AN982: *BLUETOOTH®* LOW ENERGY SOFTWARE GLUCOSE SENSOR

APPLICATION NOTE

Wednesday, 02 December 2020

Version 1.2

VERSION HISTORY

Version	Comment
1.0	First version
1.1	Minor updates
1.2	Renamed "Bluetooth Smart" to "Bluetooth Low Energy" according to the official Bluetooth SIG nomenclature

TABLE OF CONTENTS

1 **Introduction** ... 4
2 **What is *Bluetooth* Low Energy Technology?** ... 5
3 Introduction to the Bluegiga *Bluetooth* Low Energy Software ... 6
3.1 The *Bluetooth* Low Energy Stack ... 6
3.2 The *Bluetooth* Low Energy SDK ... 6
3.3 The BGAPI Protocol ... 8
3.4 The BGLib Host Library ... 9
3.5 BGScript™ Scripting Language ... 10
3.6 The Profile Toolkit ... 11
4 **Glucose profile** ... 12
4.1 Description ... 12
4.2 GATT Server: Service requirements ... 13
4.3 GATT Server: Attribute requirements ... 13
4.4 Recommended connection establishment procedures ... 14
4.5 Security requirements ... 14
5 **Implementing a Glucose Sensor** ... 15
5.1 Creating a project (**project.xml**) ... 16
5.2 Hardware configuration (**hardware.xml**) ... 17
5.3 GATT database for Glucose Sensor (**gatt.xml**) ... 18
5.4 Application Configuration (**config.xml**) ... 22
5.5 BGScript for Glucose Sensor (**glucose_sensor.bgs**) ... 23
5.6 Compiling and Installing the Firmware ... 28
5.7 Testing the Glucose Sensor ... 32
5.8 Testing with BLEGUI software ... 32
5.9 Testing with iPhone or iPad ... 38
6 **External resources** ... 42

1 Introduction

This application note discusses how to build a *Bluetooth* 4.0 glucose profile sensor using Bluegiga's *Bluetooth* 4.0 software development kit, for use with a DKBLE112 hardware evaluation board and an Apple iPhone or iPad, or other *Bluetooth* Low Energy device capable of acting as a glucose collector. The application note contains a practical example of how to build a GATT-based Glucose Profile and how to make a basic glucose sensor device using BGScript scripting language, including authenticated bonding (encryption) and on-module glucose record storage and retrieval.

Various other features of the development kit are also demonstrated in this project for the sake of instruction, such as SPI communication to the on-board LCD, potentiometer ADC readings to simulate glucose levels, and UART debug data output.

Note that the glucose profile as implemented is an official profile standardized by the *Bluetooth* SIG.

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 Introduction to the Bluegiga *Bluetooth* Low Energy Software

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

- The *Bluetooth* Low Energy Stack
- The *Bluetooth* Low Energy Software Development Kit (SDK)

3.1 The *Bluetooth* Low Energy Stack

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

3.2 The *Bluetooth* Low Energy SDK

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

The *Bluetooth* Low Energy 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* Low Energy 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* Low Energy Software in order to easily develop the *Bluetooth* Low Energy compatible end products.

Bluegiga's *Bluetooth Low Energy Software* provides a complete development framework for *Bluetooth* Low Energy application implementers.

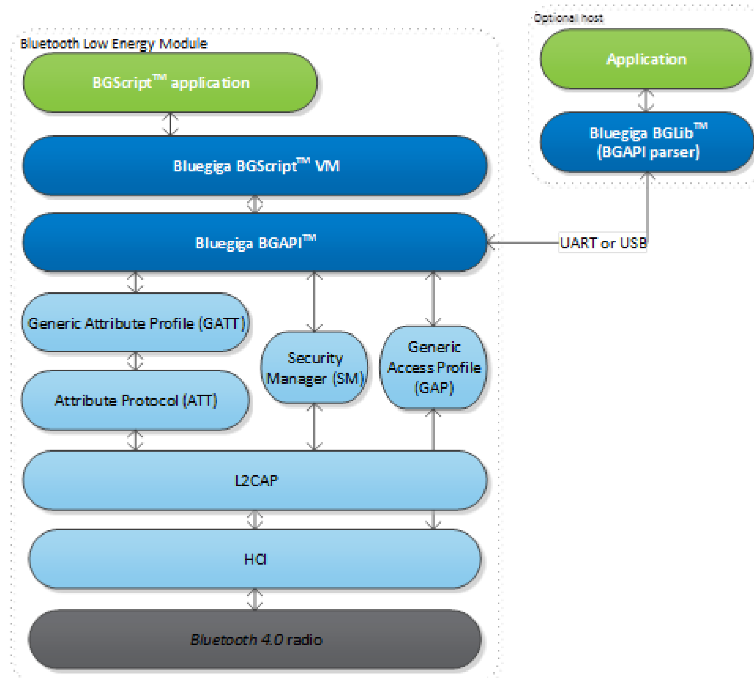


Figure 1: *Bluetooth Low Energy* Software

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

- The *Bluetooth* Low Energy stack implementing the *Bluetooth* Low Energy protocol
- **BGAPI™** APIs that enable the software developers to interface to the *Bluetooth* Low Energy Stack
- **BGScript™** Virtual Machine (VM) and scripting language which enable application code to be developed and executed directly on the *Bluetooth* Low Energy 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* Low Energy 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* Low Energy profiles, services and characteristics using simple XML templates

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

3.3 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* Low Energy hardware) or USB interfaces.

