Advanced Bluetooth Low Energy Development

Adrian Eggenberger
Senior Software Developer

Arendi AG
Eichtalstrasse 55
CH-8634 Hombrechtikon
www.arendi.ch

arendi
We are your solution.

Bluetooth SIG is increasing the scope of possible applications for Bluetooth Low Energy day by day. Many companies worldwide have already made their first project experience with Bluetooth Low Energy.

The range goes from prototypes to market ready products. It's the right time to lift your Bluetooth development to a higher level.

Using the right tools and skills can significantly improve the time to market, the robustness, the security and the power consumption. This paper will highlight three possible aspects of a development cycle and will give the reader ideas for improvement in their own projects.

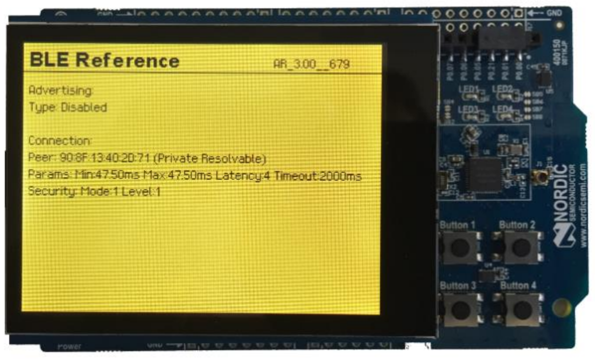
1. Introduction

"We have a good product, but we need connectivity with the smart device!" This sentence is often the beginning of new Bluetooth Low Energy (BLE) projects. Many companies started with BLE development within the last years. This article points out some potential for improvements for upcoming Bluetooth Low Energy projects. We focus on three topics:

- Device-Simulation for enhanced debugging and availability
- Real-Time Analysis to reduce power consumption
- System-Tests to improve stability and robustness

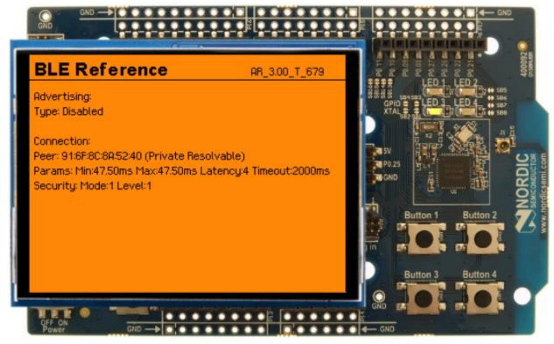
The examples and proposed ideas in this article are all based on the "Arendi BLE firmware platform". It's running on a nRF52 chip from Nordic Semiconductors with an ARM Cortex-M4. For demonstration purpose an Arduino LCD panel is attached to the nRF52 Development Kit. The system runs on our own cooperative scheduler.

Development improvement proposals in this article can also be applied to most other systems.



```
BLE Reference                AR_3.00__679
Advertising:
Type: Disabled

Connection:
Peer: 90:8F:13:40:20:71 (Private Resolvable)
Params: Min:47.50ms Max:47.50ms Latency:4 Timeout:2000ms
Security: Mode:1 Level:1
```



```
BLE Reference                AR_3.00_T_679
Advertising:
Type: Disabled

Connection:
Peer: 91:6F:8C:8A:52:40 (Private Resolvable)
Params: Min:47.50ms Max:47.50ms Latency:4 Timeout:2000ms
Security: Mode:1 Level:1
```

Figure 1 - Real Device (Left Side) versus Simulated Device (Right Side)

2. Device Simulation

In a project years ago our company faced some serious problems with hardware availability. The first prototype boards were late and the firmware team was limited to development without any hardware. This was when the device simulation concept was born. The team started to write an environment for desktop computers that emulates the behavior of the real hardware. Over time the simulated environment was adapted piece by piece to the needs in the current projects. Today we have a simulation that behaves nearly as equal as the real hardware (Figure 1).

On the left hand we have a photo of the real device. On the right hand a screen shot of the simulated version of the same device built with Microsoft Visual Studio 2015 and running on a x86 computer as WPF-application.

Concept

In a typical embedded device most of the software parts are not directly accessing the hardware. It's basically good practice to keep the platform specific stuff in small layers (e.g. Hardware Abstraction Layers).

