BLUETOOTH SMART SOFTWARE

Implementing Over-the-Air Firmware Upgrade

Tuesday, 04 March 2014

Version 1.7

VERSION HISTORY

Version	Comment
1.0	First version
1.1	Hardware reference added
1.2	Minor updates
1.3	Updated reference schematic
1.4	Updated Introduction chapter
1.5	Updated to match the Bluetooth Smart Software v.1.2.2 - external SPI flash board instructions added - 256kB instructions added
1.6	BLE113-A-256 instructions added
1.7	OTA Data Attribute length fixed to 20 bytes.

TABLE OF CONTENTS

1 Introduction

This application note discusses how to enable and perform Over-the-Air (OTA) firmware upgrade using the Bluegiga *Bluetooth* Smart Module and Software

The application note contains a practical example of how to build *Bluetooth* Smart GATT based OTA update services with the profile toolkit, how to make a BGScript application that performs the firmware upgrade.

An assumption is made that the reader of this application note is already somewhat familiar with the *Bluetooth* Smart SDK.

The OTA feature is available in the *Bluetooth* Smart Software and SDK v.1.2.2 and newer.

The OTA firmware update either requires an external SPI flash connected to the SPI interface of BLE112 or BLE113 or 256kB flash version of BLE113 (part number: BLE113-A-256).

2 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)

2.1 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.

2.2 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 using a simple scripting language called BGScript™. 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.

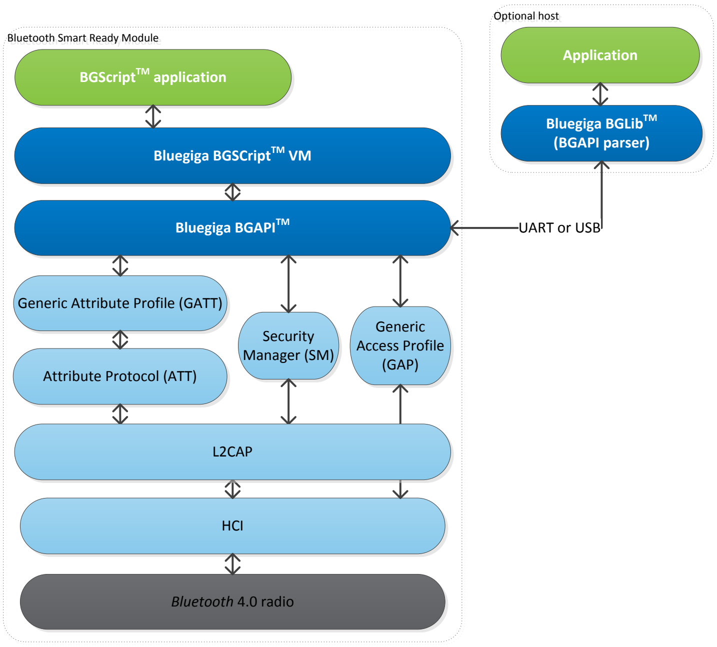

Figure 1: *Bluetooth Smart* Software

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.

2.3 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.

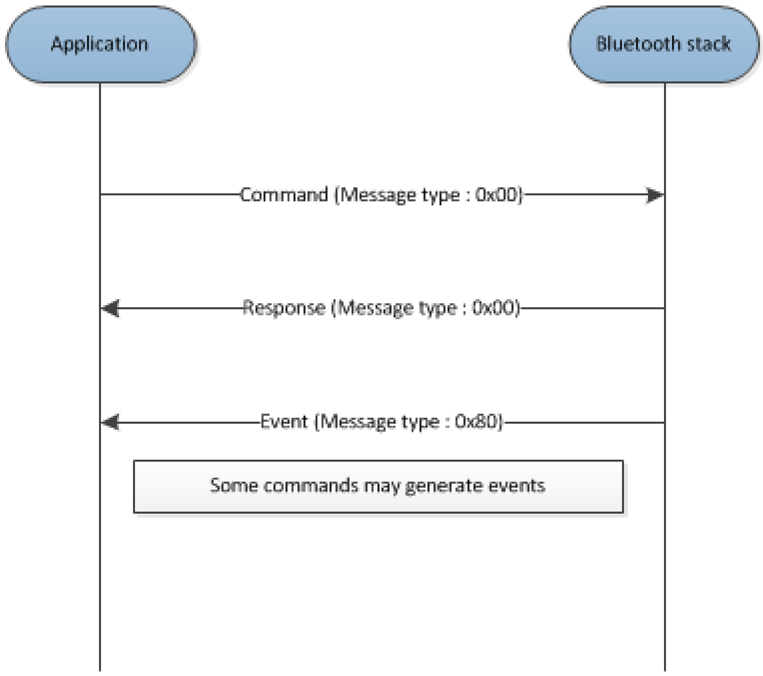

Figure 2: BGAPI protocol

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

2.4 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.

