AN980: *BLUETOOTH* SMART SDK

Developing your 1st *Bluetooth* Smart Application

Tuesday, 02 April 2019

Version 2.1

SILICON LABS

VERSION HISTORY

Version	Comment
1.0	First version
1.1	Project and Hardware configuration added
1.2	BGScript and firmware update instructions added
1.3	Better screen captures and BLEGUI example added
1.4	*Bluetooth* LE description updated
1.6	Minor updates
1.7	Updated compile and installation instructions
1.8	Chapter 3 updated
1.9	Chapter 4 updated
2.0	Minor changes
2.1	Updated notes about sleep oscillator setting, which should be configured in the hardware configuration file

TABLE OF CONTENTS

1 Introduction

This application note discusses how to start developing Bluetooth Smart applications using Bluegiga *Bluetooth* Smart modules and BLED112 Bluetooth Smart USB dongle. The application note contains a practical example of how to build Bluetooth Smart GATT based services with the profile toolkit, how to make a standalone sensor device using BGScript programming language.

2 What is *Bluetooth* low energy Technology?

Bluetooth low energy (*Bluetooth* 4.0) is a new, open standard developed by the *Bluetooth* SIG. It's targeted to address the needs of new modern wireless applications such as ultra-low power consumption, fast connection times, reliability and security. *Bluetooth* low energy consumes 10-20 times less power and is able to transmit data 50 times quicker than classical *Bluetooth* solutions.

Link: How Bluetooth low energy technology works?

Bluetooth low energy is designed for new emerging applications and markets, but it still embraces the very same benefits we already know from the classical, well established *Bluetooth* technology:

- **Robustness and reliability** - The adaptive frequency hopping technology used by *Bluetooth* low energy allows the device to quickly hop within a wide frequency band, not just to reduce interference but also to identify crowded frequencies and avoid them. On addition to broadcasting *Bluetooth* low energy also provides a reliable, connection oriented way of transmitting data.
- **Security** - Data privacy and integrity is always a concern is wireless, mission critical applications. Therefore *Bluetooth* low energy technology is designed to incorporate high level of security including authentication, authorization, encryption and man-in-the-middle protection.
- **Interoperability** - *Bluetooth* low energy technology is an open standard maintained and developed by the *Bluetooth* SIG. Strong qualification and interoperability testing processes are included in the development of technology so that wireless device manufacturers can enjoy the benefit of many solution providers and consumers can feel confident that equipment will communicate with other devices regardless of manufacturer.
- **Global availability** - Based on the open, license free 2.4GHz frequency band, *Bluetooth* low energy technology can be used in world wide applications.

There are two types of *Bluetooth* 4.0 devices:

- ***Bluetooth* 4.0 single-mode** devices that only support *Bluetooth* low energy and are optimized for low-power, low-cost and small size solutions.
- ***Bluetooth* 4.0 dual-mode** devices that support *Bluetooth* low energy and classical *Bluetooth* technologies and are interoperable with all the previously *Bluetooth* specification versions.

Key features of *Bluetooth* low energy wireless technology include:

- Ultra-low peak, average and idle mode power consumption
- Ability to run for years on standard, coin-cell batteries
- Low cost
- Multi-vendor interoperability
- Enhanced range

Bluetooth low energy is also meant for markets and applications, such as:

- Automotive
- Consumer electronics
- Smart energy
- Entertainment
- Home automation
- Security & proximity
- Sports & fitness

3 Typical *Bluetooth* 4.0 Application Architecture

3.1 Overview

Bluetooth low energy applications typically have the following architecture:

- **Server**

  Service is the device that provides the information, so these are typically the sensor devices, like thermometers or heart rate sensors. The server exposes implements services and the services expose the data in characteristics.

- **Client**

  Client is the device that collects the information for one or more sensors and typically either displays it to the user or passes it forward. The client devices typically do not implement any service, but just collect the information from the service provided by the server devices. Clients are typically devices like mobile phones, tablets and PCs.

The figure below shows the relationship of these two roles.

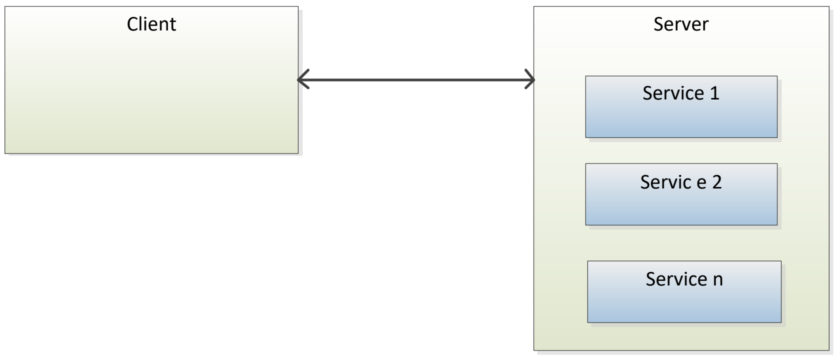


Figure 1: *Bluetooth* low energy device roles

3.2 What is a Profile?

Profiles are used to describe devices and the data they expose and also how these devices behave. The data is described by using services, which are explained later and a profile may implement single or multiple services depending on the profile specification. For example a Heart Rate Service specification mandates that the following services need to be implemented:

- Heart Rate Service
- Device Information Service

Profile specifications might also define other requirements such as security, advertisement intervals and connection parameters.

The purpose of profile specifications is to allow device and software vendors to build standardized interoperable devices and software. Standardized profiles have globally unique 16-bit UUID, so they can easily identify.

Profiles are defined in profiles specifications, which are available at:

https://www.bluetooth.com/specifications/

3.3 What Is a Service?

Services such as a Heart Rate service describes what kind of data a device exposes, how the data can be accessed and what the security requirements for that data are. The data is described using characteristics and a service may contain single or multiple characteristics and some characteristics might be optional where as some are mandatory.

Two types of services exist:

- **Primary Service**

  A primary service is a service that exposes primary usable functionality of this device. A primary service can be included by another service.

- **Secondary Service**

  A secondary service is a service that is subservient to another secondary service or primary service. A secondary service is only relevant in the context of another service.

Just like the profiles also the services are defined in service specifications and the Bluetooth SIG standardized services are available at:

https://www.bluetooth.com/specifications/gatt/services

Every service standardized by the Bluetooth SIG has a globally unique 16-bit UUID so just like the profiles also the services can be easily identified.

However not every use case can be fulfilled by the standardized service and therefore the *Bluetooth* Smart specification enables device vendors to make proprietary service. The proprietary services are described just as the standardized services, but 128-bit UUIDs need to be used instead of use 16-bit UUIDs reserved for the standard services.

3.4 What is a Characteristic?

Characteristics are used to expose the actual data. Characteristic is a value, with a known type (UINT8, UINT16, UTF-8 etc.), a known presentation format. Just like profiles and services also characteristics have unique UUID so they can be easily identified and the standardized characteristics use 16-bit UUIDs and vendor specific characteristics use 128-bit UUIDs.

Characteristics consist of:

- **Characteristic Declaration** describing the properties of characteristic value such as:
  - characteristic (UUID)
  - Access control *(*read, write, indicate etc.)
  - *Characteristic value* handle (unique handle within a single device)
- **Characteristic Value** containing the value of a characteristic (for example temperature reading).

- **Characteristic Descriptor(s)** which provide additional information about the characteristic (characteristic user description, characteristic client configuration, vendor specific information etc.).

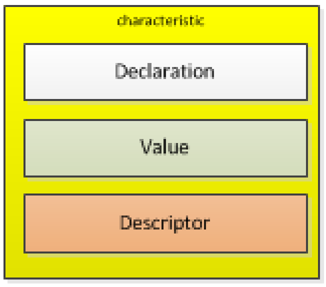


Figure 2: Characteristic structure

Standardized characteristics are defined in Characteristic Specification and the standardized characteristics are available at:

https://www.bluetooth.com/specifications/gatt/characteristics

3.5 Relationship Between Profiles, Services and Characteristics

The illustration below shows the relationship between profiles, services and characteristics.

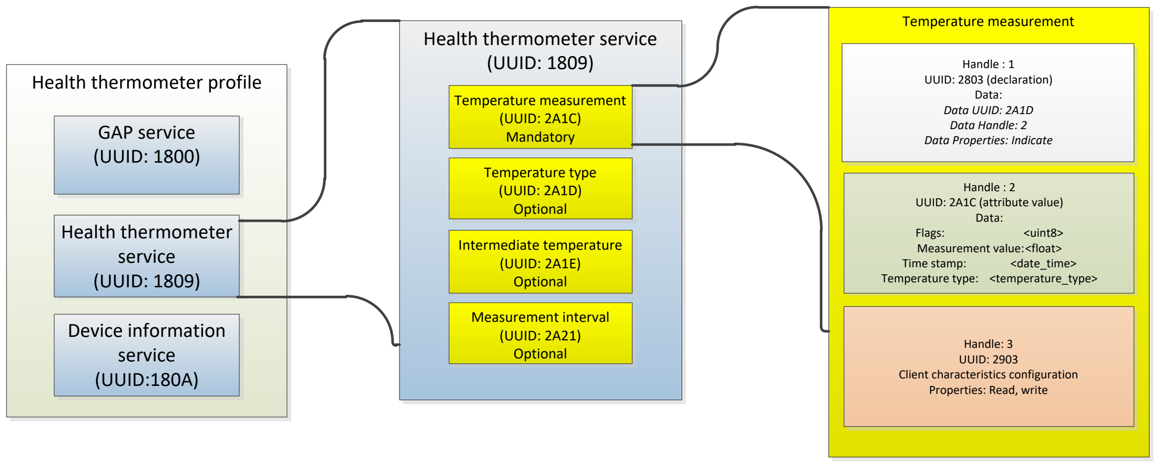

Health thermometer profile
GAP service (UUID: 1800)
Health thermometer service (UUID: 1809)
Device information service (UUID:180A)
Health thermometer service (UUID: 1809)
Temperature measurement (UUID: 2A1C) Mandatory
Temperature type (UUID: 2A1D) Optional
Intermediate temperature (UUID: 2A1E) Optional
Measurement interval (UUID: 2A21) Optional
Temperature measurement
Handle : 1
UUID: 2803 (declaration)
Data:
Data UUID: 2A1D
Data Handle: 2
Data Properties: Indicate
Handle : 2
UUID: 2A1C (attribute value)
Data:
Flags: <uint8>
Measurement value:<float>
Time stamp: <date_time>
Temperature type: <temperature_type>
Handle: 3
UUID: 2903
Client characteristics configuration
Properties: Read, write

Figure 3: Health thermometer profile

4 Introduction to the Bluegiga *Bluetooth* Smart Software

The Bluegiga *Bluetooth* Smart Software enables developers to quickly and easily develop *Bluetooth* Smart applications without in-depth knowledge of the *Bluetooth* Smart technology. The *Bluetooth* Smart Software consist of two parts:

- The *Bluetooth* Smart Stack
- The *Bluetooth* Smart Software Development Kit (SDK)

4.1 The *Bluetooth* Smart Stack

The *Bluetooth* Smart stack is a fully *Bluetooth* 4.0 single mode compatible software stack implementing slave and master modes, all the protocol layers such as L2CAP, Attribute Protocol (ATT), Generic Attribute Profile (GATT), Generic Access Profile (GAP) and security and connection management.

The *Bluetooth* Smart is meant for the Bluegiga *Bluetooth* Smart products such as BLE112, BLE113 and BLED112 and it runs on the embedded MCU used in these products so no host is needed.

4.2 The *Bluetooth* Smart SDK

The *Bluetooth* Smart SDK is a software development kit, which enables the device and software vendors to develop products on top of the Bluegiga's *Bluetooth* Smart hardware and software.