Figure 2 shows the difference in the architecture between the real device and the simulated one. Most of the files running on an embedded system can run in a simulated environment without any changes. Only a small layer at the bottom with hardware access needs to be replaced with simulation specific files. These simulation modules emulate the real hardware as real as needed. In our example from above the hardware simulation is realized with a UI for visualization and is quite complex. The required simulation complexity may vary from project to project. In some cases, a simple console frontend may be sufficient.

What about BLE?

A simulated BLE device would never live up to the real world, to real challenges of a wireless technology. Therefore, we don't simulate the BLE part but route it to an external BLE device. So the BLE part is not really simulated, it's just exported to an external BLE device.

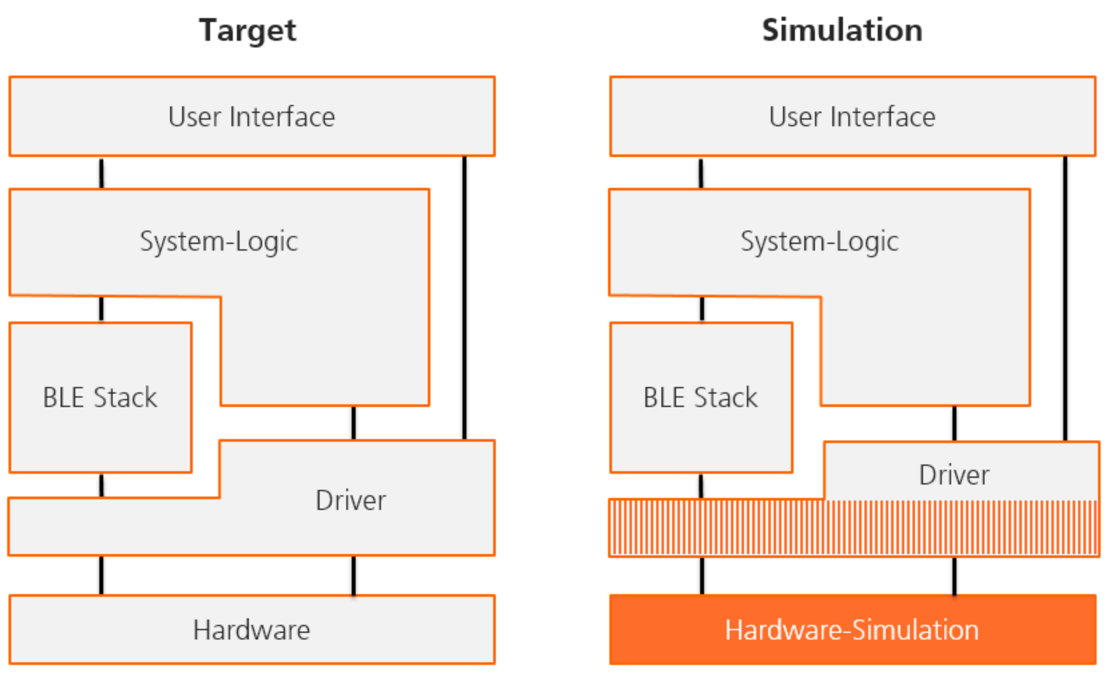


Figure 2 - Target and Simulation Architecture

The calls to the BLE hardware are packed into an UART protocol and sent to a connected BLE dongle. This dongle is programmed with a firmware that decodes the protocol and executes the calls on the real BLE chip. The responses and spontaneous events are sent back over UART to the simulation (Figure 3).

Enhanced debugger

The debugger of target development environments is often limited. Desktop development environments as we are using in our simulation, allow a deeper view in the system as the target environments.