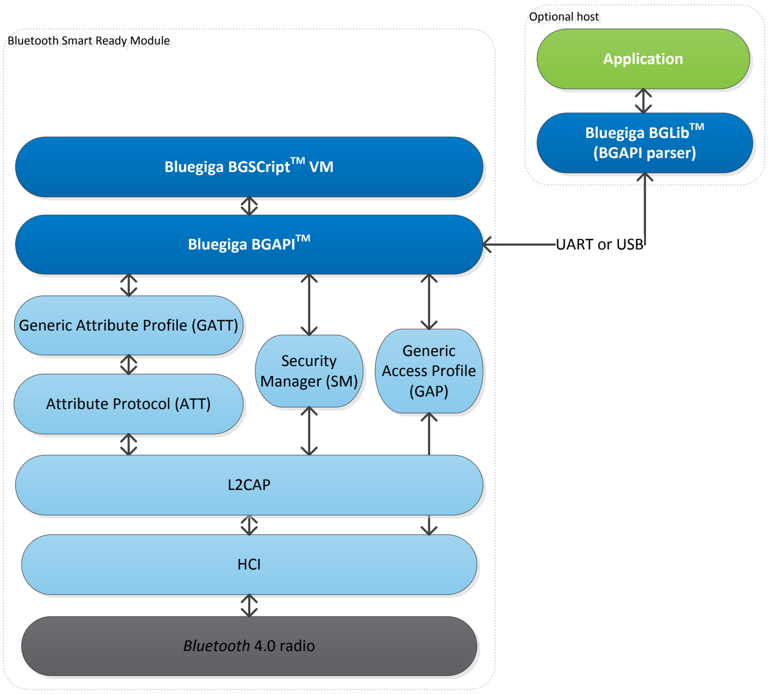

Figure 3: BGLib host library

2.5 BGScript™ 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.

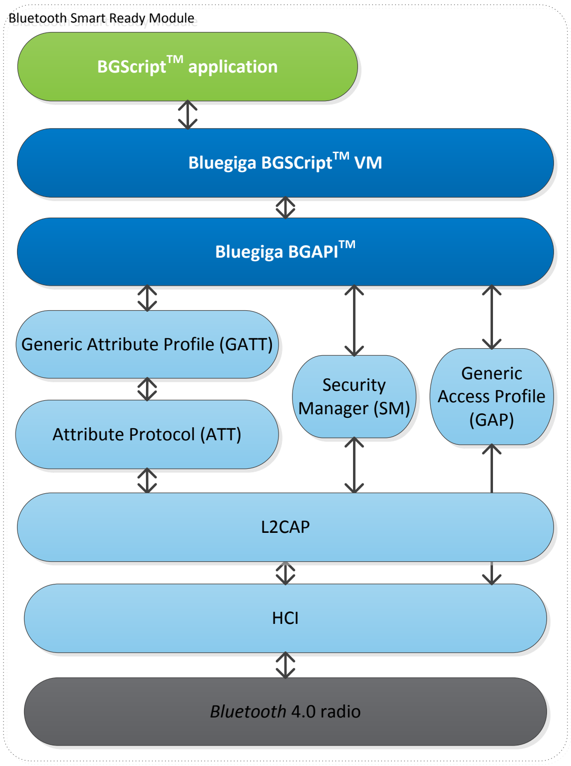

Figure 4: BGScript application model

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
```

2.6 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 5: A profile toolkit example of GAP service

3 Implementation of OTA Firmware Upgrade

In this chapter we describe and discuss an actual implementation of the OTA firmware upgrade and the necessary steps. The implementation consists of following steps:

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

3.1 Limitations of OTA firmware update

At the moment the OTA firmware update has the following limitations:

- Hardware configuration cannot be updated via OTA firmware update
- Bootloader cannot be changed with OTA firmware update
- OTA update requires at least 256kB flash memory (internal or external SPI flash)
- At the moment the OTA bootloader only works the Winbond SPI flash parts

3.2 Prerequisites

In order to perform the OTA firmware update at least 256kB of flash memory is needed. The standard BLE112 and BLE113 *Bluetooth* Smart Modules only have a 128kB flash memory, but you can simply connect a low cost SPI flash memory to one of the SPI interfaces in the BLE112 or BLE113 modules. Also a variant of BLE113 exist with on-board 256kB flash and it does not require an external SPI flash.

External SPI flash memories are typically very low cost and can be as cheap as $0.2-0.3.

Below is a reference schematic how to connect a 2 Mbit Winbond W25X20CL flash chip to the BLE113 Bluetooth Smart Module. Since February 2014 Both BLE112 and BLE113 development kits are supplied with a small carrier board containing a small carrier board with the Winbond flash memory.

3.2.1 Reference Schematic

The external flash is powered from one of the high current IO's on the BLE112 or BLE113. In this reference the flash supply is taken from P1_0. When the external flash is used, P1_0 is first driven high by the software in order to power up to power up the flash. When the flash is not used, all the lines connected to the flash are driven low to avoid leakage currents.

When the flash is not used the chip select output of the module is in high impedance so it requires a pull-up resistor. The modules MISO line requires a pull-up resistor to make sure it is pulled low instantly with the supply voltage when the flash is not used.

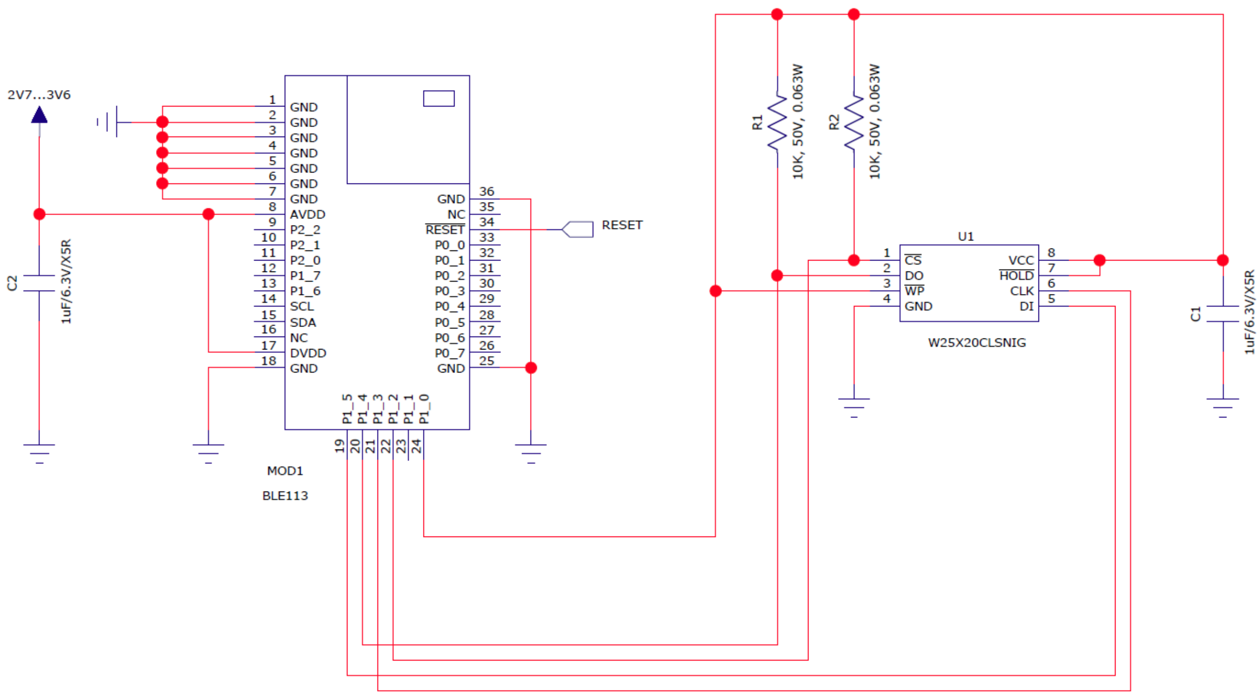

Figure 6: Example schematic

3.3 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)

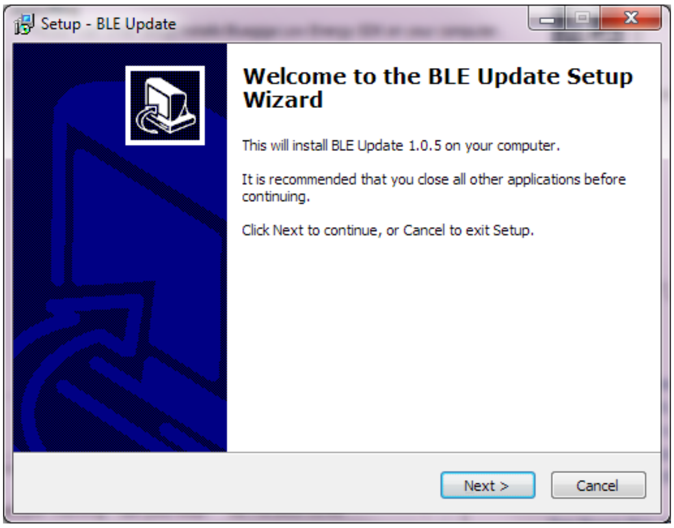

Figure 7: Installing Bluegiga *Bluetooth* Smart SDK

3.4 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="ota.bgs" />
    <image out="BLE112_OTA.hex" />
    <device type="ble112" />
    <boot fw="bootota" />
    <ota out="BLE112_OTA.ota" />
</project>
```

Figure 8: 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.
- <image> tag defines the output .HEX file containing the firmware image