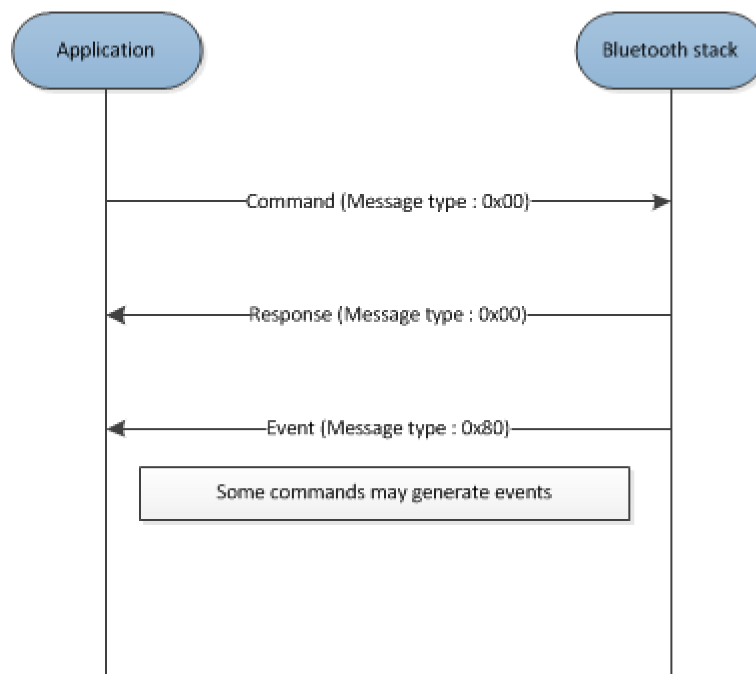


Figure 2: BGAPI protocol

The BGAPI provides access for example to the following layers in the *Bluetooth* Low Energy 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

3.4 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* Low Energy SDK. The purpose is to simplify the application development to various host environments.

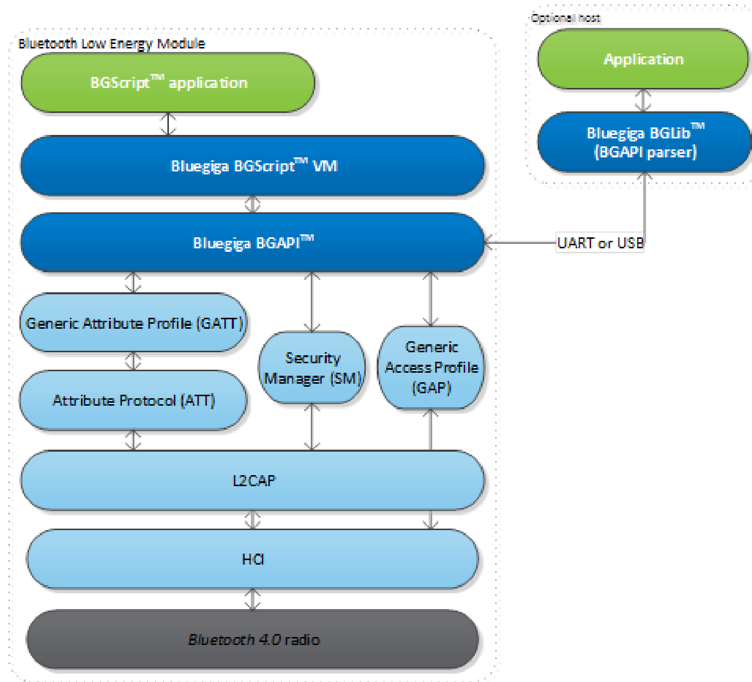


Figure 3: BGLib host library

3.5 BGScript™ Scripting Language

The *Bluetooth* Low Energy 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* Low Energy Hardware. The *Bluetooth* Low Energy modules can run simple applications along the *Bluetooth* Low Energy stack and this provides a benefit when one needs to minimize the end product's size, cost and current consumption. For developing standalone *Bluetooth* Low Energy 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.