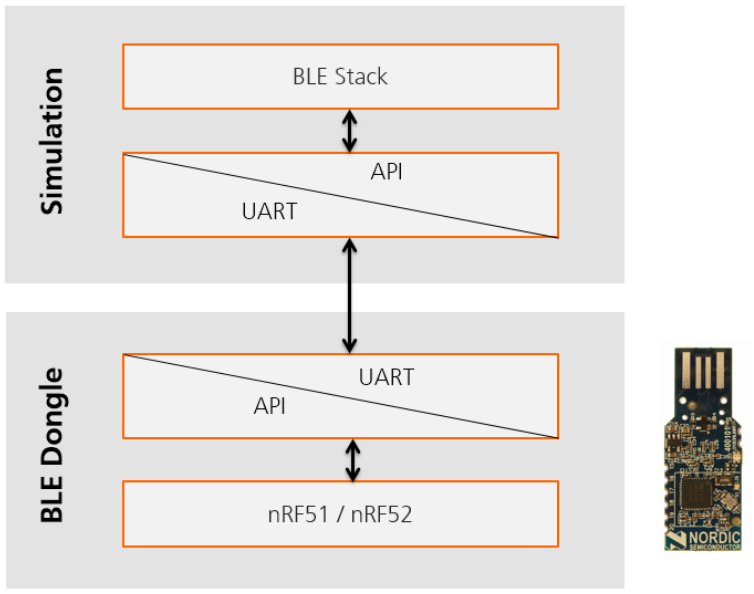


Figure 3 - BLE Support in simulated devices

How to implement BLE for a simulated device highly depends on the selected BLE solution. In some cases, the stack runs completely in hardware and can't be simulated. That would require a different solution as when the stack is available in source code and can run in the simulation.

Advantages

Availability

The availability of the hardware becomes less important. If the hardware is not yet working or only available in limited numbers or you don't have access to it at your alternative working place you can always work with the simulated device.

Hardware Failures

If your firmware needs to handle very rare hardware conditions, it could be hard to trigger and test the implementation in these cases. A simulated device gives you the option to force the failed situation and test how your system reacts.

System Knowledge

Getting diagnostic information of a target system may be hard. Limited interfaces and bandwidth reduces the possibilities for data output. The simulation runs on a fast system and is not limited in debug outputs nor in data storage.

3. Real-Time Analysis

A lot of Bluetooth Low Energy devices are powered by a battery and expected to be running 24/7 for many years. Using BLE is a good start as the technology gives you all the options you need for long battery life. Most people setup their communication correctly but waste power in the data handling. Real-Time Analysis is a general topic in embedded software development but it's absolute important in battery powered communication devices.

These questions are often difficult to answer:

How many times was the interrupt service routine called?

Which processes are active in my system?

How does the OS behave?

How long does my data handling take?

Why is Real-Time Analysis important?

In battery driven devices any optimized real time behavior may increase battery life by months or even years. Figure 4 shows a typical example for a BLE device.

The device is connected and gets some data every second. To maintain the active BLE link only a few micro amperes are required, but the handling of the incoming data requires some CPU power. With the shown power profile, the device would run approximately 2.4 years on a CR2032 coin cell. If you are able to reduce the data handling time by 20% (=200us) the battery life would increase up to 3 years.

Classic methods

There are a few well known methods for system analysis. They have all their advantages, but also some disadvantages.

Method	Disadvantage
Debug output to a terminal	Hugh influence on system timing Requires an unused output interface (e.g. UART)
Time measurements by GPIO	Only dedicated measurements Requires free and accessible GPIO pins
Debugger Breakpoints	Only dedicated measurements Often not usable in combination with RF protocols as BLE