- <device type> tag defines if the project is meant for BLE113 hardware
- <boot fw> tag defines which interface is enabled for DFU firmware upgrades. **bootota** feature is used since this example uses the OTA boot loader.
- <ota> tag defines the OTA firmware update file which is the actual firmware update file uploaded to the device over a Bluetooth LE connection.

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.

3.5 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 OTA demo application is shown below.

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

<hardware>
    <sleeposc enable="true" ppm="30" />
    <script enable="true" />
    <txpower power="15" bias="5" />
    <usart channel="0" mode="spi_master" alternate="2" polarity="negative" phase="0" endianness="msb" baud="2000000" endpoint="none" />
    <pmux regulator_pin="7" />
    <sleep enable="true" />
    <otaboot cs_port="1" cs_pin="2" power_port="1" power_pin="0" uart="0" />
</hardware>
```

Figure 9: Hardware configuration for the BLE112 *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. The BLE113 does incorporate the sleep oscillator so this value should be set to true especially in the applications where power consumption matters. The PPM value defines the sleep oscillator accuracy and MUST not be changed.
- <script> tag defines if BGScript VM and application are present. Since the example uses a BGscript application to perform the firmware upgrade we set this value to **true.**
- <txpower> tag defines the TX power level and the value 15 configures the maximum TX power level.
- <usart> tag is used to enable the SPI master interface at 2Mbps. The OTA firmware is uploaded to an external SPI flash (128kB of larger) and the SPI interface is used as the interface to the external flash chip.
- <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.
- <sleep> tag is used to enable the low power sleep modes on the device
- <otaboot> tag is used to define the SPI interface where the external SPI flash is located. This tag is only needed if the external SPI flash is used.

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.

3.5.1 Creating a Project for BLE113-A-256

BLE113-A-256 is a product variant of BLE113 with built-in 256kB flash memory and it does not require an external SPI flash to be used. A few changes in the project and hardware configuration need to be made for the BLE113-A-256 part and they are shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project>
    <gatt in="gatt.xml" />
    <hardware in="hardware256.xml" />
    <script in="ota256.bgs" />
    <image out="BLE113__256_OTA.hex" />
    <device type="ble113" memory="256" />
    <config in="config256.xml" />
    <boot fw="bootota" />
</project>
```

Figure 10: BLE113-A-256 project file

- <device type> tag defines if the project is meant for BLE113 hardware and memory="256" indicates that internal 256kB flash is available

The following change is needed to the application configuration file which is used to allocate the flash space needed for the firmware update to be loaded over-the air.