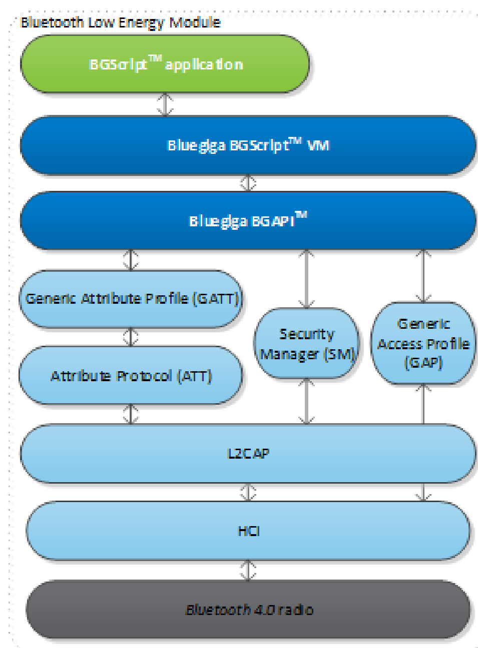


Figure 4: BGScript application model

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
```

3.6 The Profile Toolkit

The *Bluetooth* Low Energy profile toolkit a simple set of tools, which can used to describe GATT based *Bluetooth* Low Energy 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
<?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

4 Glucose profile

4.1 Description

The **Glucose Profile** enables a device to connect and interact with a glucose sensor for use in consumer and professional healthcare applications. The full Bluetooth SIG specification for the Glucose Profile is available in PDF form here:

https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=248025

An organized table structure of the Glucose profile as defined by the Bluetooth SIG is available here:

http://developer.bluetooth.org/gatt/profiles/Pages/ProfileViewer.aspx?u=org.bluetooth.profile.glucose.xml

(Other links present in this document to services and characteristics also go to the Bluetooth SIG service browser online.)

The Glucose Profile defines two roles:

1. **Glucose Sensor**

The sensor is the devices which has the actual glucose measurement device and provides the GATT structure and data storage for use by a collector (see below). A glucose sensor must include at least a partial implementation of the **Device Information** service as well as the sensor portion of the **Glucose** service, as defined by the Bluetooth SIG.

2. **Collector**

The collector is the device which gathers glucose data from the sensor. In this example, we will do partial testuse an iOS application provided by Nordic Semiconductor for a comprehensive GUI-based collector. This application note does not discuss the implementation of a collector itself.

The figure below shows the relationship of these two roles.

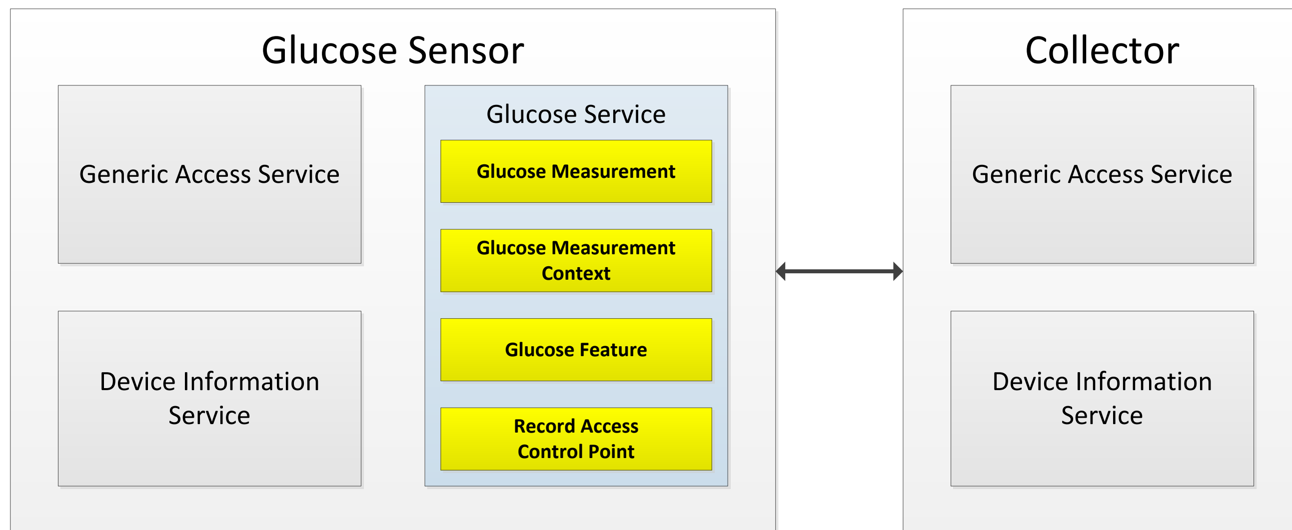


Figure 6: Glucose Profile roles

4.2 GATT Server: Service requirements

The table below describes the service requirements.

Service	UUID	Glucose Sensor
GAP service	1800	Mandatory
Glucose service	1808	Mandatory
Device Information service	180A	Mandatory

Table 1: Service requirements

4.3 GATT Server: Attribute requirements

The table below describes the structure and requirements for the attribute used in the Glucose Service. (Links go to the Bluetooth SIG online characteristic definition browser for that attribute)

Characteristic	UUID	Length	Type	Support	Security	Properties
Glucose Measurement	2A18	Variable (max 17B)	Hex	Mandatory	None	Notify
Glucose Measurement Context	2A34	Variable (max 17B)	Hex	Optional	None	Notify
Glucose Feature	2A51	2 bytes	Hex	Mandatory	None	Read
Record Access Control Point	2A52	Variable (typical 2B)	Hex	Mandatory	Writeable with Authentication	Write, Indicate

Table 2: Glucose Service structure (sensor only)

- Each **UUID** in this service is an official, adopted 16-bit ID for the characteristic
- The Glucose Measurement and Context **length** is variable depending on the features supported by the sensor. This demo emulates all specified features (though most data is arbitrarily chosen only for demo purposes). and has a length of 17 bytes. This fits within the maximum payload size for indicated values (20 bytes).
- Security is not necessary to connect to a glucose sensor in order to read a single measurement taken during connection, but it is required to access historical records which may be stored on the device. These records are only accessible through the **Record Access Control Point**. In order to write control commands to this attribute, the collector must **bond** (a.k.a. **pair**) with the sensor. A collector which connects but does not pair will not be allowed to write any values to the control point, and therefore will not be able to access any stored records.
- **Glucose Measurement** and the optional **Glucose Measurement Context** cannot be directly read, but may only be pushed (via **notifications**) from the sensor to the collector, after the collector enables client notifications for those services. **Glucose Feature** contains a constant value describing the feature set of the sensor device, and may be read as desired by the collector.

4.4 Recommended connection establishment procedures

4.4.1 Un-bonded devices

Advertisement duration	Parameter	Value
First 30 seconds (fast connection)	Advertising interval	20ms to 30ms
After 30 seconds (reduced power)	Advertising interval	1000ms to 2500ms

Table 3: Advertising parameters for un-bonded Glucose Sensor

- The Glucose Sensor should use the GAP Limited Discoverable Mode with connectable undirected advertising events when establishing an initial connection. *(For simplicity, this demo implementation uses GAP General Discoverable Mode.)*
- If the connection is not established within a time limit, the sensor may exit GAP Connectable mode. *(For simplicity, this demo implementation does not ever exit GAP Connectable mode.)*

4.4.2 Bonded devices

The following produce is uses for bonded devices:

- A Glucose Sensor shall enter the GAP Undirected Connectable Mode either when commanded by the user to initiate a connection to a Collector or when a Glucose Sensor has one or more stored records to send to a previously connected Collector.
- The Glucose Sensor should write the address of the target Collector in its White List and set its controller advertising filter policy to 'process scan and connection requests only from devices in the White List'. The advertisement parameters should be as in Table 3.
- If the connection is not established within a time limit, the sensor may exit GAP Connectable mode.

4.4.3 Link loss procedure

When connection is terminated due to link loss the sensor should attempt reconnection with the Collector by entering the GAP connectable mode using the recommended parameters from Table 3.

4.5 Security requirements

The Glucose Sensor shall bond with the Collector.

When bonding is used:

1. All supported characteristics specified by the Glucose Service shall be set to Security Mode 1 and either Security Level 2 or 3.
2. The Glucose Sensor shall use the SM Slave Security Request procedure to inform the Collector of its security requirements.
3. All characteristics specified by the Device Information Service that are relevant to this profile should be set to the same security mode and level as the characteristics in the Glucose Service.

5 Implementing a Glucose Sensor

The chapter contains step by step instructions how to implement a stand-alone Glucose Sensor with Bluegiga's *Bluetooth* 4.0 Software Development Kit. The chapter is split into following steps:

1. Creating a project
2. Defining hardware configuration
3. Building **Glucose** and **Device Information** Services with Profile Toolkit
4. Writing BGScript source code
5. Compiling the GATT database and BGScript into binary firmware
6. Installing the firmware into BLE112 or DKBLE112 hardware

The actual project comes as an example with the Bluegiga's *Bluetooth* Low Energy Software Development Kit v.1.1.1 or newer under the **`\example\glucose_sensor\`** directory.

5.1 Creating a project (**project.xml**)

The Glucose Sensor implementation is started by first creating a project file (**project.xml**), which defines the resources use by the project and the firmware output file. This project file includes a description of each of the main elements of the project.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project>
    <gatt in="gatt.xml" />
    <hardware in="hardware.xml" />
    <script in="glucose_demo.bgs" />
    <config in="config.xml" />
    <image out="out.hex" />
</project>
```