The *Bluetooth* Smart SDK supports multiple development models and the software developers can decide whether the application software runs on a separate host (a low power MCU) or whether they want to make fully standalone devices and execute their code on the MCU embedded in the Bluegiga *Bluetooth* Smart modules. The SDK also contains documentation, tools for compiling the firmware, installing it into the hardware and lot of example application speeding up the development process.

Fully standalone applications can be developed using a simple scripting language called BGScript^{TM}. Several profiles and examples are also offered as a part of the *Bluetooth* Smart Software in order to easily develop the *Bluetooth* Smart compatible end products.

Bluegiga's *Bluetooth Smart Software* provides a complete development framework for *Bluetooth* low energy application implementers.

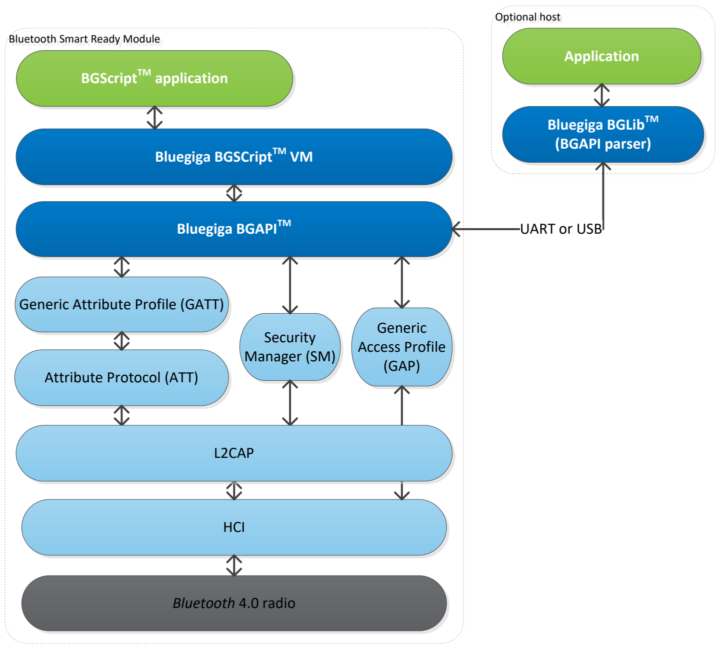


Figure 4: *Bluetooth Smart* Software

The *Bluetooth Smart* Software architecture is illustrated and it consists of the following components

- The *Bluetooth* Smart stack implementing the *Bluetooth* low energy protocol
- **BGAPI™** APIs that enable the software developers to interface to the *Bluetooth* Smart Stack
- **BGScript™** Virtual Machine (VM) and scripting language which enable application code to be developed and executed directly on the *Bluetooth* Smart hardware
- **BGLib™** lightweight host library which implements the BGAPI binary protocol and parser and is target for applications where separate host processor is used to interface to the *Bluetooth* Smart modules over UART or USB.
- **Profile Toolkit™** is a GATT based profile development tool that enables software developers quickly and easily to describe the *Bluetooth* Smart profiles, services and characteristics using simple XML templates

Each of these components are described in more detail in the following chapters.

4.3 The BGAPI Protocol

For applications where a separate host is used to implement the end user application, a transport protocol is needed between the host and the *Bluetooth* stack. The transport protocol is used to communicate with the *Bluetooth* stack as well to transmit and receive data packets. This protocol is called BGAPI and it's a lightweight binary based communication protocol designed specifically for ease of implementation within host devices with limited resources.

The BGAPI protocol is a simple command, response and event based protocol and it can be used over UART SPI (at the moment not supported by the *Bluetooth* Smart hardware) or USB interfaces.

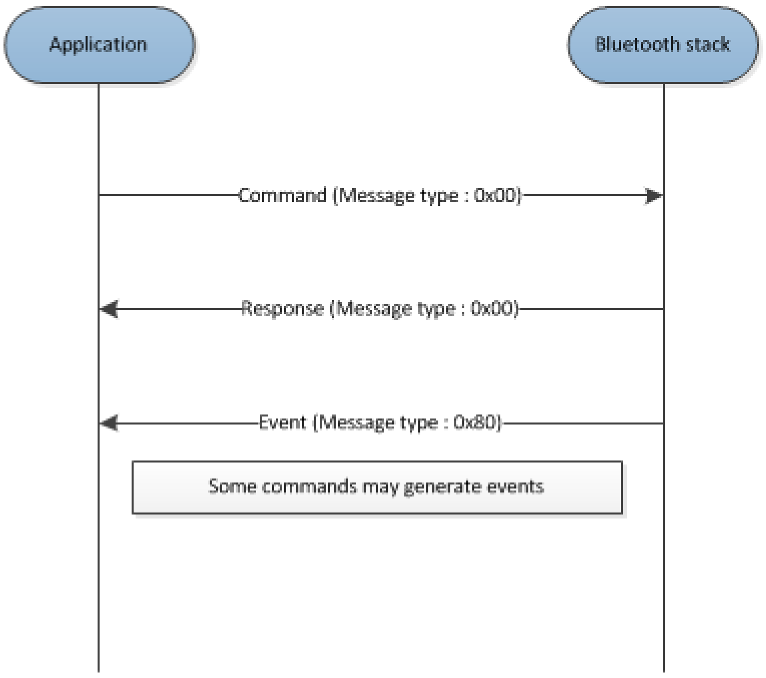

Application
Bluetooth stack
Command (Message type : 0x00)
Response (Message type : 0x00)
Event (Message type : 0x80)
Some commands may generate events

Figure 5: BGAPI protocol

The BGAPI provides access for example to the following layers in the *Bluetooth* Smart Stack:

- **Generic Access Profile** - GAP allows the management of discoverability and connetability modes and open connections
- **Security manager** - Provides access the *Bluetooth* low energy security functions
- **Attribute database** - An class to access the local attribute database
- **Attribute client** - Provides an interface to discover, read and write remote attributes
- **Connection** - Provides an interface to manage *Bluetooth* low energy connections
- **Hardware -** An interface to access the various hardware layers such as timers, ADC and other hardware interfaces
- **Persistent Store** - User to access the parameters of the radio hardware and read/write data to non-volatile memory
- **System** - Various system functions, such as querying the hardware status or reset it

4.4 The BGLib Host Library

For easy implementation of BGAPI protocol an ANSI C host library is available. The library is easily portable ANSI C code delivered within the *Bluetooth* Smart SDK. The purpose is to simplify the application development to various host environments.

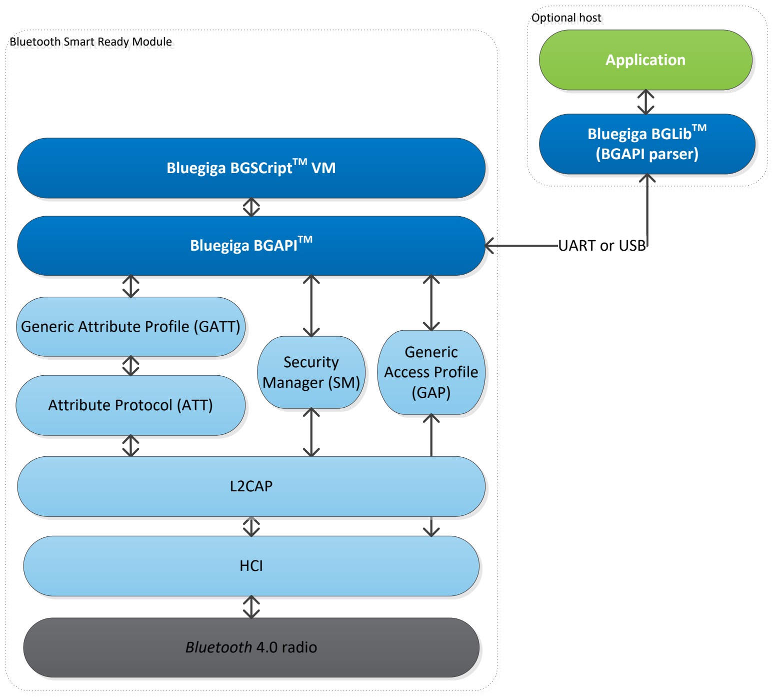


Figure 6: BGLib host library

4.5 BGScript^{TM} Scripting Language

The *Bluetooth* Smart SDK also allows the application developers to create fully standalone devices without a separate host MCU and run all the application code on the Bluegiga *Bluetooth* Smart Hardware. The *Bluetooth* Smart modules can run simple applications along the *Bluetooth* Smart stack and this provides a benefit when one needs to minimize the end product's size, cost and current consumption. For developing standalone *Bluetooth* Smart applications the SDK includes the Script VM, compiler and other BGScript development tools. BGScript provides access to the same software and hardware interfaces as the BGAPI protocol and the BGScript code can be developed and compiled with free-of-charge tools provided by Bluegiga.

Typical BGScript applications are only few tens to hundreds lines of code, so they are really quick and easy to develop and lots of readymade examples are provides with the SDK.

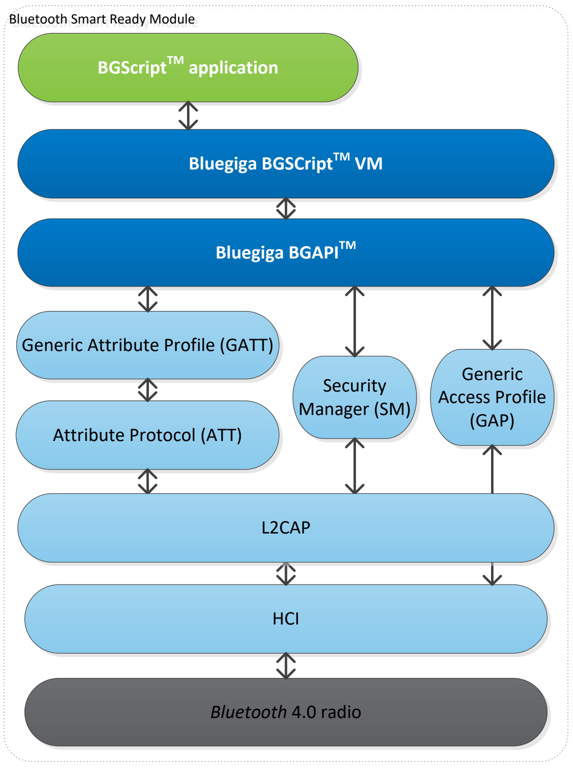


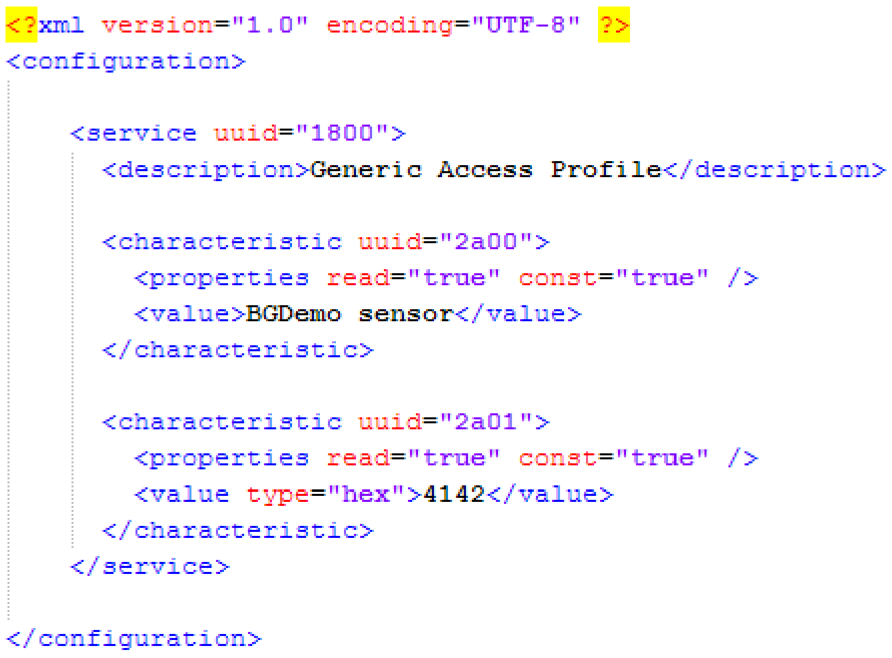
Figure 7: BGScript application model

BGScript code example:

```
# System Started
event system_boot(major, minor, patch, build, ll_version, protocol_version,hw)
           #Enable advertising mode
           call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
           #Enable bondable mode
           call sm_set_bondable_mode(1)
           #Start timer at 1 second interval (32768 = crystal frequency)
           call hardware_set_soft_timer(32768)
end
```