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
    <user_data size="0x20000" />
</config>
```

Figure 11: Config file for BLE113-A-256

- <user data> tag is used to allocate all the extra 128kB flash space for the firmware update.

Since there is no need to have the SPI interface configured for the external SPI flash the hardware configuration for BLE113-A-256 does not need to configure the SPI interface settings.

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

<hardware>
    <sleeposc enable="true" ppm="30" />
    <script enable="true" />
    <txpower power="15" bias="5" />
    <pmux regulator_pin="7" />
    <sleep enable="true" />
    <otaboot source="internal" />
</hardware>
```

Figure 12: BLE113-A-256 hardware configuration

3.6 Building the OTA Service

This section discusses the implementation of OTA GATT service using the Profile Toolkit™.

The figure below shows the OTA service required in the GATT database for the OTA update to work the service uses a 128-bit manufacturer specific UUID for both the service and the characteristics.

```
<service uuid="1d14d6ee-fd63-4fa1-bfa4-8f47b42119f0">

    <description>Bluegiga OTA Service</description>

    <characteristic uuid="f7bf3564-fb6d-4e53-88a4-5e37e0326063" id="ota_control">
        <properties write="true" />
        <value length="1" type="user" />
        <description>OTA Control Point Attribute</description>
    </characteristic>

    <characteristic uuid="984227f-34fc-4045-a5d0-2c581f81a153" id="ota_data">
        <properties write_no_response="true" />
        <value length="23" />
        <description>OTA Data Attribute</description>
    </characteristic>

</service>
```

Figure 13: OTA GATT Service

OTA service UUID: 1d14d6ee-fd63-4fa1-bfa4-8f47b42119f0

Two characteristics are required in the OTA service and they are:

Characteristic	UUID	Type	Length	Support	Security	Properties
OTA Control Point Attribute	f7bf3564-fb6d-4e53-88a4-5e37e0326063	hex	1 byte	Mandatory	none	Write
OTA Data Attribute	984227f-34fc-4045-a5d0-2c581f81a153	hex	20 bytes	Mandatory	none	Write without response

Table 1: OTA service characteristics description

The OTA control point attribute is used to control the firmware upgrade process between the device that will be updated and the device that performs the update it is a write only attribute to the control can only made by the device that performs the update. The OTA control attribute has the **user** property enabled, which means that the BGScript application will need to read to data and acknowledge it to the sender and the acknowledgement is NOT automatically handled by the *Bluetooth* stack.

The 2nd attribute is on the other hand used to transmit the data from the device that performs the update to the device that is being updated. It's a **write without response** so acknowledgements will not be passed to the application level, but handled automatically by the *Bluetooth* stack.

3.7 Writing the BGScript Code

The OTA example implements a simple BGScript application that demonstrates the OTA firmware capability and it can be used as an example and starting point to integrate the OTA firmware update to real applications. The application is implemented with BGScript scripting language and the code is explained in this chapter.

The BGScript code enables advertisements on the device so it can be discovered and connected by a remote device and it also receives the firmware update data from the remote device and stores it to an external SPI flash. Once the firmware data has been fully received the script initiates the actual update process, which is then handled by the OTA boot loader, which was enabled in the hardware configuration.

```
#init gap mode
event system_boot(major ,minor ,patch ,build ,ll_version ,protocol_version ,hw )

        #Set device to advertisement mode and allow undirected connections
        call gap_set_mode(2,2)

        # Initialize the DFU pointer
        dfu_pointer=0

        # Inti Flash retry counter and MAX retries
        retry_counter=0
        max_retries=10

        # set power pin as output and pull down
        # also set p1.1 to output (does not have internal pull-resistor)
        call hardware_io_port_config_pull(1,$7,1)
        call hardware_io_port_write(1,$7,0)
        call hardware_io_port_config_direction(1,$3)
end
```

To handle the incoming commands and DFU data from the device (DFU application) an event handler for received data must be written. The data received from the remote end can be handled with the **attributes_value** (…) event listener, which will catch an event whenever a GATT characteristic is written.

In the example application the handler only checks of the **OTA Control Point Attribute** or **OTA Data Attribute** are written. The **OTA Control Point Attribute** carries commands such as flash erase or DFU boot and the **OTA Data Attribute** carries the actual firmware update.