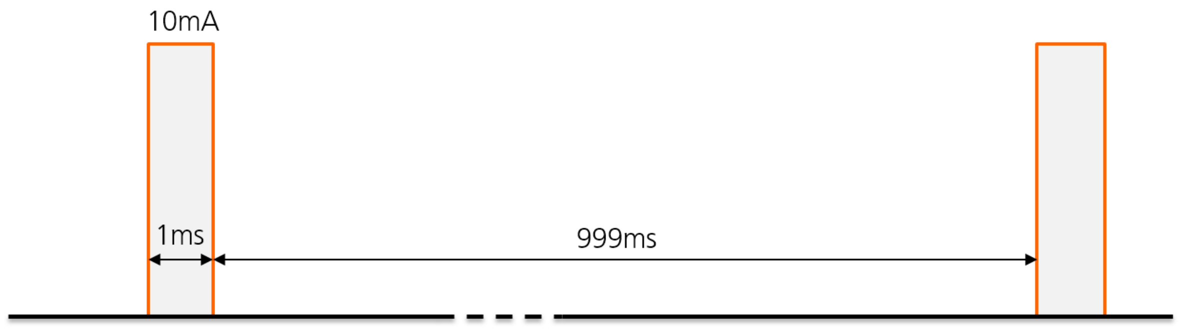


Figure 4 - Typical current profile of a BLE device

Segger SystemView

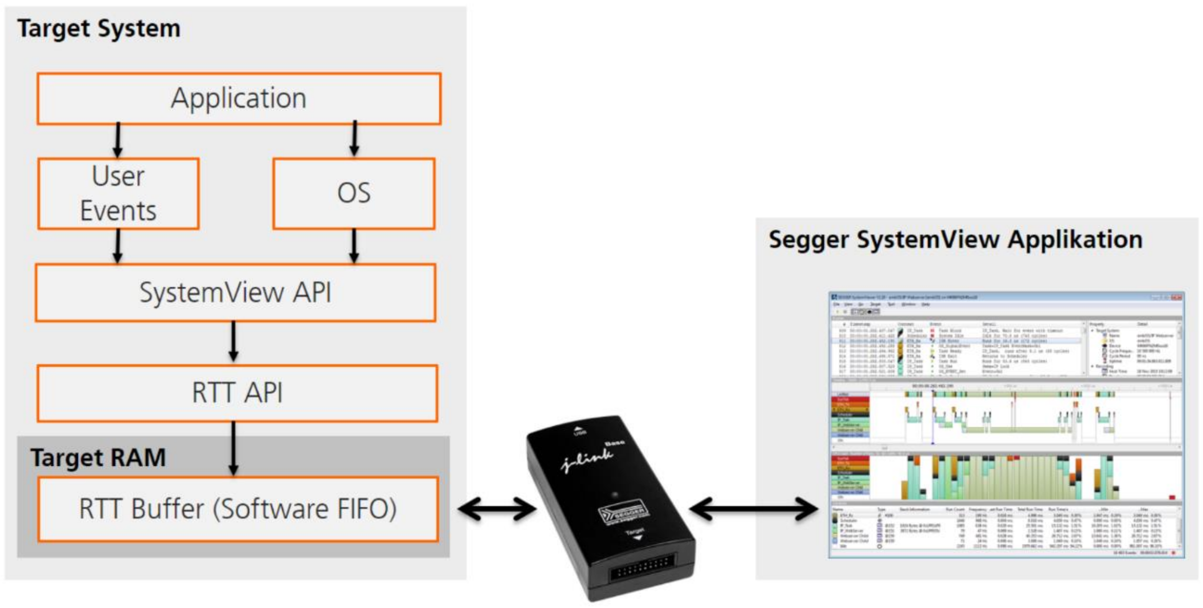


Figure 5 - SystemView concept

While the classic methods are still useful for various measurements we've added an additional method to improve our tools for system analysis. Segger SystemView is a free tool provided by *SEGGER Microcontroller* that works with J-Link debuggers from the same company. SystemView may be used with ARM Cortex and Renesas RX series.

Concept

The concept of SystemView is simple. In the normal firmware calls to the SystemView API can be added for interesting events. These calls result in fast write operations to a dedicated circular buffer in the target with minimal impact to the real time behavior and code size. The J-Link debugger will continuously ready the data from the circular buffer and send the events to the SystemView application for visualization.

Analyze your system

The provided application for SystemView visualization allows you to see the logged event. By adding the corresponding events to your OS or using an OS already supported by SystemView, the task switching and event processing can be visualized as well.

The snapshot in Figure 6 shows a RTC interrupt that triggers an event sent to the PWR and BLE process.

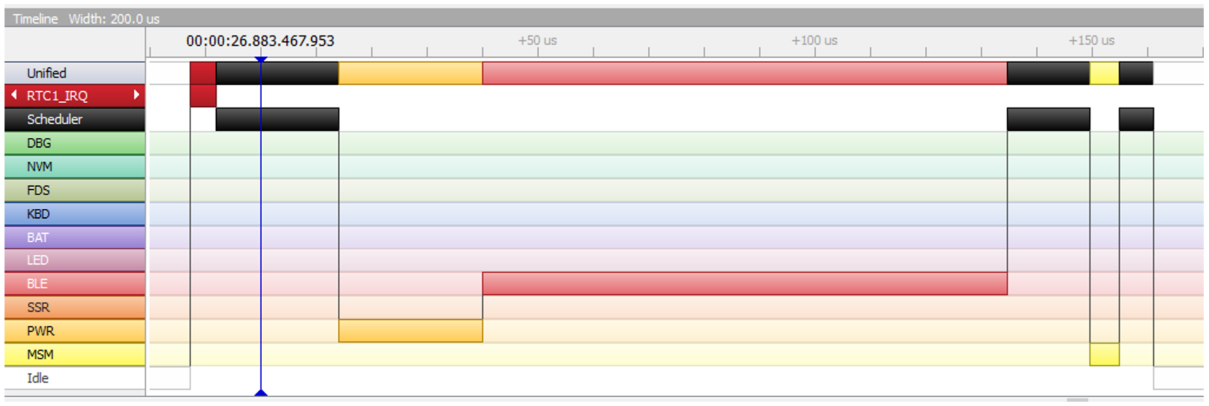


Figure 6 - SystemView visualisation

4. Automatic Testing with BLE

Everyone knows that it would be a good thing to run automatic tests on a regular basis. In practice a lot of companies release products without an implementation of automatic testing. There exist multiple types of tests that can be done automatically:

Unit testing

A unit test (also known as component test) verifies the functionality of a single module or component.