Figure 7: Project file (project.xml)

- **<gatt>** Defines the XML file containing the GATT database.
- **<hardware>** Defines the XML file containing the hardware configuration.
- **<script>** Defines the BGScript file which contains the BGScript code.
- **<config>** Defines the application configuration file.
- **<image>** Defines the output HEX file containing the firmware image.

5.2 Hardware configuration (**hardware.xml**)

The **hardware.xml** file contains the hardware configuration for BLE112 device. It describes which interfaces and functions are used and what their specific properties are.

```
<?xml version="1.0" encoding="UTF-8" ?>
<hardware>
    <sleeposc enable="true" ppm="30" />
    <usb enable="false" />
    <txpower power="15" bias="5" />
    <script enable="true" />
    <pmux regulator_pin="7" />
    <usart channel="0" mode="spi_master" alternate="2" polarity="positive" phase="1"
endianness="msb" baud="57600" endpoint="none" />
    <usart channel="1" alternate="1" baud="115200" endpoint="none" flow="false" />
    <port index="0" pull="down" />
</hardware>
```

Figure 8: Hardware configuration for Glucose Sensor

- **<sleeposc>** The 32.768KHz sleep oscillator is enabled. Sleep oscillator allows the device to enter power mode 1 or 2 between *Bluetooth* operations, for example between connection intervals. This should always be used.
- **<usb>** USB interface is disabled to save power and allow the device to go to low-power modes. If USB is enabled, no low-power modes will be used.
- **<txpower>** TX power is set to +3dBm value. Every step represents roughly a 1dBm change and the range of the parameter is 15 to 0, corresponding TX power values from +3dBm to -24dBm.
- **<script>** Scripting is enabled as the Glucose Sensor application is implemented with with the BGScript scripting language.
- **<pmux>** Enables automatic management of the DC/DC converter on the DKBLE112. This prevents momentarily large current draws from the CR2032 battery during transmissions, if battery power is used, and will extend the life of the battery.
- **<usart channel=”0”>** Enables one USART interface used for SPI data communications. In this configuration, USART 0 is used in alternate configuration 2, which allows communication with the SPI-based LCD on the DKBLE112.
- **<usart channel=”1”>** Enables the second USART interface used for UART data communications. In this configuration, USART 1 is used in alternate configuration 1 and with 115200 bps baud rate. RTS/CTS flow control is **disabled**, since this allows the BLE112 to send data regardless of whether anything is connected to the port, which prevents the buffer from filling up and potentially locking the module.
- **<port>** Configures Port 0 pins with pull-down (for later use with GPIO interrupts).