The event handled code is below and

```
# Incoming data event listener
# Handles OTA Control Point Attribute (commands) and OTA Data Attribute (firmware update) writes
# and performs the necessary actions
event attributes_value(connection, reason, handle, offset, value_len, value_data)

    # save connection handle, is always 0 if only slave
    curr_connection=connection

    # Check if OTA control point attribute is written by the remote device and execute the command
            # Command 0 : Erase flash block 0 (0x0-0x1FFFF)
            # Command 1 : Erase flash block 1 (0x10000-0x3FFFF)
            # Command 2 : Reset DFU data pointer
            # Command 3 : Boot to DFU mode
    # In case of errors application error code 0x80 is returned to the remote device
    if handle = ota_control then
        #attribute is user attribute, reason is always write_request_user
        if value_len >1 || offset >0 then
            # Not a valid command -> report application error code : 0x80
            call attributes_user_write_response(connection, $80)
        else
            command=value_data(0:1)
            if command = 0 then     # Command 0 received -> Erase block 0
                #reset retry counter
                retry_counter = 0
                # pull power and chip select pins up
                # write enable, cs down
                call hardware_io_port_config_direction(1,$7)
                call hardware_spi_transfer(0,1,"\x06")
                call hardware_io_port_config_direction(1,$3)

                # erase block 0 : 0-1ffff
                call hardware_io_port_config_direction(1,$7)
                call hardware_spi_transfer(0,4,"\xd8\x00\x00\x00")
                call hardware_io_port_config_direction(1,$3)

                # start timer to poll for erase complete
```

```
                call hardware_set_soft_timer(6000,0,1)
            end if

            if command = 1 then # Command 1 received -> Erase block 1

                #write enable
                call hardware_io_port_config_direction(1,$7)
                call hardware_spi_transfer(0,1,"\x06")
                call hardware_io_port_config_direction(1,$3)

                # erase block 1 : 10000-3ffff
                call hardware_io_port_config_direction(1,$7)
                call hardware_spi_transfer(0,4,"\xd8\x01\x00\x00")
                call hardware_io_port_config_direction(1,$3)

                # start timer to poll for erase complete
                call hardware_set_soft_timer(6000,0,1)
           end if

           if command = 2 then # Command 2 received -> Erase DFU pointer
                dfu_pointer=0
                call attributes_user_write_response(curr_connection, 0)
           end if

           if command = 3 then # Command 3 received -> Booth to DFU mode
                call system_reset(1)
           end if

           if command = 4 then # Command 4 received ->
           #pull power and chip select pins up
                call hardware_io_port_write(1,$1,$1)
                call attributes_user_write_response(curr_connection, $0)
            end if

           if command > 4 then # Unknown command -> report application error code : 0x80
                call attributes_user_write_response(curr_connection, $80)
           end if
       end if
   end if
```

```
    # Check if OTA data attribute is written which carries the firmware update
    # and store the data to the external SPI flash
    if handle = ota_data then
        # NOTE: when programming page, address cannot wrap over 256 byte boundary.
        # This must be handled in the remote DFU application
        # This is write no response attribute, no need to handle response to other end
        # TODO: handle zero length writes
        spi_response(0:1)=2             # page program command

        # flip endianness for address
        tmp(0:4)=dfu_pointer
        spi_response(1:1)=tmp(2:1)
        spi_response(2:1)=tmp(1:1)
        spi_response(3:1)=tmp(0:1)

        # enable SPI flash write mode
        call hardware_io_port_config_direction(1,$7)
        call hardware_spi_transfer(0,1,"\x06")
        call hardware_io_port_config_direction(1,$3)

        #write data
        call hardware_io_port_config_direction(1,$7)
        call hardware_spi_transfer(0,4,spi_response(0:4))
        # send data in next transfer, leave chip select asserted
        call hardware_spi_transfer(0,value_len,value_data(0:value_len))
        call hardware_io_port_config_direction(1,$3)
        # it can take up to 800 us for full page to program
        # loop couple of times for write to complete
        call hardware_io_port_config_direction(1,$7)
        call hardware_spi_transfer(0,2,"\x05\x00")(result,channel,spi_len,spi_response(0:2))
        # start polling
        a = spi_response(1:1)
        while a&1
            call hardware_spi_transfer(0,1,"\x00")(result,channel,spi_len,spi_response(0:1))
            a = spi_response(0:1)
        end while
        call hardware_io_port_config_direction(1,$3)
        # increase DFU offset
        dfu_pointer=dfu_pointer+value_len
    end if
end
```

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

An additional event handler is needed to check if the flash writes are ready and more data can be received from the remote end. The event handler below checks if the flash is ready or waits if it's not. Once the flash is ready a status code 0 is sent to the remote device indicating that more data can be received.

```
# Timer expired event handler
# Poll flash and if it's ready, and send response to the remote device (DFU application)
event hardware_soft_timer(handle)
      if handle=0 then
          call hardware_io_port_config_direction(1,$7)
          call hardware_spi_transfer(0,2,"\x05\x00")(result,channel,spi_len,spi_response(0:3))
          call hardware_io_port_config_direction(1,$3)

          # Check if max retries have been reached
          if (retry_counter < max_retries) then
             # Increase retry counter
             retry_counter = retry_counter + 1
          else
             # Could not talk to the flash : Report error core 0x90
             call attributes_user_write_response(curr_connection, $90)
          end if
          # Flash was not ready - check again later
        if spi_response(1:1) & 1 then
              call hardware_set_soft_timer(6000,0,1)
          else
             # Flash was ready, send response to the remote device (DFU application)
             call attributes_user_write_response(curr_connection, 0)
          end if
      end if
end
```

The final event handler simple makes the device discoverable and connectable in case of a disconnection.

```
# Disconnection event handler
# Makes the device visible and connectable
event connection_disconnected(handle,result)
      # in case if disconnect, return to advertisement mode
      call gap_set_mode(gap_general_discoverable,gap_undirected_connectable)
end
```

3.8 Compiling and Installing the Firmware

3.8.1 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 (or the development kir) 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 **BLE112-project.bgproj**)
7. Press **Update**

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

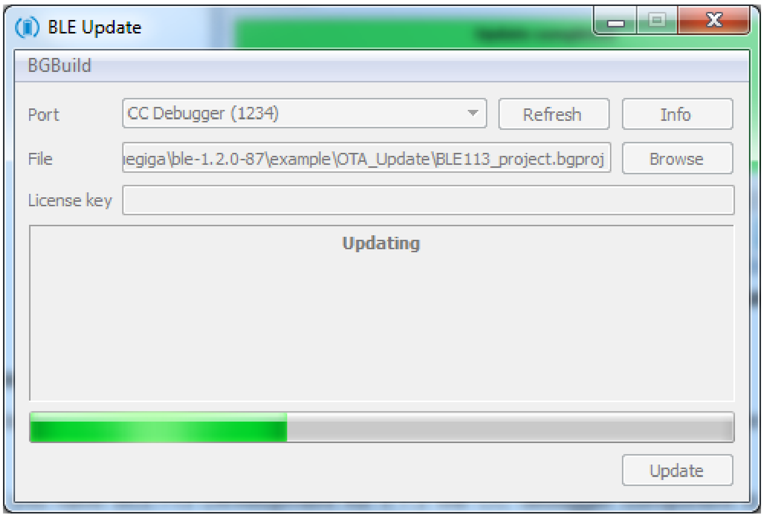

Figure 14: Compile and install with BLE Update tool

Note:

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

If you have BLE112 or BLE113 Development Kit v.1.2 with the on-board CC Debugger, do the following:

- 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.

