ElcoMaster Xamarin SDK Getting Started

The ElcoMaster Xamarin SDK will allow you to integrate a range of Made for iPod/iPhone/iPad Elcometer gauges into your our Xamarin iOS, Xamarin Android, Windows .Net and Mono Linux applications. The current SDK supports the following Made for iPod/iPhone/iPad gauges & Bluetooth 5.0 gauges:

- Elcometer 456C & 456TV (Standard and Top Models)
- Elcometer 224C (Top Model)
- Elcometer 510 (Top Model)
- Elcometer 480 (Top Models)
- Elcometer MTG6/MTG8
- Elcometer PTG8
- Elcometer 311C (Top Model)
- Elcometer 415C (Top Model)
- Elcometer 130 (Profiler and Top Models)
- Elcometer 500
- Elcometer 319 (Top Model)

This getting started guide will give an overview of how to connect, download batches and live measurements from these gauges.

Within the SDK zip there are multiple demo projects for iOS, Android and Windows (both Win32 & UWP) that should help to understand the processes.

To use it requires Microsoft Visual Studio for Windows or Mac, or MonoDevelop on Linux.

Bluetooth Classic & Bluetooth 5.0 (LE)

The Bluetooth technology used by various gauges will vary. Bluetooth 5.0 LE gauges will be identified as Bluetooth 5.0 compatible in the gauges Bluetooth screen.

The process to connect to the gauge is different for both technologies.

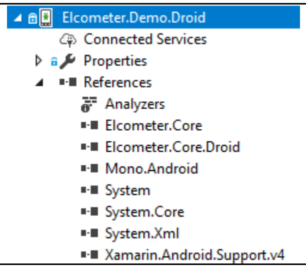
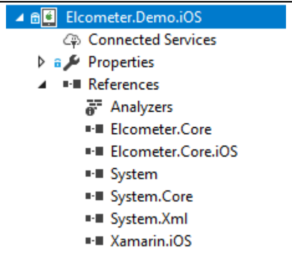
Also note that on Microsoft Windows devices only the .Net 6.0 library supports both Bluetooth LE and Classic.

Installation & Requirements

The SDK library is supplied as a common PCL assembly and a platform specific assembly.

Native Xamarin & Xamarin Forms Shared Project

For native Xamarin and Xamarin Forms shared projects you will need to add both the PCL (Elcomter.Core.dll) and platform specific library (Elcometer.Core.Droid.dll for Android & Elcometer.Core.iOS.dll for iOS) to your project references, .e.g.

 Elcometer.Demo.Droid Connected Services Properties References Analyzers Elcometer.Core Elcometer.Core.Droid Mono.Android System System.Core System.Xml Xamarin.Android.Support.v4	 Elcometer.Demo.iOS Connected Services Properties References Analyzers Elcometer.Core Elcometer.Core.iOS System System.Core System.Xml Xamarin.iOS
Android	iOS

For Windows desktop projects the PCL (Elcometer.Core.dll) and windows class library (Elcometer.Core.Windows.dll) need to be added.

For Linux projects the PCL (Elcometer.Core.dll) and class library (Elcometer.Core.Linux.dll) need to be added.

All Elcometer SDK assemblies are located in the Assemblies folder in the SDK zip file.

Xamarin Forms PCL

For Xamarin Forms PCL projects you will need to add the Elcometer.Core.dll assembly to your PCL project references and the appropriate platform specific library to your platform project references.

Permissions

Both Android and iOS need specific permissions to be assigned in the project. For iOS the following keys needs to added to the info.plist

