, at least one complete sine wave is measured. Reasonable values for the sampling are in the range of 0.1 to 1 s.	System.Float
MaxEqTime	The impedance measurement requires a stationary state. This means that before the actual measurement starts, the sine wave is applied during MaxEqTime only to reach the stationary state. The maximum number of equilibration sine waves is however 5. The minimum number of equilibration sines is set to 1, but for very low frequencies, this time is limited by MaxEqTime. The maximum time to wait for stationary state is determined by the value of this parameter. A reasonable value might be 5 seconds. In this case this parameter is only relevant when the lowest frequency is less than 1/ 5 s so 0.2 Hz.	System.Float

7.15.1 Time Scan

In a Time Scan impedance spectroscopy measurements are repeated for a specific amount of time at a specific interval. The SDK does not support this feature fully, we recommend you to design your own implementation for this that suits your demands.

Property	Description	Type
RunTime	RunTime is not the total time of the measurement, but the time in which a measurement iteration can be started. If a frequency scan takes 18 seconds and is measured at an interval of 19 seconds for a RunTime of 40 seconds three iterations will be performed.	System.Float
IntervalTime	IntervalTime specifies the interval at which a measurement iteration should be performed, however if a measurement iteration takes longer than the interval time the next measurement will not be triggered until after it has been completed.	System.Float

7.15.2 Potential Scan

In a Potential Scan impedance spectroscopy measurements are repeated over a range of DC potential values. The SDK does not support this feature fully, we recommend you to design your own implementation for this that suits your demands.

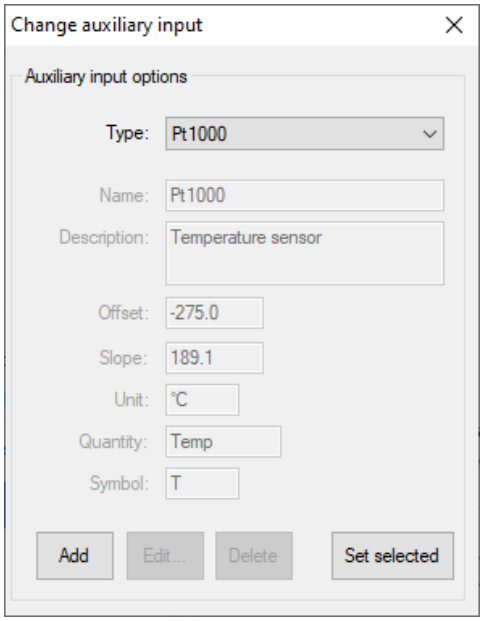
Property	Description	Type
BeginPotential	The DC potential of the applied sine wave to start the series of iterative measurements at.	System.Float
EndPotential	The DC potential of the applied sine wave at which the series of iterative measurements ends.	System.Float
StepPotential	The size of DC potential step to iterate with.	System.Float

7.16 Recording extra values (BiPot, Aux, CE Potential…)

The **PalmSens.Method.ExtraValueMsk** property allows you to record an additional value during your measurement. Not all techniques support recording extra values, the **SupportsAuxInput** and **SupportsBipot** properties are used to indicate whether a technique supports the recording of these values. The default value for **PalmSens.Method.ExtraValueMsk** is **PalmSens.ExtraValueMask.None**.

- None, no extra value recorded (default)
- Current
- Potential
- WE2, record BiPot readings (The behavior of the second working electrode is defined with the method's **BipotModePS** property. **EnumPalmSensBipotMode.Constant** sets it to a fixed potential and **EnumPalmSensBipotMode.Offset** sets it to an offset of the primary working electrode. The value in Volt of the fixed or offset potential is defined with the method's **BiPotPotential** property.)

- AuxInput, similar to PSTrace it is possible to configure the readings of the auxilliary input. Using the **PalmSens.AuxInput.AuxiliaryInput** class you can assign a name, offset, gain and unit to the auxilliary input. The following example demonstrates how to set up the Pt1000 temperature sensor from PSTrace.




```
psCommSimpleWPF.comm.AuxInputSelected = new PalmSens.AuxInput.AuxiliaryInputType(true, "Pt1000", "Temperature sensor", -275f, 189.1f,
             new PalmSens.Units.Temperature());
```

  The can be ignored and set to true, the second argument is the name, third is the description, fourth the offset, fifth the slope and the final argument is an instance of one of the unit classes in the **PalmSens.Units** namespace.
- Reverse, record reverse current as used by Square Wave Voltammetry
- PolyStatWE, not supported in the PalmSens SDK
- DCcurrent, record the DC current as used with AC Voltammetry
- CEPotential, PalmSens 4 only

The PSSDKBiPotAuxExample example project demonstrates how to record extra values.