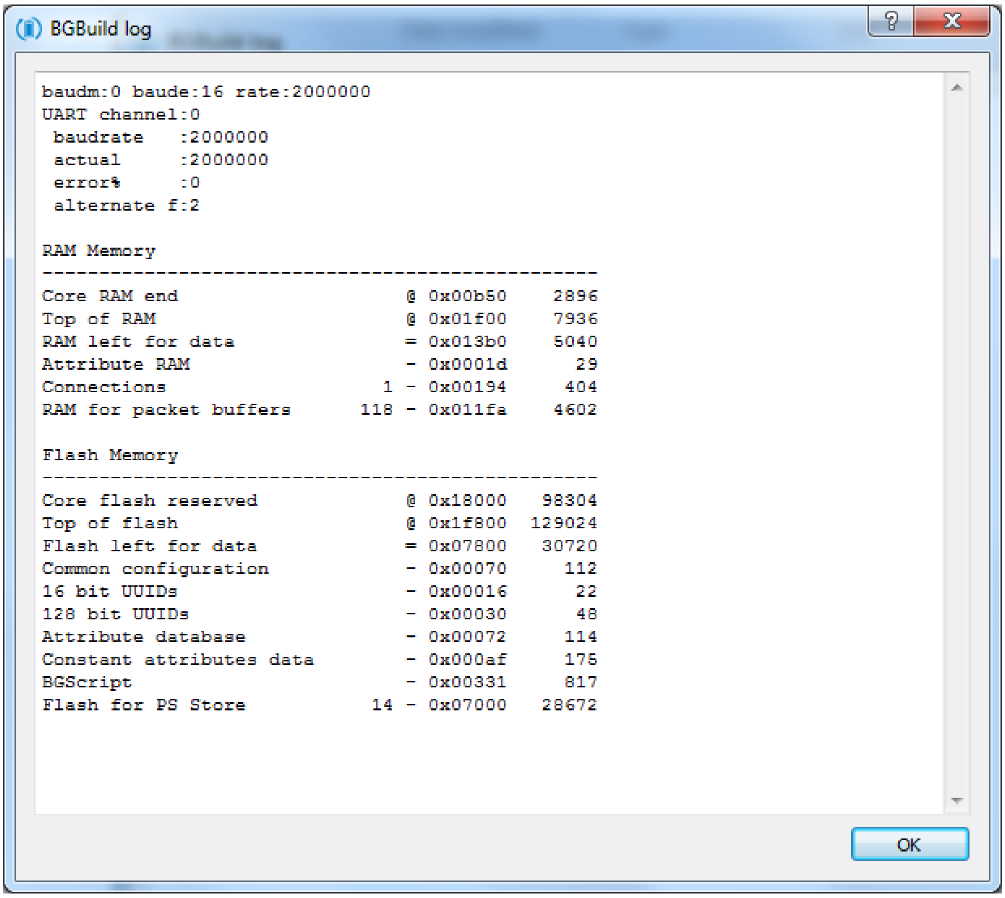

Figure 15: BLE Update build log

3.8.2 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>*

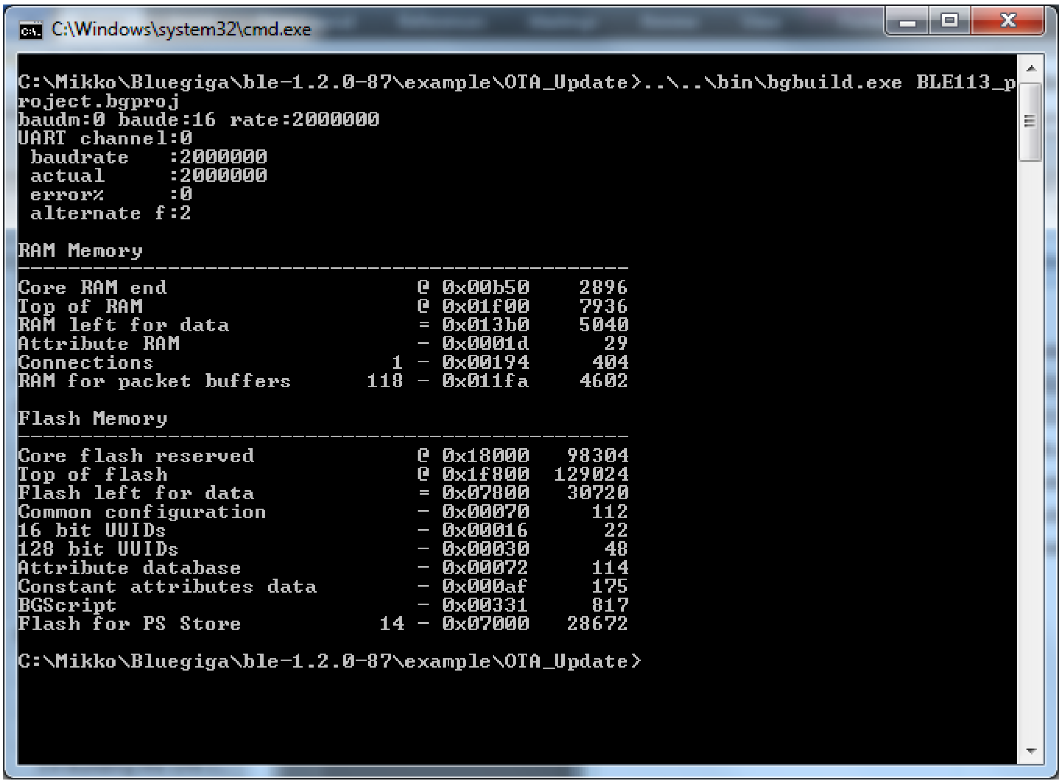

Figure 16: Compiling with BGBuild.exe

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.

4 Testing the OTA Update with BLEGUI

4.1 Using BLEGUI

This section describes how to test the OTA update example 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 responses and has a simple user interface to display device data.

4.1.1 Discovering the OTA Device