4.6 The Profile Toolkit

The *Bluetooth* Smart profile toolkit a simple set of tools, which can used to describe GATT based *Bluetooth* Smart services and characteristics. The profile toolkit consists of a simple XML based description language and templates, which can be used to describe the devices GATT database. The profile toolkit also contains a compiler, which converts the XML to binary format and generates API to access the characteristic values.



```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

    <service uuid="1800">
      <description>Generic Access Profile</description>

      <characteristic uuid="2a00">
        <properties read="true" const="true" />
        <value>BGDemo sensor</value>
      </characteristic>

      <characteristic uuid="2a01">
        <properties read="true" const="true" />
        <value type="hex">4142</value>
      </characteristic>
    </service>

</configuration>
```

Figure 8: A profile toolkit example of GAP service

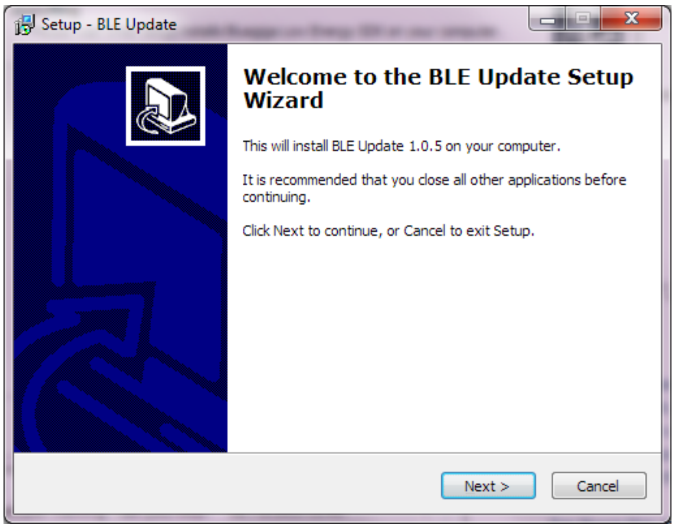
5 Implementation of “BGDemo” Sensor

In this chapter we discuss an actual implementation of a standalone *Bluetooth* Smart sensor called “BGDemo”. The implementation consists of following steps:

1. Installing the tools
2. Setting up the project
3. Defining hardware configuration
4. Building a GATT based service database with profile toolkit
5. Writing a simple BGScript that defines the sensors functionality
6. Compiling the GATT data base and BGScript into a binary firmware
7. Installing the firmware into BLE112 or BLED112 hardware
8. Testing it out

5.1 Installing the Tools

1. Download the latest install the Bluegiga *Bluetooth* Smart SDK from the Bluegiga web site
2. Run the executable
3. Follow the on-screen instructions and install the SDK to the desired directory
4. Perform a Full Installation (BLE SDK and TI tools)



Welcome to the BLE Update Setup Wizard

This will install BLE Update 1.0.5 on your computer.

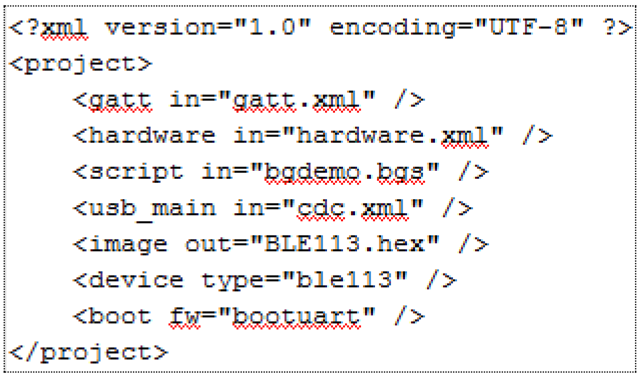
It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Figure 9: Installing Bluegiga *Bluetooth* Smart SDK

5.2 Creating a Project

The project is started by creating a project file. The file is a simple XML formatted document and defines all the other files the included in the project. An example of a complete project file is shown below:



```
<?xml version="1.0" encoding="UTF-8" ?>
<project>
    <gatt in="gatt.xml" />
    <hardware in="hardware.xml" />
    <script in="bgdemo.bgs" />
    <usb_main in="cdc.xml" />
    <image out="BLE113.hex" />
    <device type="ble113" />
    <boot fw="bootuart" />
</project>
```

Figure 10: Project file

- The project configuration is described within the <project> tags
- <gatt> tag defines the .XML file containing the GATT data base
- <hardware> tag defines the .XML file containing the hardware configuration
- <script> tag defines the .BGS file containing the BGScript code. If the project does not contain a BGScript code, this tag can be simply left out.
- <usb_main> tag defines the .XML file containing the USB descriptors description. If the project does not use USB interface, this tag can be simply left out.
- <image> tag defines the output .HEX file containing the firmware image
- <device type> tag defines if the project is meant for BLE112 or BLE113 hardware
- <boot fw> tag defines which interface is enabled for DFU firmware upgrades

The exact syntax and options of the project file can be found from the *BLE112 and BLE113 Configuration Guide* and the syntax is not fully described in this document.

NOTE:

For applications targeted for BLE112 module, the USB should be disabled as the USB interface will continually draw around 1mA of power.

WARNING:

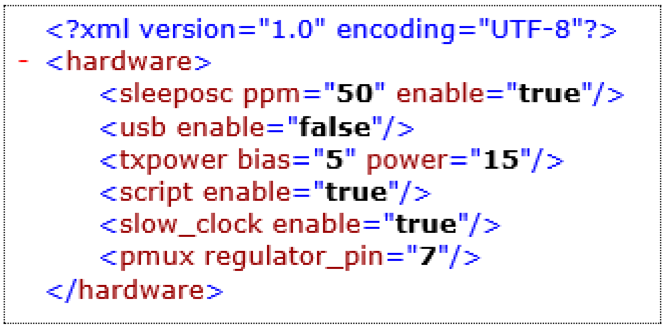
If the firmware is to be installed into the BLED112 USB dongle the USB CDC configuration MUST BE included in the project file. If this is not included in the project file and the compiled firmware is installed into the BLED112 USB dongle, the USB interface will be disabled and the dongle stops from working.

5.3 Hardware Configuration

Once the project is configured the next logical step is the hardware configuration of your Bluetooth Smart module. In this document we use the BLE113 *Bluetooth* Smart Module as a target platform.

If the default project template is used, the file where the hardware configuration remains is called **hardware.xml**.

An example of a hardware configuration used in BGDemo application is shown below.



```
  <?xml version="1.0" encoding="UTF-8"?>
- <hardware>
      <sleeposc ppm="50" enable="true"/>
      <usb enable="false"/>
      <txpower bias="5" power="15"/>
      <script enable="true"/>
      <slow_clock enable="true"/>
      <pmux regulator_pin="7"/>
  </hardware>
```

Figure 11: Hardware configuration for the BLE113 *Bluetooth* Smart Module

- The hardware configuration is described within the <hardware> tags
- <sleeposc> tag defines whether the sleep oscillator is enabled or not. The Sleep oscillator allows low power sleep modes to be used. In BLE112, BLE113, and BLE121LR this option MUST be configured to enable the external sleep oscillator, while in the BLED112 this option MUST be set to **false**, since the USB dongle does not contain the required external oscillator. Setting this option to **true** with device type set to "BLED112" will result in a compilation error in BGBuild. The PPM value defines the sleep oscillator accuracy. Increasing this value may improve connection stability in extreme temperature conditions, but also may have negative influence on current consumption.
- <usb> tag defines if USB is to be enabled or not. The BLE113 (unlike BLE112 or BLED112) does not have USB interface so we leave the setting to **false**.
- <txpower> tag defines the TX power level and the value 15 configures the maximum TX power level.
- <script enable> tag defines if BGScript VM and application are present. Since our example uses a BGscript application we set this value to **true.**
- <slow clock> tag enabled the slow the MCU clock when there is radio activity and reduces the peak power consumption. The option is enabled by setting the value to **true.**
- <pmux regulator_pin> configuration defines which GPIO pin is used to control an external DC/DC converter. An external DC/DC converter can be used to lower the peak power consumption during radio activity and the Bluetooth Smart software will automatically enable or disable the DC/DC based on the software status. The DKBLE112 and DKBLE113 development kits have the DC/DC converter, so this feature is enabled.

NOTE:

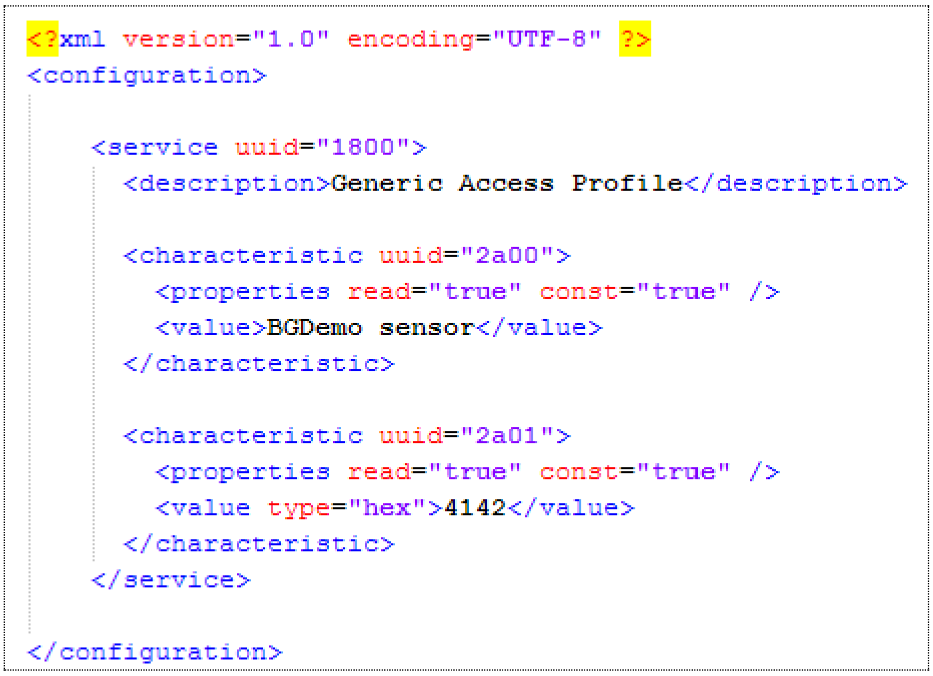
- Enabling the <slow clock> feature will corrupt high speed UART transmissions, so if UART is used in your application this feature MUST NOT be enabled.

5.4 Building a GATT Database with Profile Toolkit

This section discusses the implementation of a GATT database so the services and characteristics exposed by a device. The service database is created with the Profile Toolkit™ tools, which simply is are just XML based description language and templates.

5.4.1 Generic Access Profile Service

Every *Bluetooth* Smart device needs to implement a GAP service. The GAP service is very simple and consists of only two characteristics. An example implementation of GAP service is show below.



```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

    <service uuid="1800">
      <description>Generic Access Profile</description>

      <characteristic uuid="2a00">
        <properties read="true" const="true" />
        <value>BGDemo sensor</value>
      </characteristic>

      <characteristic uuid="2a01">
        <properties read="true" const="true" />
        <value type="hex">4142</value>
      </characteristic>
    </service>

</configuration>
```

Figure 12: GAP service

5.4.1.1 Service Description

A Bluetooth Smart service is described within the <service> tags. For every service you need to define a UUID as shown in the example above.