The example is designed to work with the DKBLE112 development kit in the default configuration, so it can be easily tested with the DKBLE112 and an iPhone running the Nordic Semiconductor **nRF Ready** app.

5.3 GATT database for Glucose Sensor (**gatt.xml**)

This section describes how to define the **Glucose** Profile's services using Bluegiga's Profile Toolkit.

The **Glucose** Profile contains three services:

1. **Generic Access** Profile (GAP) service
2. **Device Information** service
3. **Glucose** service

Optionally, if the application requires it, other services can be implanted, such as the **Battery Status** service. This demo project implements the **Battery Status** service as well.

5.3.1 Generic Access Profile (GAP) service

Every *Bluetooth* Low Energy 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.

The service has two characteristics, which are explained in Table 4. In this example the characteristics are read-only, so they are also marked as **const.** Constant values are stored on the flash of BLE112 and the value is defined in the GATT database. Constant values cannot be changed.

```
<!-- 1800: org.bluetooth.service.generic_access -->
<service uuid="1800" id="generic_access">
    <description>Generic Access</description>

    <!-- 2A00: org.bluetooth.characteristic.gap.device_name -->
    <characteristic uuid="2A00" id="c_device_name">
        <description>Device Name</description>
        <!-- glucose profile v1.0 optional spec: device_name is writable, not enabled here -->
        <properties read="true" const="true" />
        <!-- It's a good idea to keep this <= 19 characters, for proper display on iOS -->
        <value>BGT Glucose Demo</value>
    </characteristic>

    <!-- 2A01: org.bluetooth.characteristic.gap.appearance -->
    <characteristic uuid="2A01" id="c_appearance">
        <description>Appearance</description>
        <properties read="true" const="true" />
        <!-- 1024: Generic Glucose Meter, Generic category -->
        <value type="hex">0400</value>
     </characteristic>

</service>
```

Figure 9: GAP service

Characteristic	UUID	Type	Support	Security	Properties
Device name	2A00	UTF8	Mandatory	None	Read (optionally write)
Appearance	2A01	16-bit	Mandatory	None	Read

Table 4: GAP service description

If the device name needs to be changeable by the remote device, then the write property should be enabled.

5.3.2 Glucose Service

The sensor is the device implementing the GATT Server, so it must also implement the **Glucose** service.

The **Glucose** service is defined as below:

Characteristic	UUID	Length	Type	Support	Security	Properties
Glucose Measurement	2A18	Variable (max 17B)	Hex	Mandatory	None	Notify
Glucose Measurement Context	2A34	Variable (max 17B)	Hex	Optional	None	Notify
Glucose Feature	2A51	2 bytes	Hex	Mandatory	None	Read
Record Access Control Point	2A52	Variable (typical 2B)	Hex	Mandatory	Writeable with Authentication	Write, Indicate

Table 5: Glucose Service description

The **Glucose** Service is created by adding the code below to the **gatt.xml** file:

```
<!-- 1808: org.bluetooth.service.glucose -->
<service uuid="1808" advertise="true">
    <description>Glucose Service</description>
    <characteristic uuid="2A18" id="c_glucose_measurement">
        <description>Glucose Measurement</description>
        <properties notify="true" />
        <value length="17" variable="true" />
    </characteristic>
    <characteristic uuid="2A34" id="c_glucose_measurement_context">
        <description>Glucose Measurement Context</description>
        <properties notify="true" />
        <value length="17" variable="true" />
    </characteristic>
    <characteristic uuid="2A51" id="c_glucose_feature">
        <description>Glucose Feature</description>
        <properties read="true" const="true" />
        <value length="2" type="hex">07FF</value>
    </characteristic>
    <characteristic uuid="2A52" id="c_record_access_control_point">
        <description>Record Access Control Point</description>
        <properties indicate="true" write="true" authenticated_write="true" />
        <value length="17" variable="true" />
    </characteristic>

</service>
```

Figure 10: Glucose Service (sensor only)

The **Glucose** service is explained below:

- First, the **advertise="true"** option is needed for the for the **<service>** tag. When the sensor advertises, the UUID of the glucose service (**0x1808**) is included in the data field of the advertisement packets, so the device can be easily identified by the demote devices. Many devices (including iPhones/iPads) often filter scans based on UUID, so if this is not included in the advertisement packet, your device will not be seen by the collector.
- Each attribute is given an **id** field (e.g. "**c_glucose_measurement**"), which is used in the included BGScript code for a named reference to the numeric attribute handle. Numeric handles are assigned

during the compile process based on the structure of the GATT database, so it is helpful to use named references instead so that you do not need to know the handles beforehand. There is no official naming convention requirement for these IDs. The only requirement is that they must be alphanumeric (letters, numbers, and underscore characters only), and they must not overlap with any BGScript keywords.