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
        <string>com.elcometer.gaugecom1</string>
</array>
<key>NSBluetoothPeripheralUsageDescription</key>
<string>We use Bluetooth to connect to the various Elcometer inspection devices.</string>
<key>NSBluetoothAlwaysUsageDescription</key>
<string>We use Bluetooth to connect to the various Elcometer inspection devices.</string>
```

Up to Android 11 the BLUEOOTH, BLUETOOTH_ADMIN and ACCESS_COARSE_LOCATION permissions need to be enabled in the project properties. The course location is required to allow the ability to retrieve the nearby Bluetooth devices when showing the device picker.

For Android 12+ BLUETOOTH_SCAN & BLUETOOTH_CONNECT permissions are required.

For Android 6.0+ the AccessCoarseLocation permission will also need to be dynamically requested using the standard Android activity APIs *CheckSelfPermission* and *RequestPermission*. Check the Android demo projects for more information.

For more detailed information consult the Apple and Google developer documentation.

Initialisation

Both the platform specific assemblies offer a singleton that can be initialised and used to access the SDK services. Alternatively if you are using a MVVM framework with dependency injection you could initialise the services directly in your IOC container.

Using the Singleton

The singleton on both platforms can be initialised with the following:

```
ElcometerCore.Instance.Initialise();
```

Access to the *GaugeService* and *MessagingService* can then be achieved by accessing via the *ElcometerCore.Instance.GaugeService* & *ElcometerCore.Instance.MessagingService* properties.

Using an IOC Container

Using an IOC container you will need to register all the services manually. The exact syntax of the registration of the service will differ depending on the IOC container you are using, but it would look something like the following.

On the platform project side you would register the platform specific services.

```
container.RegisterSingleton<IConnectionService, ConnectionService>();
container.RegisterSingleton<IPlatformService, PlatformService>();
```

On the PCL project side you would register the platform independent services.

```
container.RegisterSingleton<IGaugeTypeService, GaugeTypeService>();
container.RegisterSingleton<IMessagingService, MessagingService>();
container.RegisterSingleton<IGaugeService, GaugeService>();
container.RegisterSingleton<ISimpleBatchService, SimpleBatchService>();

// Register the available batch types that can be downloaded
ElcometerCoreRegister.RegisterBatches(container.Resolve<ISimpleBatchService>());

// Register the gauges that can be communicated with
ElcometerCoreRegister.RegisterGaugeTypes(container.Resolve<IGaugeTypeService>());
```

The *ElcometerCoreRegister* is a static class that is provided in the SDK PCL assembly. Once the services are registered above you can use your IOC container to do constructor injection into your ViewModels.

Connecting to a Gauge

Connection to a gauge is achieved by using the platform specific picker dialog in the *ConnectionService* class.

Android:

```
public async Task ShowPickerClassic(Activity activity, string title, string contains = null)
```

```
public async Task ShowPickerLE(Activity activity, string title, string contains = null)
```

iOS

```
public async Task ShowPickerClassic(string contains = null)
```

```
public async Task ShowPickerLE(string contains = null)
```

Windows Desktop:

```
public void ShowPicker(IWin32Window parent, string contains = null)
```

Linux:

```
picker is not supported - see end of document regards linux support.
```

An additional *IBluetoothPickerService service* is available in the Xamarin.Forms demo which shows how to wrap these platform specific methods into a cross platform call picker.

All platform implementations will show a picker dialog, on iOS it is a OS level dialog that Apple provides, on Android and Windows it is a custom dialog that mimics the same interface. They will both show the nearby Elcometer gauges and allow the user to select one. On selection of the gauge if it is not paired a pairing process will begin. Finally the gauge will be connected.

To determine which gauges are currently connected the property *Gauges* in the *GaugeService* can be monitored. It is an *ObservableCollection* so the *CollectionChanged* event can be subscribed to.

```
ElcometerCore.Instance.GaugeService.Gauges.CollectionChanged += Gauges_CollectionChanged;
```

The item in the *Gauges* collection is of type *IGauge* and holds information about the connected gauge and also offers methods that allow batches to be downloaded from the gauge.

Download Live Measurements

Once a gauge has been connected its live measurements can be captured. This can be done by registering for live measurements in the *MessagingService*, e.g.

```
ElcometerCore.Instance.MessagingService.Subscribe<ILiveReadingMessageParams>(this,
ElcometerCoreMessages.LiveReadingMessage, OnLiveReading);
```

The *ElcometerCoreMessages.LiveReadingMessage* is the message Id that registers for live measurements.

The OnLiveReading above is a method that will be called by the *MessagingService* when a live reading is received. It will also be called in the main UI thread context.