For the GAP service the globally unique 16-bit UUID is : **1800**

In the example above we also use optional <description> tag is used to identify the service name. This is optional tag and can be considered to be a comment in the XML file and is not used in the actual device.

The Bluetooth SIG standardized services and UUIDs are available at:

https://www.bluetooth.com/specifications/gatt/services

5.4.1.2 Characteristics Description

Characteristics belonging to a service are described within <characteristic”> tags and they must be inside the <service> tags of the service they belong to.

A service may have one or more characteristics. The GAP service, used as an example, contains two characteristics, which are:

- **Device name (UUID: 2A00)**

  This is a device’s user friendly name (similar to the friendly name used in *Bluetooth* classic)

- **Device appearance (UUID: 2A01)**

  This identifies the devices type (similar to the Class-of-Device used in *Bluetooth* classic)

Characteristics also must have unique UUIDs and they need to be defined in the GATT database as shown above.

Standardized characteristics are defined in Characteristic Specification and the standardized characteristics are available at:

https://www.bluetooth.com/specifications/gatt/characteristics

Characteristics properties:

Service characteristics are described using the <properties> tag. The properties define how the characteristic can be accessed by a remote device. In the GAP service both the values are defined read only. Now since the values are read only they can be marked as **const** meaning the values are constant and they will be stored on the flash memory during the firmware installation.

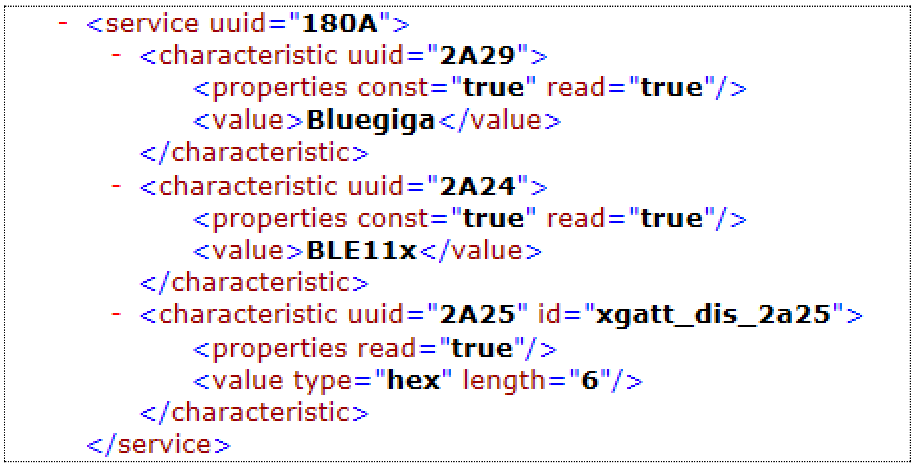
Characteristics values:

The characteristic’s value is defined within the <value and </value> tags. The device appearance is a hex value, so **hex** flag is used.

The exact syntax and more examples of services and characteristics definitions can be found from the *Profile Toolkit Developer Guide*.

5.4.2 Device ID

The second service implemented in our example is the Device ID service. The DI service exposes information about the vendor of the device and optional information for example about the devices firmware and hardware versions. In this example we implement a fairly minimalistic DI service with only a few characteristics. The DI service description is very similar to the GAP service and has only a few differences. The XML description is shown below.



```
- <service uuid="180A">
  - <characteristic uuid="2A29">
      <properties const="true" read="true"/>
      <value>Bluegiga</value>
    </characteristic>
  - <characteristic uuid="2A24">
      <properties const="true" read="true"/>
      <value>BLE11x</value>
    </characteristic>
  - <characteristic uuid="2A25" id="xgatt_dis_2a25">
      <properties read="true"/>
      <value type="hex" length="6"/>
    </characteristic>
  </service>
```

Figure 13: Device ID service

The global UUID for the DI service is: 180A

Link: Device ID Service

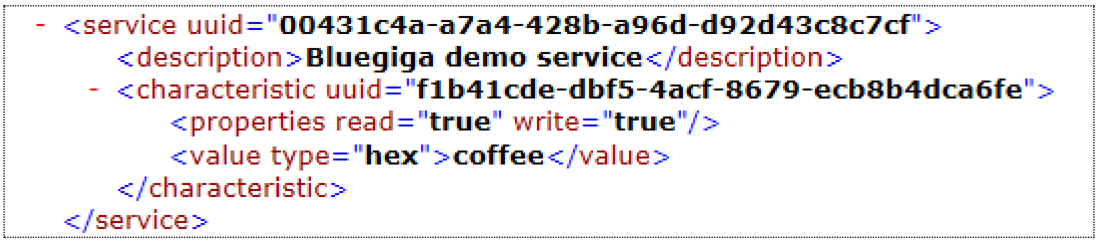
Three characteristic are defined and they are Manufacturer Name String, Model Number String and Serial Number String. All of these characteristics have UTF-8 format and they are **ready only** values. The two first values we permanently store to the flash and mark them **const**, but the third value is unique to every device and later we want to be able to modify the value with our BGscript code. Therefore we do no mark it const and we also define and ID for the value **xgatt_dis_2a35** which we later use in the BGScript code **to** write the devices Bluetooth address to the serial number string.

Link: Manufacturer Name String, Model Number String and Serial Number String

5.4.3 A Manufacturer Specific Service

The third service used in this example is a manufacturer specific service. *Bluetooth* Smart devices can have services defined by manufacturers which are not standardized by the *Bluetooth* SIG. The service structure is exactly the same however, manufacturer specific services MUST use 128-bit long UUIDs instead of the 16-bit UUIDs reserved for the standardized services.

The 128-bit UUIDs do not need to be applied or registered, but can be generated using for example online tools such as this site: http://www.uuidgenerator.net/



```
- <service uuid="00431c4a-a7a4-428b-a96d-d92d43c8c7cf">
    <description>Bluegiga demo service</description>
  - <characteristic uuid="f1b41cde-dbf5-4acf-8679-ecb8b4dca6fe">
      <properties read="true" write="true"/>
      <value type="hex">coffee</value>
    </characteristic>
  </service>
```

Figure 14: Proprietary service

The example service above has one characteristic which can be either read or written.

5.5 Writing BGScript Code

This example implements a standalone sensor device without an external host processor. The sensor side application is created with BGScript scripting language and the code is shown below.

BGScript uses an event based programming approach. The script is executed when an event takes place, and the programmer may register listeners for various events.

Our example BGDemo application uses the following BGScript code.

```
dim tmp(10)
dim addr(6)

# Boot Event listener
event system_boot(major ,minor ,patch ,build ,ll_version ,protocol_version ,hw )

        #Read local BT address
        call system_address_get( )(addr(0:6))

        # Write BT address to DI service serial number string
        call attributes_write(xgatt_dis_2a25,0,6,addr(0:5))

        #set bondable mode
        call sm_set_bondable_mode(1)

        #set to advertising mode
        call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
end

# Disconnection event listener
event connection_disconnected(handle,result)
        #connection disconnected, continue advertising
        call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
end
```

The BGScript has two event listeners defined:

- `system_boot(…)` **event listener**

  When the system is started (power up) a boot event is generated and this event listener will catch it. This event is the entry point for all the BGScript applications. In the example above, when the system is started, the BGSCript code reads the local devices MAC (Bluetooth) address, stores it to the Device ID services serial number string, enables bonding mode in case the remote device wants to do pairing and finally starts the advertisement procedure so the device becomes visible and connectable to other devices.

- `connection_disconnected(…)` **event listener**

  The second event handler is executed when a *Bluetooth* Smart connection is lost or closed by the remote device and it simply enables the advertisement mode again.

The BGScript functions and events can be found from the *Bluetooth Smart Software API reference* document.

5.6 Compiling and Installing the Firmware

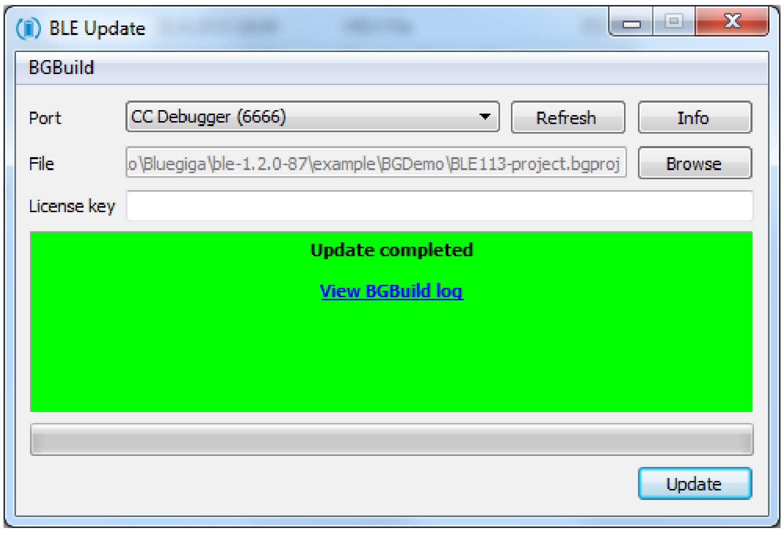
5.6.1 Using BLE Update tool

When you want to test your project, you need to compile the hardware settings, the GATT data base and BGScript code into a firmware binary file. The easiest way to do this is with the BLE Update tool that can be used to compile the project and install the firmware to a Bluetooth Smart Module using a CC debugger tools

In order to compile and install the project:

1. Connect CC debugger to the PC via USB
2. Connect the CC debugger to the debug interface on the BLE112 or BLE113
3. Press the button on CC debugger and make sure the led turns green
4. Start **BLE Update** tool
5. Make sure the CC debugger is shown in the **Port** drop down list
6. Use Browse to locate your **project** file (for example **BLE113-project.bgproj**)
7. Press **Update**

   BLE Update tool will compile the project and install it into the target device.



Port: CC Debugger (6666)
File: o\Bluegiga\ble-1.2.0-87\example\BGDemo\BLE113-project.bgproj
License key:
Update completed
View BGBuild log