- **Read**, **write**, and **notify** properties are enabled on the various attributes as required by the service specification. Using **notify** means that the collector will be able to “subscribe” (enable notifications) to this attribute, and then any value updates done by the sensor will be automatically pushed out to the collector. **Indications** and **notifications** are the same except that **indications** are acknowledged by the remote end, while **notifications** are not (similar to the difference between TCP and UDP within the IP network protocol). Note that on the “**c_record_access_control_point**” attribute, both **write** and **authenticated_write** are present. This is required; you cannot use simply **authenticated_write** without also enabling **write**.

5.3.3 Summary

The full GATT database implementation is shown below. The source code in the demo project has additional comments which explain the specification requirements and actual values used in much more detail.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <service uuid="1800" id="generic_access">
        <description>Generic Access</description>
        <characteristic uuid="2A00" id="c_device_name">
            <description>Device Name</description>
            <properties read="true" const="true" />
            <value>BGT Glucose Demo</value>
        </characteristic>
        <characteristic uuid="2A01" id="c_appearance">
            <description>Appearance</description>
            <properties read="true" const="true" />
            <value type="hex">0400</value>
        </characteristic>
    </service>
    <service uuid="180A" id="device_information">
        <description>Device Information</description>
        <characteristic uuid="2A29" id="c_manufacturer_name">
            <description>Manufacturer Name</description>
            <properties read="true" const="true" />
            <value>Bluegiga</value>
        </characteristic>
        <characteristic uuid="2A24" id="c_model_number">
            <description>Model Number</description>
            <properties read="true" const="true" />
            <value>BG-BLE-GLUCOSE</value>
        </characteristic>
        <characteristic uuid="2A25" id="c_serial_number">
            <description>Serial Number</description>
            <properties read="true" const="true" />
            <value>123456789</value>
        </characteristic>
        <characteristic uuid="2A27" id="c_hardware_revision_string">
            <description>Hardware Revision String</description>
            <properties read="true" const="true" />
            <value>H1.0.0</value>
        </characteristic>
        <characteristic uuid="2A26" id="c_firmware_revision_string">
            <description>Firmware Revision String</description>
            <properties read="true" const="true" />
            <value>F1.0.0</value>
        </characteristic>
        <characteristic uuid="2A28" id="c_software_revision_string">
            <description>Software Revision String</description>
            <properties read="true" const="true" />
            <value>S1.0.0</value>
        </characteristic>
        <characteristic uuid="2A23" id="c_system_id">
            <description>System ID</description>
            <properties read="true" const="true" />
            <value type="hex">112233FFFE778899</value>
        </characteristic>
    </service>
    <service uuid="1808" advertise="true">
        <description>Glucose Service</description>
        <characteristic uuid="2A18" id="c_glucose_measurement">
            <description>Glucose Measurement</description>
            <properties notify="true" />
            <value length="17" variable="true" />
        </characteristic>
        <characteristic uuid="2A34" id="c_glucose_measurement_context">
            <description>Glucose Measurement Context</description>
            <properties notify="true" />
            <value length="17" variable="true" />
        </characteristic>
        <characteristic uuid="2A51" id="c_glucose_feature">
            <description>Glucose Feature</description>
            <properties read="true" const="true" />
            <value length="2" type="hex">07FF</value>
        </characteristic>
        <characteristic uuid="2A52" id="c_record_access_control_point">
            <description>Record Access Control Point</description>
            <properties indicate="true" write="true" authenticated_write="true" />
            <value length="17" variable="true" />
        </characteristic>
    </service>
    <service uuid="180F" id="battery_service">
        <description>Battery Service</description>
        <characteristic uuid="2A19" id="c_battery_level">
            <description>Battery Level</description>
            <properties read="true" notify="true" />
            <value length="1" type="hex" />
        </characteristic>
    </service>
</configuration>
```

Figure 11: Glucose Sensor Profile GATT database

5.4 Application Configuration (**config.xml**)

The **config.xml** file contains the application configuration for BLE112 device. This file is not mandatory for your project since all of the default values are typically acceptable, but it is useful for specific requirements such as script timeout control (as below) or UART optimization settings.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<config>
    <script_timeout value="0" />
</config>
```

Figure 7: Application configuration for Glucose Sensor

- **<script_timeout>** Disables script timeout. This generally not advisable without a specific reason, but in this implementation, accessing the stored records through the **Record Access Control Point** attribute can take more than the default number of allowed script operations to complete, especially if there are a large number of stored records. This is therefore used to ensure that all stored records may be retrieved without issue. The default value (1000) is enough for only about 10 records.

The other settings available in **config.xml** are not necessary for this demo.

5.5 BGScript for Glucose Sensor (**glucose_sensor.bgs**)

The example implements a standalone glucose sensor device where no external host processor is needed. The Glucose sensor application is created as a BGScript script application and the BGScript code is explained in this chapter.

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.

The glucose sensor BGScript application uses the following event listeners:

5.5.1 System: Boot event (**system_boot**)

When the system is started or reset, a **system_boot** event is generated. This event listener should be the entry point for all the BGScript applications, and provides a perfect opportunity for initializing any required variables.

In the glucose sensor demo, the following tasks take place in the **system_boot** event handler:

- Initialize status tracking variables and tick counter
- Set up GPIO interrupts for catching button presses
- Enable bonding
- Begin advertising
- Start a 1-second continuous timer
- Load information from public store (PS) keys concerning stored records
- Initialize battery level characteristic to 95% (testing value)
- Output debug boot data to UART
- Initialize DKBLE112 LCD and display status

The actual project source code contains this code along with many detailed comments.