- 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 OTA example device 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 OTA device is powered on and the OTA example application is installed to is you should see the device in the BLEGUI software.

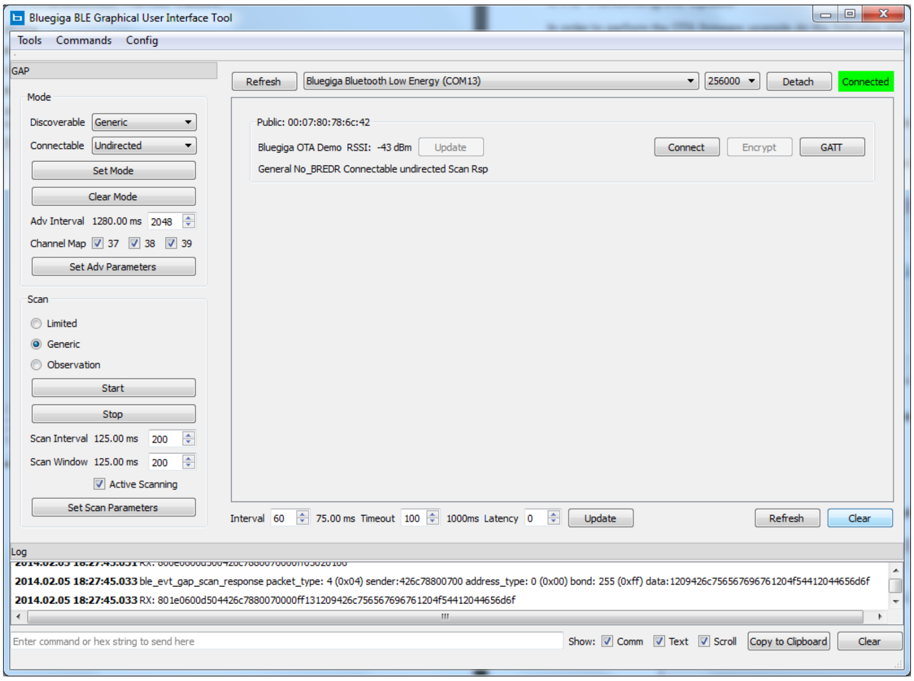

Figure 17: Discovering the OTA device

4.1.2 Checking the OTA Characteristic Handle Values

- **Connect** the OTA device
- Perform a GATT **service discovery**
- Select the OTA service (UUID: 1d14d6ee-fd63-4fa1-bfa4-8f47b42119f0)
- Perform a **descriptors discovery**
- Note the characteristic handle values for the **OTA Control Point Attribute** and **OTA Data Attribute** (15 and 18 in the example application)