Figure 15: Compile and install with BLE Update tool

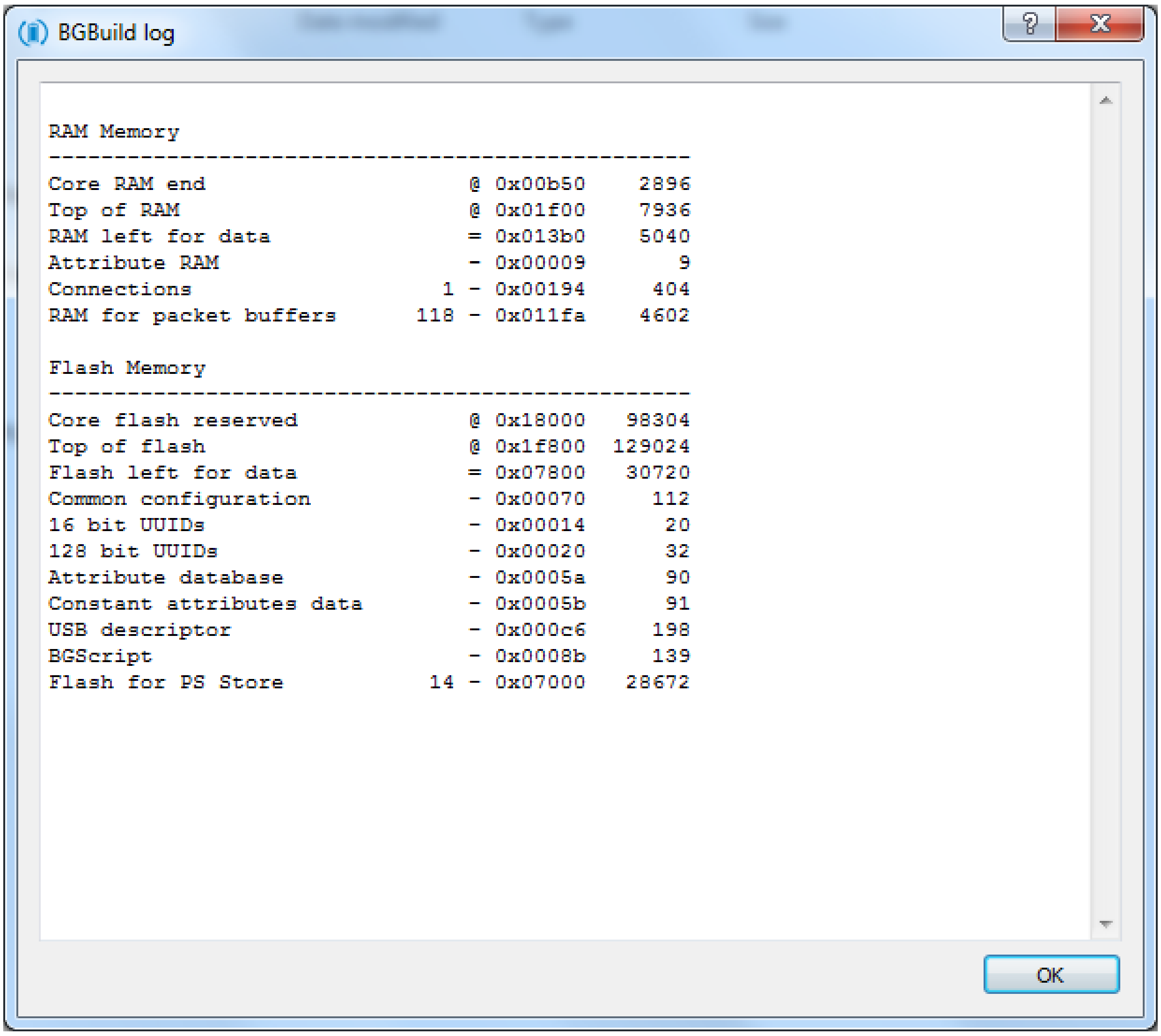
Note:

You can also double clikc the .BGPROJ file and it will automatically open the BLE Update tool.

If you have BLE113 Development Kit v.1.2 the CC debugger component is already placed on the kit and you simply need to:

- Connect the **DEBUGGER** USB port to the PC
- Turn the **DEBUGGER** switch to **MODULE**
- Press the **RESET DEBUGGER** button and make sure the **DEBUGGER** led turns green

The **View Build Log** opens up a dialog that shows the bgbuild compilere output and the RAM and Flash memory allocations.



```
RAM Memory
--------------------------------------------------
Core RAM end                    @ 0x00b50    2896
Top of RAM                      @ 0x01f00    7936
RAM left for data               = 0x013b0    5040
Attribute RAM                   - 0x00009       9
Connections                   1 - 0x00194     404
RAM for packet buffers      118 - 0x011fa    4602

Flash Memory
--------------------------------------------------
Core flash reserved             @ 0x18000   98304
Top of flash                    @ 0x1f800  129024
Flash left for data             = 0x07800   30720
Common configuration            - 0x00070     112
16 bit UUIDs                    - 0x00014      20
128 bit UUIDs                   - 0x00020      32
Attribute database              - 0x0005a      90
Constant attributes data        - 0x0005b      91
USB descriptor                  - 0x000c6     198
BGScript                        - 0x0008b     139
Flash for PS Store           14 - 0x07000   28672
```

Figure 16: BLE Update build log

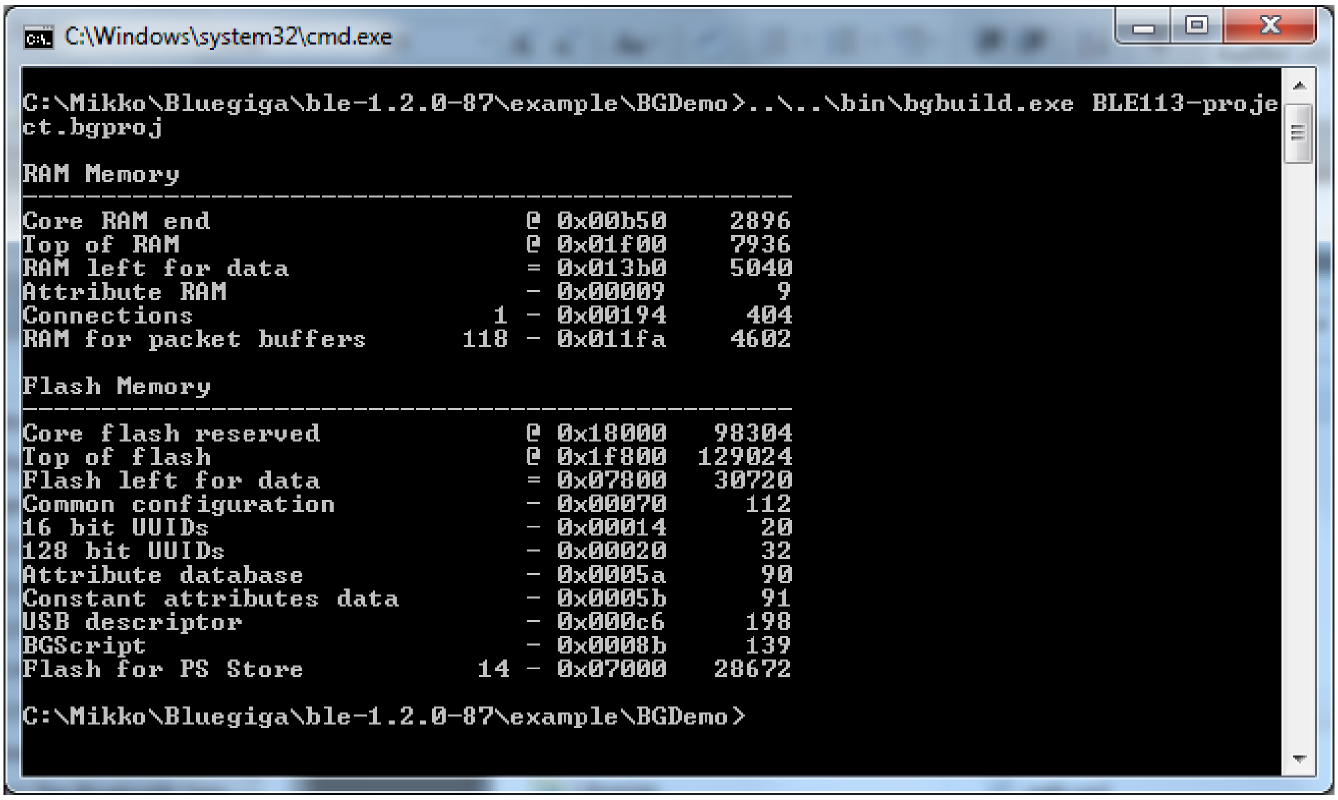
5.6.2 Compiling Using bgbuild.exe

The project can also be compiled with the **bgbuild.exe** command line compiler. The BGBuild compiler simply generates the firmware image file, which can be installed to the BLE112 or BLE113.

In order to compile the project using BGBuild:

1. Open Windows Command Prompt (cmd.exe)
2. Navigate to the directory where your project is
3. Execute BGbuild.exe compiler

   Syntax: ***bgbuild.exe <project file>***



C:\Windows\system32\cmd.exe

```
C:\Mikko\Bluegiga\ble-1.2.0-87\example\BGDemo>..\..\bin\bgbuild.exe BLE113-proje
ct.bgproj

RAM Memory
-------------------------------------------------
Core RAM end                     @ 0x00b50    2896
Top of RAM                       @ 0x01f00    7936
RAM left for data                = 0x013b0    5040
Attribute RAM                    - 0x00009       9
Connections                    1 - 0x00194     404
RAM for packet buffers       118 - 0x011fa    4602

Flash Memory
-------------------------------------------------
Core flash reserved              @ 0x18000   98304
Top of flash                     @ 0x1f800  129024
Flash left for data              = 0x07800   30720
Common configuration             - 0x00070     112
16 bit UUIDs                     - 0x00014      20
128 bit UUIDs                    - 0x00020      32
Attribute database               - 0x0005a      90
Constant attributes data         - 0x0005b      91
USB descriptor                   - 0x000c6     198
BGScript                         - 0x0008b     139
Flash for PS Store            14 - 0x07000   28672

C:\Mikko\Bluegiga\ble-1.2.0-87\example\BGDemo>
```

Figure 17: Compiling with BGBuild.exe

If the compilation is successful a .HEX file is generated, which can be installed into a Bluetooth Smart Module.

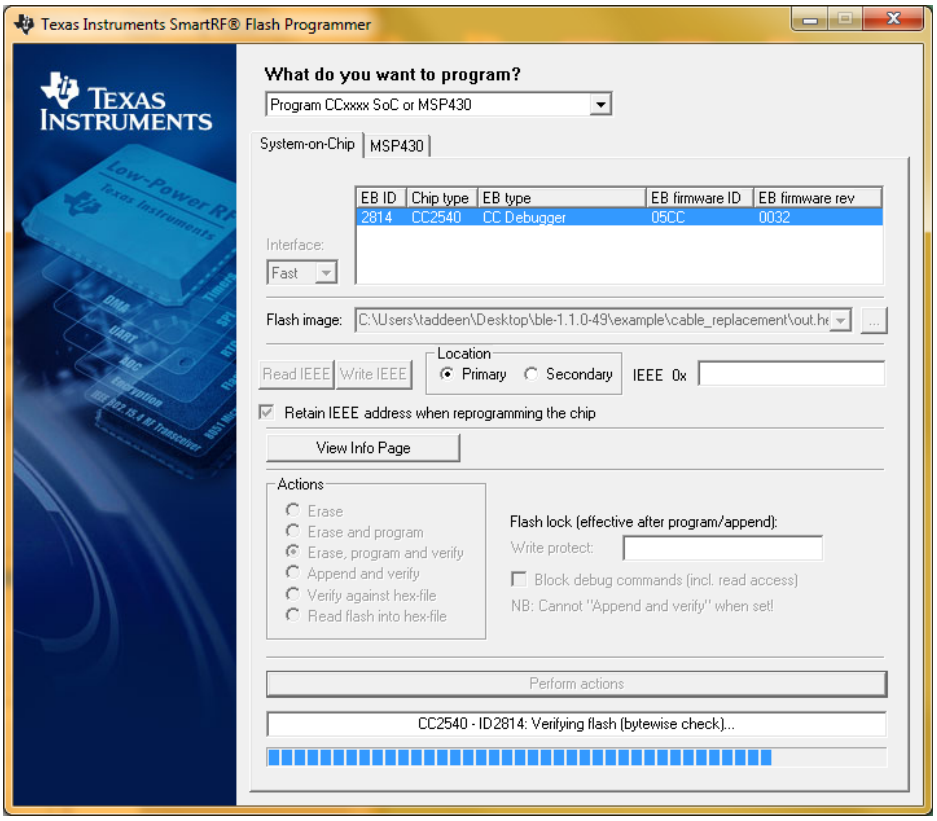
On the other hand if the compilation fails due to syntax errors in the BGScript or GATT files, and error message is printed.

5.6.3 Installing the firmware with TI's Flash Tool

Texas Instruments flash tool can also be used to install the firmware into the target device using the CC debugger.