Integration testing

The integration test verifies that modules in the device work together as expected. They focus on the interfaces between modules.

System testing

The system test is a test of a complete integrated system. Typically, the system requirements and system interfaces are covered by this test.

In this paper we focus on the testing of Bluetooth Low Energy. The BLE interface is a typical external system interface of a device. Therefore, BLE should be covered by a system test.

System-Test setup

System tests are typically executed by a test framework. Which framework is the best for your environment depends on various factors:

- Platform of the test system
- Test script language
- Support for existing software components
- Knowledge of already used frameworks

In our case we have chosen NUnit because it fits well with our other C# components and works perfectly on our development computers and servers.

In a traditional system test setup, you have a test script that has a few options for information exchange with the device under test (DUT). There could be a communication protocol or a direct hardware access to a modified DUT (e.g. pressing buttons, get led state or even more).

To add support for BLE we add a "nRF51 Dongle" from *Nordic Semiconductors* to the test system. The dongle can be accessed by a serial protocol (see also the section *What about BLE?* at page 2). The dongle will act as the counterpart of the DUT. If your DUT is implementing the Peripheral Role for example the dongle will implement the Central Role that connects to the peripheral.

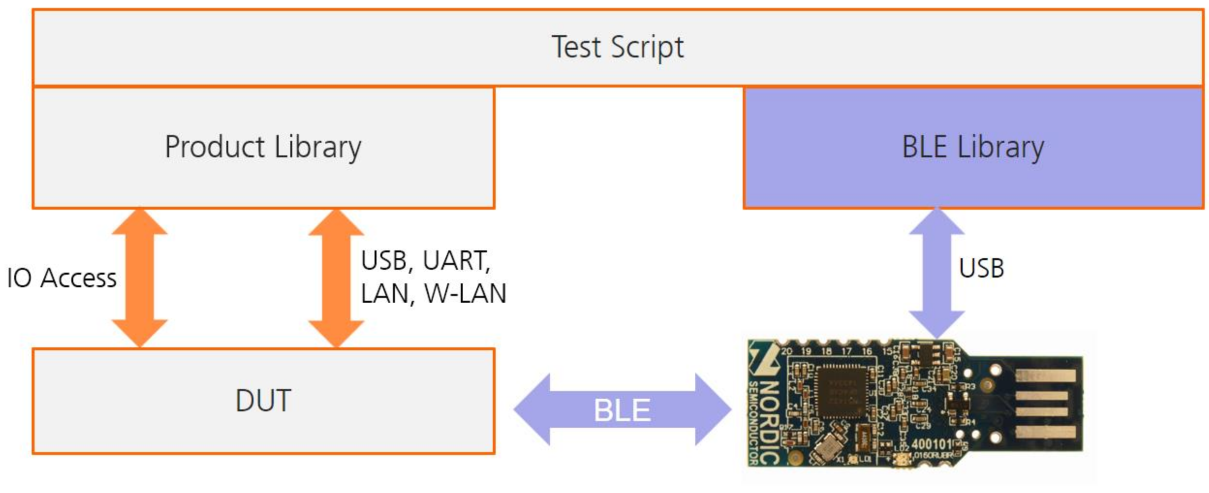


Figure 7 - System-Test setup for BLE tests

Writing tests

The typical test framework allows to implement code for test setup and test teardown. The setup function should be used to setup communication with the DUT and the BLE dongle. In the teardown function the communication with DUT and BLE dongle can be terminated. For best reproducibility it's recommended to reset the DUT before any test to ensure equal conditions for every test run.

In Figure 7 two library components are placed between test script and interfaces. The libraries provide simple functions for the scripts to access the corresponding interfaces. This will help to keep the test scripts as short as possible and prevent redundant code in multiple test scripts.