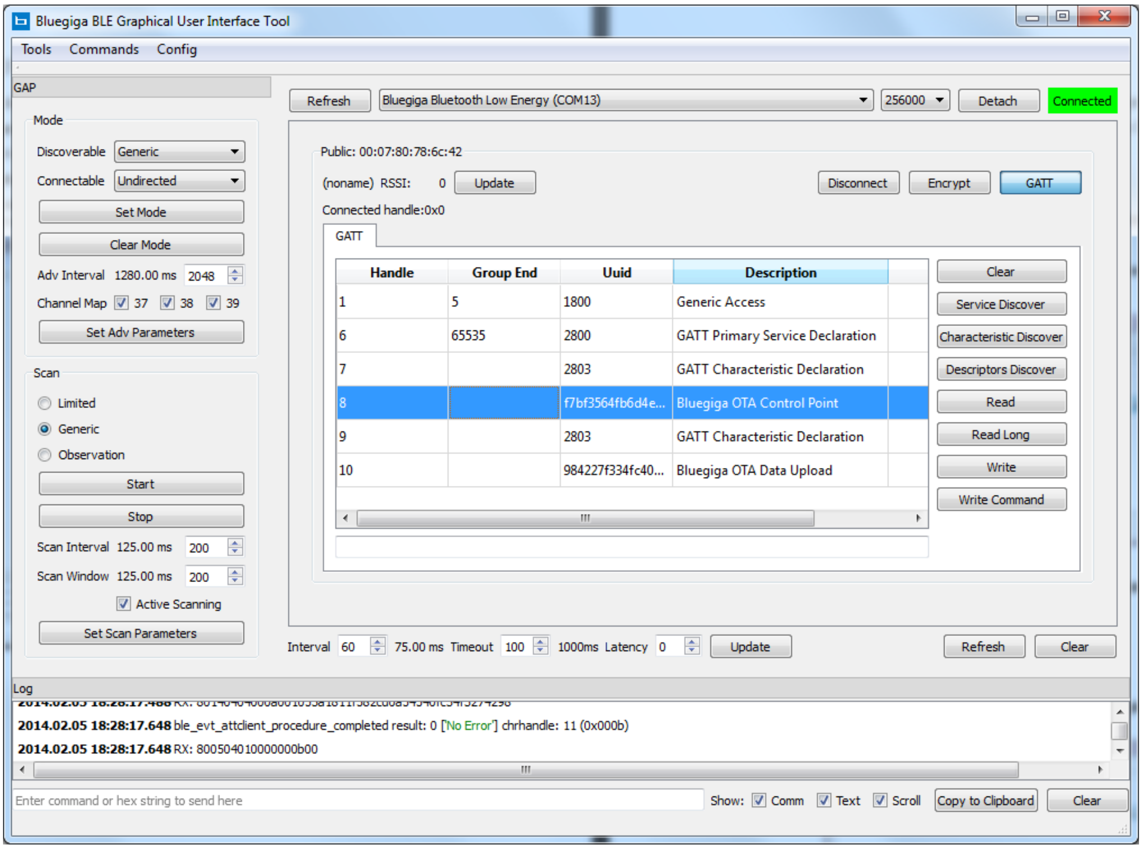

Figure 18: OTA characteristic handle values

4.1.3 Performing the Update

In order to perform the OTA firmware upgrade do the following steps

- Go to **Commands -> DFU -> Over the Air** Upgrade menu
- Select the desired firmware file (.OTA file)
- Select the connection handle of the device you want to update
    - Double check that the connection exists
- Write the **OTA Control Point Attribute** and **OTA Data Attribute handles** to the dialog
- Press **Upload**

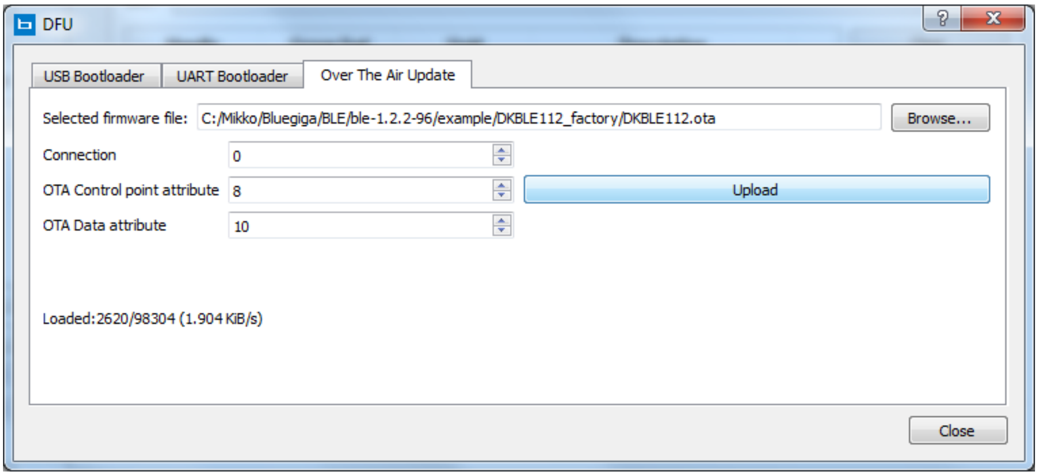

Figure 19: Performing OTA Update

4.1.4 Verifying the Update

Wait for the update the finish and verify you see a message **OTA Completed** message in the dialog and not an error message.

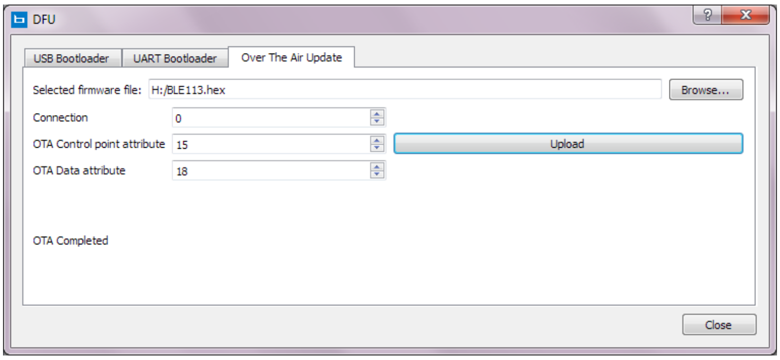

Figure 20: Successful OTA Firmware Update

5 Current Consumption

The average current consumption of BLE113 during OTA firmware update with slow clock disabled is 10.5 mA. The peak current is 27 mA. This assumes maximum TX power and the use of external DC/DC converter.

6 Contact information

Sales:	sales@bluegiga.com
Technical support:	www.bluegiga.com/support/
Orders:	orders@bluegiga.com
WWW:	www.bluegiga.com www.bluegiga.hk

Head Office / Finland:

Phone: +358-9-4355 060
Fax: +358-9-4355 0660
Sinikalliontie 5A
02630 ESPOO
FINLAND

Postal address / Finland:

P.O. BOX 120
02631 ESPOO
FINLAND

Sales Office / USA:

Phone: +1 770 291 2181
Fax: +1 770 291 2183
Bluegiga Technologies, Inc.
3235 Satellite Boulevard, Building 400, Suite 300
Duluth, GA, 30096, USA

Sales Office / Hong-Kong:

Elite Business Center
15/F, Millenium City 3
370 Kwun Tong Road
Kwun Tong
Kowloon
Hong Kong