In order to install the firmware with TI flash tool:

1. Connect CC debugger to the PC via USB
2. Connect the CC debugger to the debug interface on the BLE112
3. Press the button on CC debugger and make sure the led turns green
4. Start **TI flash tool** tool
5. Select program **CCxxxx SoC or MSP430**
6. Make sure the target device is recognized and displayed in the System-on-Chip field
7. Make sure **Retain IEEE address..** field is checked
8. Select the .HEX file you want to program to the target device
9. Select **Erase, Program and Verify**
10. Finally press **Perform actions** and make sure the installation is successful



Texas Instruments SmartRF® Flash Programmer

What do you want to program? Program CCxxxx SoC or MSP430

System-on-Chip | MSP430

EB ID	Chip type	EB type	EB firmware ID	EB firmware rev
2814	CC2540	CC Debugger	05CC	0032

Interface: Fast

Flash image: C:\Users\taddeen\Desktop\ble-1.1.0-49\example\cable_replacement\out.he

Read IEEE | Write IEEE | Location: [✓ Primary] Secondary | IEEE 0x

[✓] Retain IEEE address when reprogramming the chip

View Info Page

Actions: Erase | Erase and program | [✓ Erase, program and verify] | Append and verify | Verify against hex-file | Read flash into hex-file

Flash lock (effective after program/append):

Write protect:

Block debug commands (incl. read access)

NB: Cannot "Append and verify" when set!

Perform actions

CC2540 - ID2814: Verifying flash (bytewise check)...

Figure 18: TI's flash programmer tool

Note:

TI Flash tool should **NOT** be used with the Bluegiga *Bluetooth* Smart SDK v.1.1 or newer, but BLE Update tool should be used instead. The BLE112 and BLED112 devices contain a security key, which is needed for the firmware to operate and if the device is programmed with TI flash tool, this security key will be erased.

6 Testing the BGDemo Sensor

6.1 Using BLEGUI

This section describes how to test the BGDemo sensor implementation using BLEGUI software.

BLEGUI is a simple PC utility that can be used to control a Bluegiga *Bluetooth* Smart device over UART or USB. BLEGUI software sends the BGAPI commands to the device and parses the reponses and has a simple user interface to display device data.

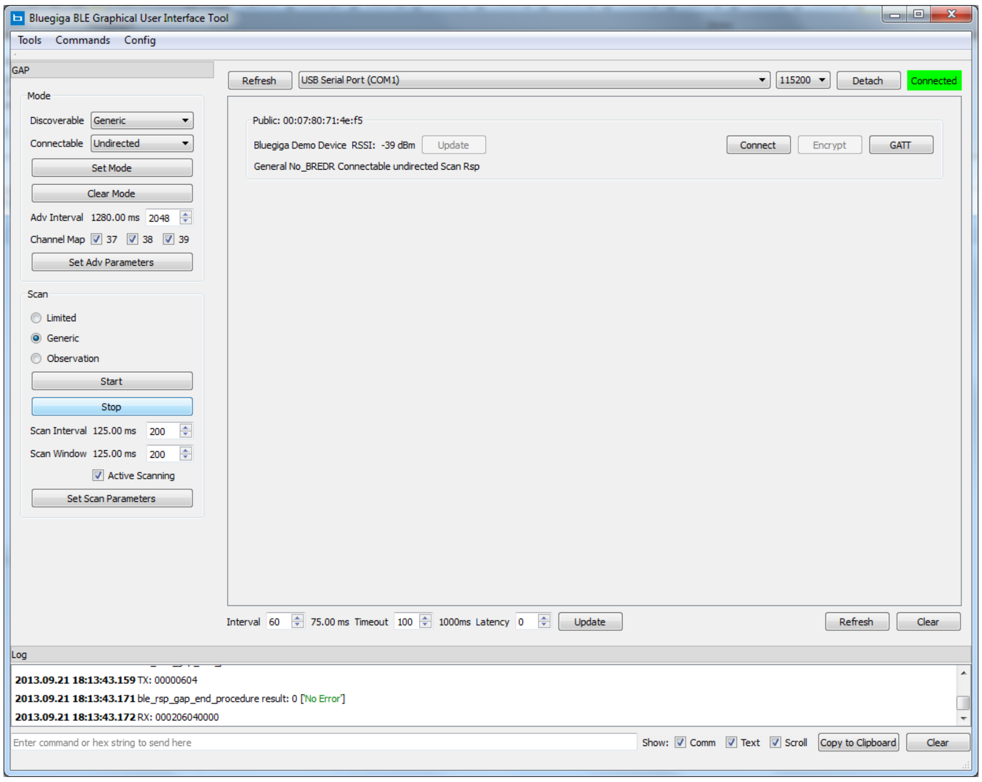
6.1.1 Discovering the BGDemo Sensor

- Connect for example a BLED112 USB dongle to your PC
- Make use the USB/CDC driver gets installed and a Virtual COM port gets created
- Open BLEGUI software and attach the device in the virtual COM port to the BLEGUI

As soon as the BGDemo sensor is powered on it starts to advertise. A BLED112 USB dongle can for example be used to scan for the sensor.

- Enable **Active Scanning**
- Press **Set Scan Parameters**
- Select **Generic** scan mode
- Press **Scan**

If the BGDemo device is power on and the BGDemo application is installed to is you should see the device in the BLEGUI software.



USB Serial Port (COM1) 115200

Discoverable: Generic
Connectable: Undirected
Adv Interval 1280.00 ms 2048
Channel Map ✓ 37 ✓ 38 ✓ 39

Scan: Generic
Scan Interval 125.00 ms 200
Scan Window 125.00 ms 200
✓ Active Scanning

Public: 00:07:80:71:4e:f5
Bluegiga Demo Device RSSI: -39 dBm
General No_BREDR Connectable undirected Scan Rsp

Interval 60 75.00 ms Timeout 100 1000ms Latency 0

```
2013.09.21 18:13:43.159 TX: 00000604
2013.09.21 18:13:43.171 ble_rsp_gap_end_procedure result: 0 ['No Error']
2013.09.21 18:13:43.172 RX: 000206040000
```

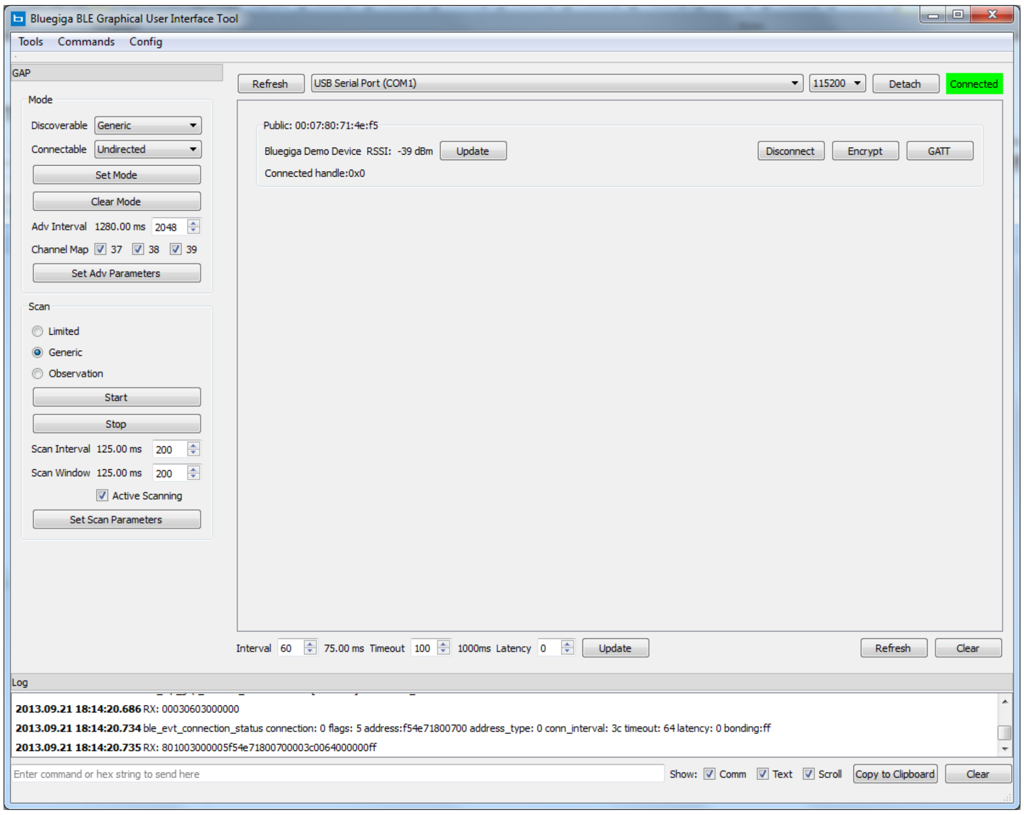
Enter command or hex string to send here

Show: ✓ Comm ✓ Text ✓ Scroll

Figure 19: Discoverting the BGDemo device

6.1.2 Establishing a Connection

Simply select the BGDemo sensor device and press the **Connect** button in the BLEGUI application.



Discoverable: Generic
Connectable: Undirected
Adv Interval: 1280.00 ms 2048
Channel Map: ✓ 37 ✓ 38 ✓ 39
Scan: Limited [✓ Generic] Observation
Scan Interval: 125.00 ms 200
Scan Window: 125.00 ms 200
✓ Active Scanning
USB Serial Port (COM1) 115200
Public: 00:07:80:71:4e:f5
Bluegiga Demo Device RSSI: -39 dBm
Connected handle:0x0
Interval 60 75.00 ms Timeout 100 1000ms Latency 0

```
2013.09.21 18:14:20.686 RX: 00030603000000
2013.09.21 18:14:20.734 ble_evt_connection_status connection: 0 flags: 5 address:f54e71800700 address_type: 0 conn_interval: 3c timeout: 64 latency: 0 bonding:ff
2013.09.21 18:14:20.735 RX: 801003000005f54e71800700003c0064000000ff
```

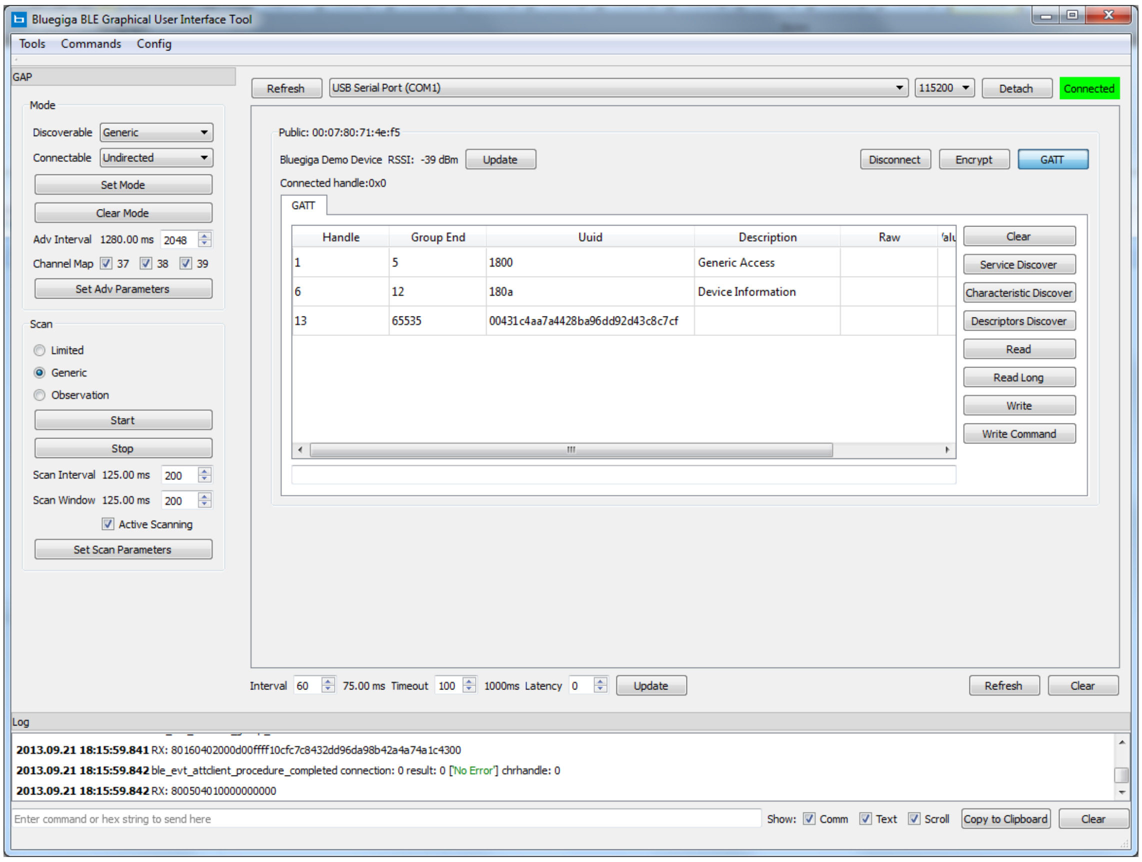
Enter command or hex string to send here
Show: ✓ Comm ✓ Text ✓ Scroll