```
void OnLiveReading(object sender, ILiveReadingMessageParams args)
```

The arguments supplied by the callback are the sender which is IGauge object of the connected gauge that measured the reading and an *ILiveReadingMessageParams* interface that allows the reading values to be retrieved.

To retrieve the values first create an empty *Batch* class of the type required. This can be done using the *CreateEmptyBatch* method in the *ILiveReadingMessageParams object, e.g.*

```
var dummyBatch = args.CreateEmtpyBatch(ElcometerCore.Instance.BatchService, "");
```

The live reading columns can then be retrieved by passing the newly created batch to the *ILiveReadingMessageParams GetReadings* method, e.g.

```
var readings = args.GetReadings(dummyBatch);
```

GetReadings returns a collection of *LiveReadingItem* objects which allow the formatted Value to retrieved either as a string (*Value* property), a string postfixed with the unit type (*ValueEx* property) or the numeric value (*NumericValue* property), e.g.

```
string readingString = "";

foreach (var readingColumn in readings)
{
        readingString += readingColumn.Value + ", ";
}
```

The above would generate a single string of all readings that make up an individual measurement.

NOTE that many gauges that take a single measurement may return 3 reading columns per measurement. This is due to the scan mode feature that many of these gauges support, where they can return the mean, min, max of a scanned area as one measurement. When in normal mode they will still return 3 readings, but the min, max *Value* property will be '-' and the *NumericValue* will be NaN. Also note while a scan is taking place live measurements will be sent as well, but in this case the *IsPreview* property of the *ILiveReadingMessageParams* object will be true.

Downloading Batches

Batches are collections of measurements stored on a gauge, these can be downloaded by the SDK. The first step is to put the gauge into a mode where batches can be retrieved, this can be achieved by calling the *StartBatching* method of the *IGauge* object, this can take a few seconds so it is advisable if you don't want to block your user interface to run this in a Task, e.g.

```
await Task.Run(() => Gauge.StartBatching());
```

Once complete the *GetBatches* method that retrieves the list of batches can be called, again this can take a while to complete depending on the gauge type and the number of batches being retrieved.

```
var batches = await Task.Run(() => Gauge.GetBatches());
```

The list contains *IGaugeBatch* objects.

To download specific batches the *DownloadBatchesTo IGauge* method along with *BatchService* class can be used.

First the *BatchService* list of batches should be cleared, e.g.

```
ElcometerCore.Instance.BatchService.Batches.Clear();
```

Then the *DownloadBatchesTo* method can be called passing it a list of *IGaugeBatch* objects we want to download and the *BatchService* class, e.g.

```
await Task.Run(() => Gauge.DownloadBatchesTo(selectedBatches, ElcometerCore.Instance.BatchService));
```

Above the *selectedBatches* variable is a list of *IGaugeBatch* objects you wish to download selected from the list retrieved with the *GetBatches* method, again this can take a while to download.

Once completed the *BatchService Batches* property will contain the downloaded batch data.

You can retrieve the measurements stored in a downloaded batch via the *GetReadings* method, e.g.

```
readings = "";

foreach (var batch in ElcometerCore.Instance.BatchService.Batches)
{
        foreach (var reading in batch.GetReadings())
        {
                for (int i = 0; i < batch.ColumnCount; i++)
                {
                        readings += reading.ToDisplayString(i) + ", ";
                }
                readings += "\n";
        }
}
```

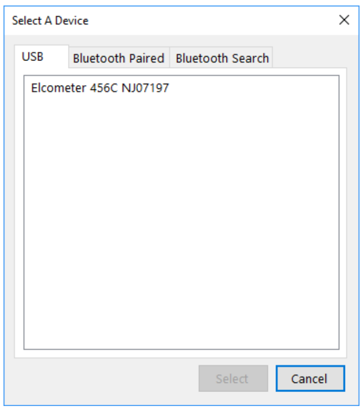
The above would display all the readings in all download batches.

Once all batches are downloaded the gauge can be returned to live measurement mode by sending the *EndBatching* method, e.g.