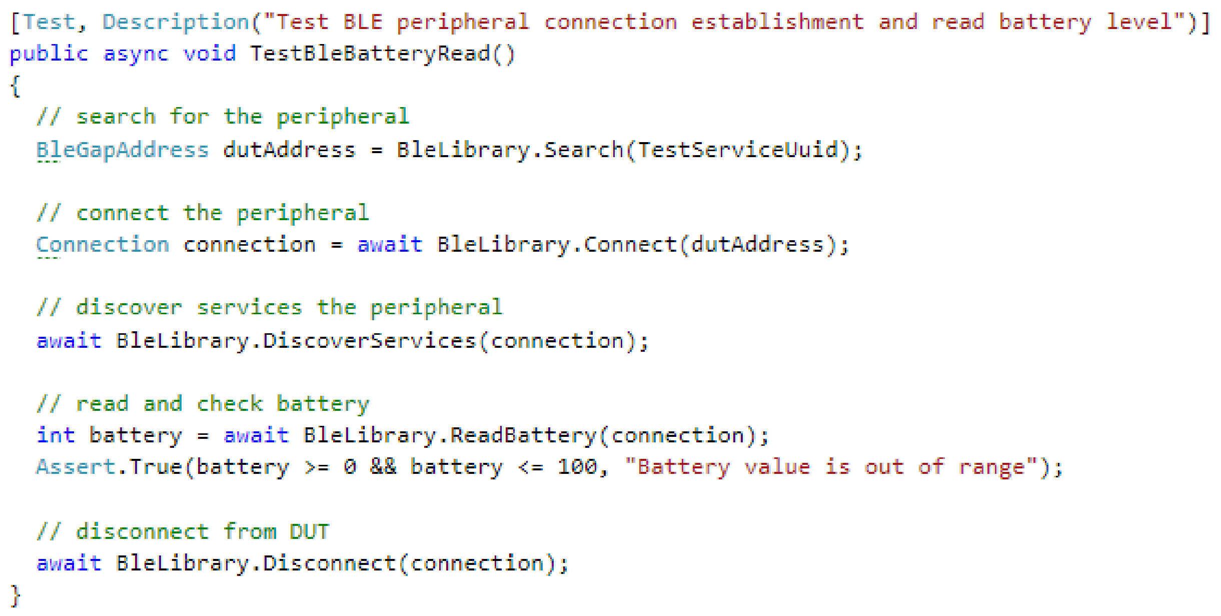
The script example (Figure 8) shows a test that tries to discover the DUT by scanning for it over BLE. Once detected it is connected and all services discovered. The battery value is read by accessing the battery service characteristic and verified if it is within a valid range. To complete the test, the device is disconnected. In the test script there is no error handling (e.g. if the DUT couldn't be detected) because any unexpected behavior will terminate the test with an error caused by a thrown uncatched exception.

Scripts as the discussed one are a good start but you may have noticed, that we don't communicate with DUT except over BLE. This is only possible if the DUT is always in the right state and advertises as connectable device. In a more complex scenario it may be required to force the DUT into a certain state by triggering a button on the DUT or sending some commands over the communication interface.

Running test

Once you have written a bunch of tests it's time to setup the automatic testing. It is recommended to execute the tests by a build management system. If you don't want to setup a build system a regular executed script may work as well. The following steps should be executed at least once per night to create and test the nightly build for your device.

1) Checkout a clean copy of the projects
2) Build the nightly firmware build
3) Build the system test project
4) Program the nightly build on the DUT
5) Run your system tests
6) Notify one or multiple team members if some of the tests fail



```
[Test, Description("Test BLE peripheral connection establishment and read battery level")]
public async void TestBleBatteryRead()
{
  // search for the peripheral
  BleGapAddress dutAddress = BleLibrary.Search(TestServiceUuid);

  // connect the peripheral
  Connection connection = await BleLibrary.Connect(dutAddress);

  // discover services the peripheral
  await BleLibrary.DiscoverServices(connection);

  // read and check battery
  int battery = await BleLibrary.ReadBattery(connection);
  Assert.True(battery >= 0 && battery <= 100, "Battery value is out of range");

  // disconnect from DUT
  await BleLibrary.Disconnect(connection);
}
```

Figure 8 - Script to read battery voltage over BLE service

5. Conclusion

The proposed methods may give you some options how you could improve your development in different project stages. Caused by various development environments and hardware setups the proposed methods must be adapted to the actual project. Implementing these methods will be time consuming for the first project but you'll get a lot better product quality and performance at the end. Upcoming projects based on the same environment will benefit automatically from this investment with minimal cost.

Adrian Eggenberger, October 2016