5.5.2 *Bluetooth*: Connection event (**connection_status**)

When the *Bluetooth* connection is established a connection event occurs. For this purpose, an event listener is added to the BGScript code which tracks the connection status, sends debug data out the UART port, and updates the LCD appropriately.

Note: because **bonding** is used, when a collector bonds with the sensor, a second **connection_status** event will be fired when the connection becomes ecrypted. If/when this occurs using this demo, the LCD will be updated again to show “Encrypted” instead of simply “Connected”. Remember that you will not be able to access stored records using the Record Access Control Point if the connection is not encrypted.

5.5.3 *Bluetooth*: Disconnection event (**connection_disconnected**).

If the *Bluetooth* connection is a lost, a disconnection event occurs. For this purpose, an event listener is added to the BGScript and it does almost the same as the boot event listener; it sets the application state and restarts the advertisement procedure.

Note: a Bluetooth *Low Energy* device will not automatically resume advertising when a connection is lost. It will be put into an **idle** state. If you want the device to resume advertising (typically desirable), you must explicitly do this.

5.5.4 Data: Receiving control data from the remote device (**attributes_value**)

The ATTribute protocol is used to transmit data over a *Bluetooth* connection. A remote device can use an ATT write operation to write up to 20 bytes of data. With the glucose sensor demo, only one attribute is writable, the "Record Access Control Point." Therefore, in this project, the **attributes_value** event will only occur then if the connection is encrypted (since **authenticated write** is required for this attribute).

The BGSciprt code checks to make sure the attribute handle matches the **c_record_access_control_point** value first (not technically necessary in this case since only one attribute is writable, but very good practice), and then parses the value written. This attribute is used for sending commands such as "**Report stored records**" or "**Delete stored records**." The glucose profile describes many possible commands, but for this demo, only the "**Report stored records**" command is implemented (opcode value = 0x01).

According to the profile specification, the first byte of the value written to the attribute is the opcode value, and the second byte is the operator. When the "**Report stored records**" opcode is used, the operator byte controls which records should be reported. For this deom, only the "**All records**" (operator = 0x01), "**First record**" (operator = 0x05), and "**Last record**" (operator = 0x06) are implemented, since these are the only operations implemented in the nRF demo iOS app.

In this project, records are stored in a ring buffer implemented using the PS key storage space provided right on the BLE112 module itself. There are 128 key slots available, each holding up to 32 bytes. The first key is available at address 0x8000, and the last one at 0x807F. The last slot (0x807F) is used for storing the configuration data and stored record count, leaving 127 more slots. The first record is stored in 0x8000, the second in 0x8001, and so on. When 0x807E is reached, the next record will be stored on 0x8000. The ring buffer is maintained using a "head" index variable and a "record count" variable.

Each glucose reading generates a 17-byte **measurement** value and a corresponding 16-byte **measurement context** value. This is a total of 33 bytes, which is too big to fit in a single 32-byte PS key. Therefore, the final two bytes of the measurement value (which is the "**status**" subfield of the value) are not stored. In an ideal implementation, this would not be necessary.

The First/Last/All buttons in the nRF demo app each write a value to this attribute, and the resulting operation is done inside the **attributes_write** event handler. This involves reading the requested record(s) from the PS key storage area, then writing the **measurement** and **measurement context** values to the local GATT attributes, which subsequently pushes them to the connected client using notifications.

For detail on exactly how this is done, refer to the project source code and comments.

5.5.5 IO: Detecting button presses (**hardware_io_port_status**)

For controlling the behavior of the sensor, this project makes use of two buttons on the DKBLE112 board.

- **P0_0** is used to trigger a new glucose reading (using the ADC value read through the potentiometer also on the DKBLE112).
- **P0_1** is used to reset all stored records and bonding information, if present.

Pressing the buttons on the DKBLE112 momentarily brings those lines to a logic HIGH state. These signals are configured to generate interrupts by the following line back in the **system_boot** event handler:

```
call hardware_io_port_config_irq(0, 3, 0)
```

The first parameter specifies the port, the second is the bitmask for which pins to enable interrupts on, and the third is the rising or falling edge setting. Interrupts can currently only be enabled on Port 0 and Port 1. The parameters used in the command above are as follows:

0 = Port 0

3 = *0b*00000011, bits 0 and 1 are set, so interrupts enables on Pin 0 and Pin 1 of Port 0

0 = Rising edge

Remember that in **hardware.xml**, we also configured Port 0 to be pulled down, so that the rising edge will be more reliably detected. The I/O signals are not de-bounced in this demo.

The **P0_0** button press initiates an ADC read, which triggered immediately but takes a few milliseconds to complete. Therefore, the ADC read generates another event (**hardware_adc_result**) which we also catch. Although we only use one simultaneous or consecutive ADC read operation in this project, note that multiple ADC reads must be cascaded such that the second is not triggered until the first completes.

The **P0_1** button press manually clears all bonding entry storage slots, sets the stored record count to zero, and moves the record storage ring buffer head index back to the beginning. It does not actually *erase* the PS key values, since this is not necessary.