```
Task.Run(() => Gauge.EndBatching());
```

Windows Desktop (Win32)

The Windows desktop has two methods of connecting to gauges, as with the mobile platforms a picker dialog can be used to select a USB, paired Bluetooth or nearby Bluetooth gauge.



Elcometer 456C NJ07197

This can be shown calling the *ShowPicker* method in the *ConnectionService* class. This picker will find both Classic and LE Bluetooth devices.

Once a gauge is selected it is paired (if required) and then a connection is established.

Another option is to enumerate and connect directly to a gauge without any user interaction. The Windows Desktop SDK offers 3 methods to enumerate gauges.

```
public List<BluetoothDeviceInfo> GetNearbyBluetoothDevices();
public List<BluetoothDeviceInfo> GetPairedBluetoothDevices();
public List<USBDeviceInfo> GetUSBDevices();
```

All 3 methods are in the *ConnectionService* class. They all provide a list of devices that can be passed to *GaugeService.Connect* to initiate a connection.

```
public IGauge Connect(IDeviceInfo deviceInfo);
```

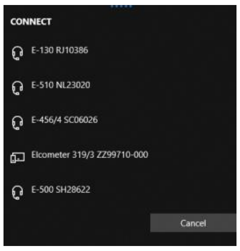
A connected gauge can then be subsequent disconnected using the GaugeService.Disconnect method.

```
public void Disconnect(IGauge gauge);
```

Windows UWP

The Windows UWP currently only supports connecting via Bluetooth to gauges.

The standard UWP device picker is wrapped via the *ShowPickerAsync* method in the *ConnectionService* class.



CONNECT

E-130 RJ10386

E-510 NL23020

E-456/4 SC06026

Elcometer 319/3 ZZ99710-000

E-500 SH28622

Once a gauge is selected it is paired using the standard Windows process (if required) and then a connection is established.

Another option is to enumerate and connect directly to a gauge without any user interaction. The Windows UWP SDK offers 2 methods to enumerate gauges.

```
public List<BluetoothDeviceInfo> GetNearbyBluetoothDevices();
public List<BluetoothDeviceInfo> GetPairedBluetoothDevices();
```

All 2 methods are in the *ConnectionService* class. They all provide a list of devices that can be passed to *GaugeService.Connect* to initiate a connection.

```
public IGauge Connect(IDeviceInfo deviceInfo);
```

A connected gauge can then be subsequent disconnected using the GaugeService.Disconnect method.

```
public void Disconnect(IGauge gauge);
```

Linux

If you wish to open the examples on the Linux distributions then you will need the latest version of MonoDevelop (https://www.monodevelop.com/download/).

Due to the lack of any standard way to enumerate USB and Bluetooth devices on the various Linux distributions the Linux support is limited to manually connecting to a specific Elcometer Bluetooth device.

The type of gauge and device path name needs to be known before a connection can be attempted.

Typically when a USB device is connected onto a PC running Linux you will get a new device appearing in /dev (e.g. /dev/ttyACM0).

e.g. If this gauge was a 456 you would initiate the connection like the following.

```
var device = new USBDeviceInfo("Elcometer-456/4", "/dev/ttyACM0");

// initialise the connection
var gauge = ElcometerCore.Instance.GaugeService.Connect(device);

// download batches, live readings, etc
```

See the Elcometer.Demo.Linux project for a more information.

***NOTE* a typical issue is insufficient privileges to read/write to a device. You can temporarily change the access permissions by doing something like the following (if you device was on /dev/ttyACM0):**

```
sudo chmod 666 /dev/ttyACM0
```

The USB production descriptions (used in the `USBDeviceInfo` constructor) of the supported gauges are as follows:

Gauge	Description
Elcometer 130	ELCOMETER-130/2
Elcometer 319	Elcometer 319
Elcometer 480	ELCOMETER-480
Elcometer 500	ELCOMETER-500
Elcometer 510	ELCOMETER-510
Elcometer 304	ELCOMETER-304
Elcometer 307	ELCOMETER-307
Elcometer MTG	ELCOMETER-MTG
Elcometer PTG	ELCOMETER-PTG
Elcometer 224C	Elcometer-224/2
Elcometer 311C	Elcometer-311/2
Elcometer 415C	Elcometer-415/2
Elcometer 456C	Elcometer-456/4
Elcometer 456TV	A456