7.17 Multiplexer

The **PalmSens.Method** class is also used to specify the multiplexer settings for sequential and alternating measurements. Alternating multiplexer measurements restricted to the chronoamperometry and chronopotentiometry techniques.

The enumerator property **PalmSens.Method.MuxMethod** defines the type multiplexer measurement.

```
methodCA.MuxMethod = MuxMethod.None; //Default setting, no multiplexer
methodCA.MuxMethod = MuxMethod.Alternatingly;
methodCA.MuxMethod = MuxMethod.Sequentially;

//The channels on which to measure are specified in a boolean array
PalmSens.Method.UseMuxChannel: methodCA.UseMuxChannel = new bool[] { true, true, false, false, false, false, false, true };
```

The code above will perform a measurement on the first two and last channels of an 8-channel multiplexer. For a 16-channel multiplexer you would also need to assign true or false to the last 8 channels.

Alternating multiplexer measurement can only measure on successive channels and must start with the first channel (i.e. it is possible to alternatingly measure on channels 1 through 4 but it is not possible to alternatingly measure on channel 1, 3 and 5). The multiplexer functionality is demonstrated in the PSSDKMultiplexerExample project.

7.17.1 Multiplexer settings

When using a MUX8-R2 multiplexer the multiplexer settings must be set digitally instead of via the physical switches on the earlier multiplexer models. The type of multiplexer should be specified in the connected device's capabilities, when the multiplexer is connected before connecting to the software the capabilities are updated automatically. Otherwise, when using the MUX8-R2 the **PalmSens.Devices.DeviceCapabilities.MuxType** should be set to **PalmSens.Comm.MuxType.Protocol** manually or by calling **PalmSens.Comm.CommManager.ClientConnection.ReadMuxInfo**, **PalmSens.Comm.CommManager.ClientConnection.ReadMuxInfoAsync** when connected asynchronously.

For the MUX8-R2 the settings for a measurement are set in **PalmSens.Method.MuxSett** property with an instance of the **PalmSens.Method.MuxSettings** class. For manual control these settings can be set using the **PalmSens.Comm.ClientConnection.SetMuxSettings** function, **PalmSens.Comm.ClientConnection.SetMuxSettingsAsync** when connected asynchronously.

```
method.MuxSett = new Method.MuxSettings(false)
{
        CommonCERE = false,
        ConnSEWE = false,
        ConnectCERE = true,
        OCPMode = false,
        SwitchBoxOn = false,
        UnselWE = Method.MuxSettings.UnselWESetting.FLOAT
};
```

7.18 Versus OCP

The versus open circuit potential settings (OCP) are defined in the **PalmSens.Method.OCPmode**, **PalmSens.Method.OCPMaxOCPTime**, and **PalmSens.Method.OCPStabilityCriterion** properties. The OCPmode is a bitmask specifies which of the following technique dependent properties or combination thereof will be measured versus the OCP potential:

- Linear Sweep Voltammetry:
    1. BeginPotential = 1
    2. EndPotential = 2
- (Fast) Cyclic Voltammetry
    1. Vtx1Potential = 1
    2. Vtx2Potential = 2
    3. BeginPotential = 4
- Chronoamperometry
    1. Potential = 1
- Impedance Spectroscopy (Fixed potential and Time Scan)
    1. Potential = 1
- Impedance Spectroscopy (Potential Scan)
    1. BeginPotential = 1
    2. EndPotential = 2

The progress and result of the versus OCP measurement step are reported in the **PalmSens.Comm.MeasureVersusOCP** class, which can be obtained by subscribing to the

PalmSens.Comm.CommManager.DeterminingVersusOCP event which is raised when the versus OCP measurement step is started.

```
//Defining versus OCP measurement step for a Cyclic Voltammetry measurement
_methodCV.OCPmode = 7; //Measure the (Vtx1Potential) 1 + (Vtx2Potential) 2 +
(BeginPotential) 4 = 7 versus the OCP potential
_methodCV.OCPMaxOCPTime = 10; //Sets the maximum time the versus OCP step can take to
10 seconds
_methodCV.OCPStabilityCriterion = 0.02f; //The OCP measurement will stop when the
change in potential over time is less than 0.02mV/s, when set to 0 the OCP measurement
step will always run for the OCPMaxOCPTime
```

7.19 Properties for EmStat Pico

There are two method parameters specific to the EmStat Pico. The **PalmSens.Method.PGStatMode** property sets the mode in which the measurement should be run, low power, high speed or max range. This mode can be set for all techniques but Electrochemical Impedance Spectroscopy. The second property is **PalmSens.Method.SelectedPotentiostatChannel** which let you choose on which channel the EmStat Pico should run the measurement.