Figure 20: Opening a connection

6.1.3 Making GATT Service Discovery

In order to see the supplied services in the BGDemo device do the following steps

- Press the **GATT** button to start GATT tool
- Press **Service discover** button to start a GATT primary service discovery procedure



Public: 00:07:80:71:4e:f5

Bluegiga Demo Device RSSI: -39 dBm

Connected handle:0x0

Handle	Group End	Uuid	Description	Raw	'alu
1	5	1800	Generic Access		
6	12	180a	Device Information		
13	65535	00431c4aa7a4428ba96dd92d43c8c7cf			

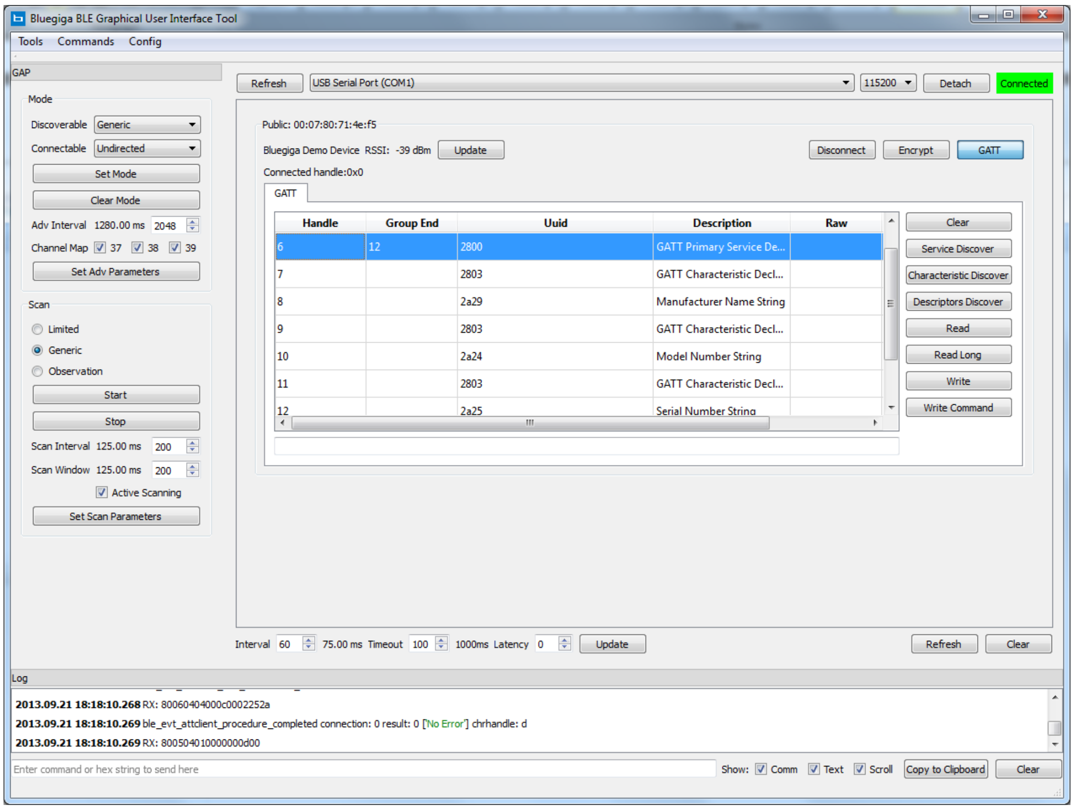
```
2013.09.21 18:15:59.841 RX: 80160402000d00ffff10cfc7c8432dd96da98b42a4a74a1c4300
2013.09.21 18:15:59.842 ble_evt_attclient_procedure_completed connection: 0 result: 0 ['No Error'] chrhandle: 0
2013.09.21 18:15:59.842 RX: 8005040100000000000
```

Figure 21: GATT service discovery

The three services defined in the GATT data base are visible in the device.

6.1.4 Reading the Serial Number String

- To read the DI service's serial number string, which contains the MAC address, do the following steps:
    - Select the Device ID service (UUID: 180A)
    - Make **Descriptors discovery**



Discoverable: Generic
Connectable: Undirected
Adv Interval 1280.00 ms 2048
Channel Map ✓ 37 ✓ 38 ✓ 39
Scan: Limited [✓ Generic] Observation
Scan Interval 125.00 ms 200
Scan Window 125.00 ms 200
✓ Active Scanning

Public: 00:07:80:71:4e:f5
Bluegiga Demo Device RSSI: -39 dBm
Connected handle:0x0

Handle	Group End	Uuid	Description	Raw
6	12	2800	GATT Primary Service De...	
7		2803	GATT Characteristic Decl...	
8		2a29	Manufacturer Name String	
9		2803	GATT Characteristic Decl...	
10		2a24	Model Number String	
11		2803	GATT Characteristic Decl...	
12		2a25	Serial Number String	

Interval 60 75.00 ms Timeout 100 1000ms Latency 0

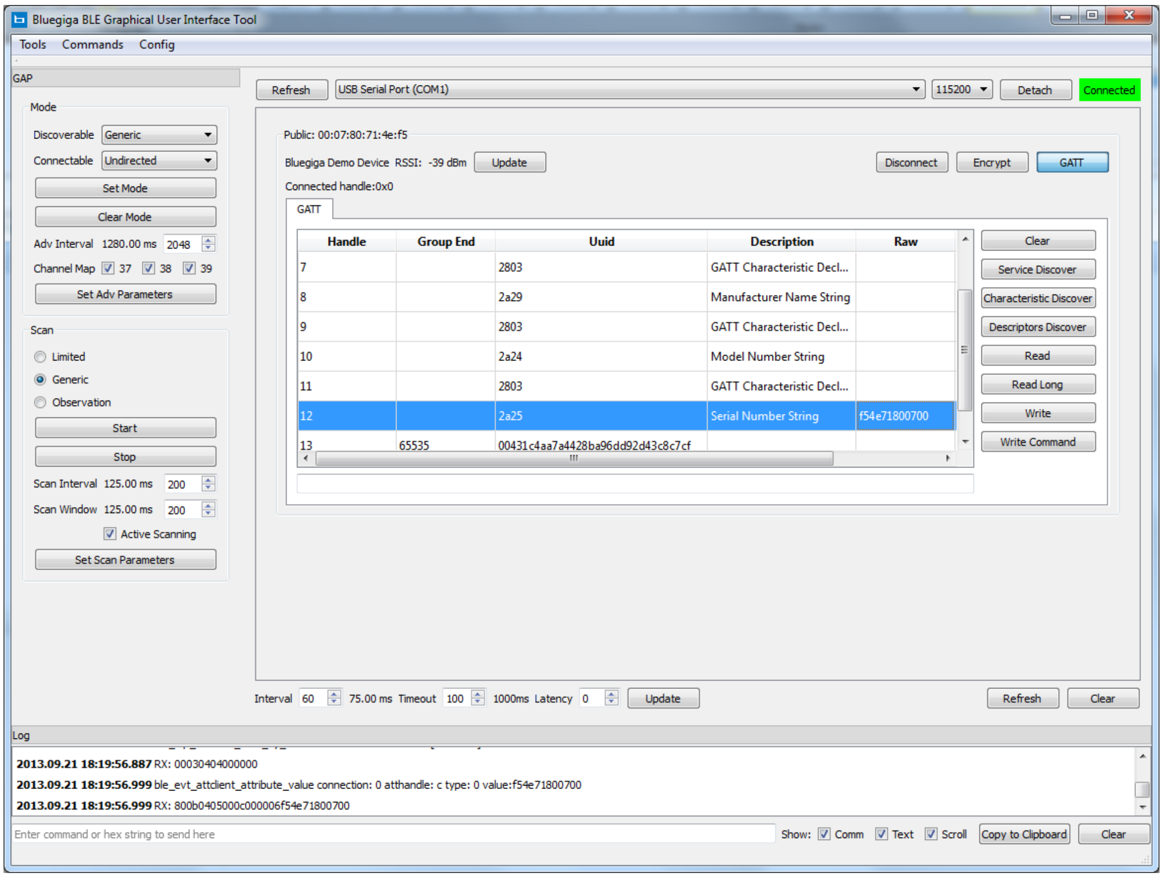
```
2013.09.21 18:18:10.268 RX: 80060404000c0002252a
2013.09.21 18:18:10.269 ble_evt_attclient_procedure_completed connection: 0 result: 0 ['No Error'] chrhandle: d
2013.09.21 18:18:10.269 RX: 800504010000000d00
```

Enter command or hex string to send here
Show: ✓ Comm ✓ Text ✓ Scroll

Figure 22: GATT descriptor discovery

The Serial Number String is stored in the UUID a2a5 as defined in the GATT database. The value is read only and to read it:

- Select the **Serial Number String** characteristic
- Press **Read**



Mode

Discoverable: Generic
Connectable: Undirected
Adv Interval 1280.00 ms 2048
Channel Map ✓ 37 ✓ 38 ✓ 39

Scan

Limited
[✓ Generic]
Observation

Scan Interval 125.00 ms 200
Scan Window 125.00 ms 200
✓ Active Scanning

Public: 00:07:80:71:4e:f5

Bluegiga Demo Device RSSI: -39 dBm

Connected handle:0x0

Handle	Group End	Uuid	Description	Raw
7		2803	GATT Characteristic Decl...	
8		2a29	Manufacturer Name String	
9		2803	GATT Characteristic Decl...	
10		2a24	Model Number String	
11		2803	GATT Characteristic Decl...	
12		2a25	Serial Number String	f54e71800700
13	65535	00431c4aa7a4428ba96dd92d43c8c7cf		

Interval 60 75.00 ms Timeout 100 1000ms Latency 0

Log

```
2013.09.21 18:19:56.887 RX: 00030404000000
2013.09.21 18:19:56.999 ble_evt_attclient_attribute_value connection: 0 atthandle: c type: 0 value:f54e71800700
2013.09.21 18:19:56.999 RX: 800b0405000c000006f54e71800700
```

Enter command or hex string to send here

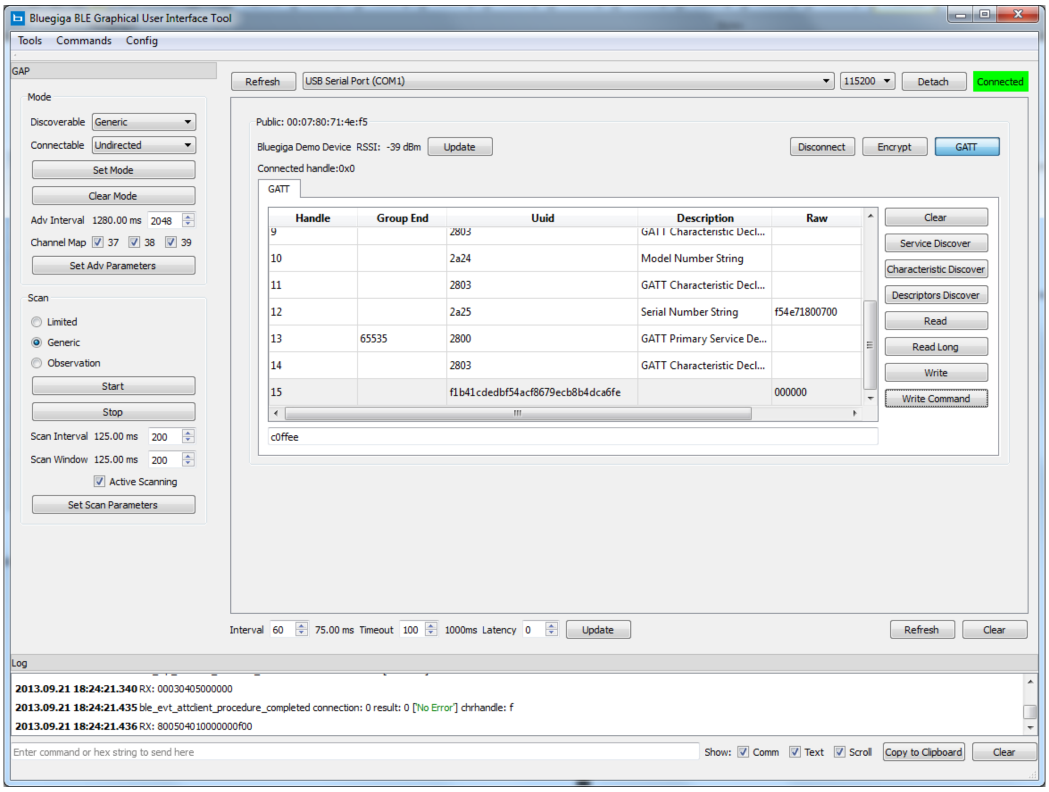
Figure 23: Reading Serial Number String

The MAC address is displayed in the **Raw** column.

6.2 Reading and Writing the Manufacturer Specific Service

In order to write and read the value of our proprietary characteristic

- **Connect** to the BGDemo sensor
- Make GATT **service discovery**
- Select the proprietary service and make **descriptor discovery**
- Press **Read** in order to read the value:
    - Note that the value does not contain any real data by default, since it was not marked as **const** but zero's are returned
- To write the value:
    - Select the proprietary characteristic
    - Write the desired value to the line below the GATT view (c0ffee)
    - Press **Write**



Mode

Discoverable: Generic
Connectable: Undirected
Adv Interval 1280.00 ms 2048
Channel Map ☑ 37 ☑ 38 ☑ 39

Scan

○ Limited
● Generic
○ Observation

Scan Interval 125.00 ms 200
Scan Window 125.00 ms 200
☑ Active Scanning

Public: 00:07:80:71:4e:f5

Bluegiga Demo Device RSSI: -39 dBm Update

Connected handle:0x0

Handle	Group End	Uuid	Description	Raw
9		2803	GATT Characteristic Decl...	
10		2a24	Model Number String	
11		2803	GATT Characteristic Decl...	
12		2a25	Serial Number String	f54e71800700
13	65535	2800	GATT Primary Service De...	
14		2803	GATT Characteristic Decl...	
15		f1b41cdedbf54acf8679ecb8b4dca6fe		000000

c0ffee

Interval 60 75.00 ms Timeout 100 1000ms Latency 0 Update

Log

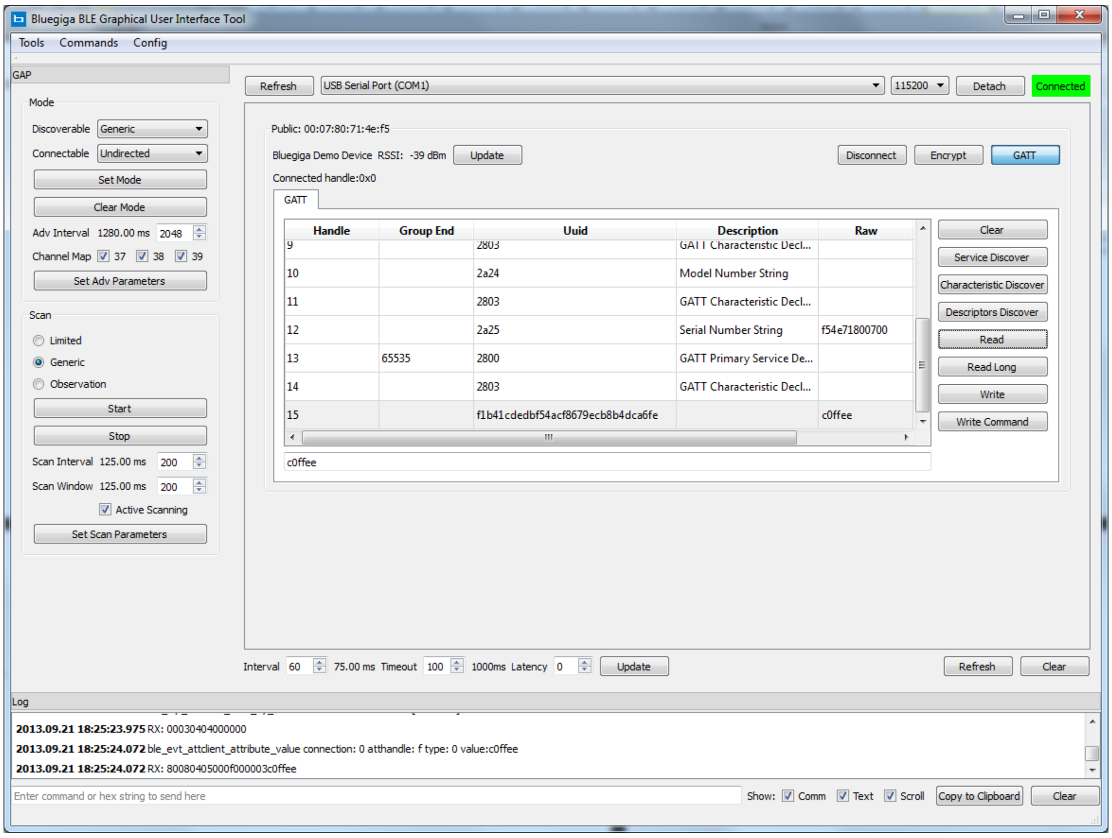
```
2013.09.21 18:24:21.340 RX: 00030405000000
2013.09.21 18:24:21.435 ble_evt_attclient_procedure_completed connection: 0 result: 0 ['No Error'] chrhandle: f
2013.09.21 18:24:21.436 RX: 800504010000000f00
```

Enter command or hex string to send here

Show: ☑ Comm ☑ Text ☑ Scroll

Figure 24: Writing a characteristic value

- To make sure the value got written, simply read it again.



Public: 00:07:80:71:4e:f5

Bluegiga Demo Device RSSI: -39 dBm

Connected handle:0x0

Handle	Group End	Uuid	Description	Raw
9		2803	GATT Characteristic Decl...	
10		2a24	Model Number String	
11		2803	GATT Characteristic Decl...	
12		2a25	Serial Number String	f54e71800700
13	65535	2800	GATT Primary Service De...	
14		2803	GATT Characteristic Decl...	
15		f1b41cdedbf54acf8679ecb8b4dca6fe		c0ffee

c0ffee

Interval 60 75.00 ms Timeout 100 1000ms Latency 0

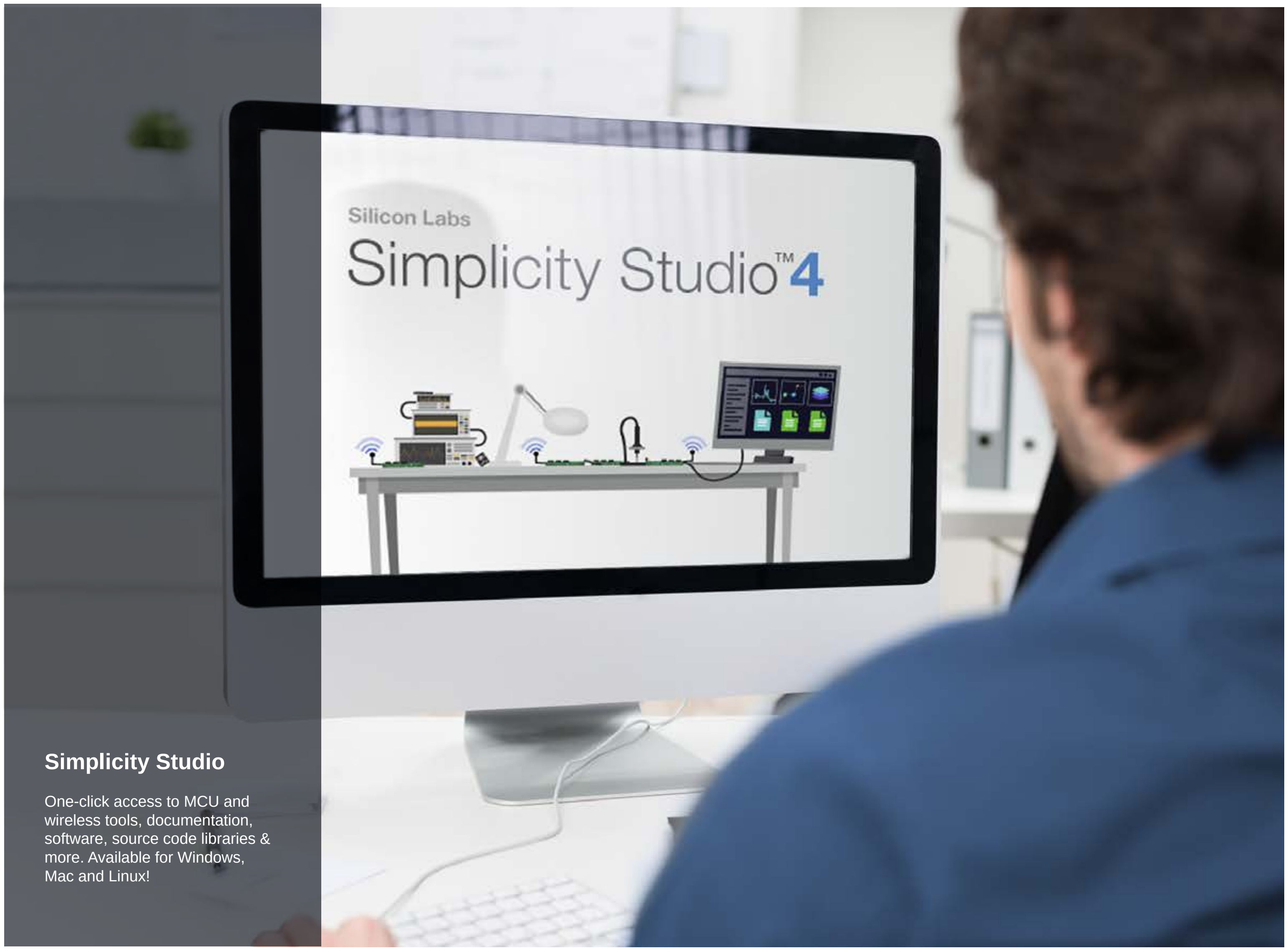
```
2013.09.21 18:25:23.975 RX: 00030404000000
2013.09.21 18:25:24.072 ble_evt_attclient_attribute_value connection: 0 atthandle: f type: 0 value:c0ffee
2013.09.21 18:25:24.072 RX: 80080405000f000003c0ffee
```

Note:

If you reset the BGDemo sensor the value written to the proprietary characteristic will be lost, since the example BGScript code will not store the value to the flash memory.

If you want to store the value permanently use for example the PS key API commands to write the value to the PS key storage in your BGScript code.

Disconnecting from the device will keep the characteristic value, since as long as the software runs, the value will be kept in RAM.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio	SW/HW	Quality	Support and Community
www.silabs.com/IoT	*www.silabs.com/simplicity*	*www.silabs.com/quality*	*community.silabs.com*

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com