5.5.6 Timer: 1-second clock tick (**hardware_soft_timer**)

For basic 1-second "tick" tracking, we enabled the soft timer back in the **system_boot** event handler with the following line:

```
call hardware_set_soft_timer(32768, 0, 0)
```

The first parameter specifies the interval relative to the main oscillator, which is 32768 Hz. The timer ticks every (*value* / 32768) seconds, which in this case is exactly 1 second. The accuracy of this timer is +/- 30ppm, so it is suitable for basic tick tracking of this kind.

The second parameter is the assigned timer **handle** (should always be "0" in the v1.1 SDK), and the third is whether it is **repeating** (0) or **single shot** (1). We want it to fire every second, so it is configured to be repeating.

Inside the timer event handler, we mainly do simple animation on the SPI LCD on the DKBLE112 to indicate activity or revert temporary status messages back to the default "Advertising" or "Connected" or "Encrypted" message. While advertising, a single "?" character will blink on and off once per second. While connected (or encrypted), it will alternate between the "+" and "*" characters.

This event handler also triggers a battery level ADC reading, which is used to update the **c_battery_status** attribute value when obtained later in the **hardware_adc_result** handler. The internal battery level ADC channel is **15**.

5.5.7 ADC Result: Battery and Glucose measurement (**hardware_adc_result)**

This event is triggered when the previously requested ADC read operation completes. We handle two specific cases in this demo: the battery value, and the potentiometer value used to emulate a glucose concentration reading. A real glucose sensor would use actual hardware capable of detecting the glucose concentration in a blood sample. Since this hardware is not available on our dev kit, we use the potentiometer connected to **P0_6** instead.

The battery level is computed from the ADC read range (0-32768) and scaled to the required 0-100 level, where 0% = 2.0v and 100% = 2.52v (determined empirically with a fresh CR2032). This single byte is written to the **c_battery_level** attribute, which is then notified to or read by the collector as desired.

The glucose measurement and context values are more complex, but built one byte at a time in great detail in the BGScript code according to the glucose service definition. The particular structure of these two values depends on which features are implemented on the glucose sensor itself. For this demo, we emulate all available features that are described by the specification.

The 17-byte **Glucose Measurement** attribute is built with the following data:

Byte(s)	Field Name	Data Type	Demo Value	Notes
0	Flags	8-bit hex	0x1B	Time Offset field enabled Glucose Concentration units are kg/L Type field enabled Sample location field enabled Sensor Status field enabled Context Information enabled *(sent via **Glucose Measurement Context** attribute)*
1:2	Sequence Number	16-bit hex	Variable	Starts at 0, increments for each new reading
3:9	Base Time	7-byte hex	Variable	3:4=year, 5=month, 6=day, 7=hour, 8=minute, 9=second *0xYYYYMMDDHHmmss*
10:11	Time Offset	16-bit hex	Variable	Seconds to offset from **Base Time** value
12:13	Glucose Concentration (kg/L)	16-bit SFLOAT	Variable	Depends on DKBLE112 potentiometer setting
14 [7-4]	Type field	4-bit hex	0x1	0x1 = Capillary Whole Blood
14 [3-0]	Sample Location	4-bit hex	0x1	0x1 = Finger
15:16	Sensor Status	16-bit hex	0x0000	0 indicates no error

Table 6: Glucose Measurement Structure

The 17-byte **Glucose Measurement Context** attribute is built with the following data:

Byte(s)	Field Name	Data Type	Demo Value	Notes
0	Flags	8-bit hex	0x5F	Carbohydrate ID enabled Carbohydrate units are **kg** Meal enabled Tester enabled Health enabled Exercise Duration enabled Exercise Intensity enabled Medication ID enabled Medication units are **kg** HbA1C precise percentage enabled
1:2	Sequence Number	16-bit hex	Variable	Must match corresponding measurement
3	Carbohydrate ID	8-bit hex	0x03	0x03 = Dinner
4:5	Carbohydrate (kg)	16-bit SFLOAT	0x13E1	0xE113 = 2.75 kg in SFLOAT format *(0b1110 0001 0001 0011, little endian)*

6	Meal	8-bit hex	0x02	0x02 = Postprandial
7 [7:4]	Tester	4-bit hex	0x2	0x2 = Health Care Professional
7 [3:0]	Health	4-bit hex	0x4	0x4 = Under Stress
8:9	Exercise Duration	16-bit hex	0x1E00	0x001E = 30 seconds *(little-endian)*
10	Exercise Intensity	8-bit hex	0x64	0x64 = 100% (range = 0-100)
11	Medication ID	8-bit hex	0x02	0x02 = Short acting insulin
12:13	Medication (kg)	16-bit SFLOAT	0x05E0	0xE005 = 0.05 kg in SFLOAT format *(0b1110 0000 0000 0101, little-endian)*
14:15	HbA1c (%)	16-bit SFLOAT	0x1DE2	0xE21D = 5.41% in SFLOAT format *(0b1110 0010 0001 1101, little-endian)*

Table 7: Glucose Measurement Context Structure

These two attributes are built field-by-field in the **hardware_adc_result** event handler. All of the field values in both attributes are static (which they would not be in a real glucose sensor), with the exception of the sequence number, glucose concentration value, and time offset. The time offset field value is set to whatever the sequence number is, which gives the effect that each glucose measurement appears to be take one second after the previous one.

For a detailed look at how this code actually works, refer to the “**glucose_sensor.bgs**” BGScript source file.

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 Low Energy 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.

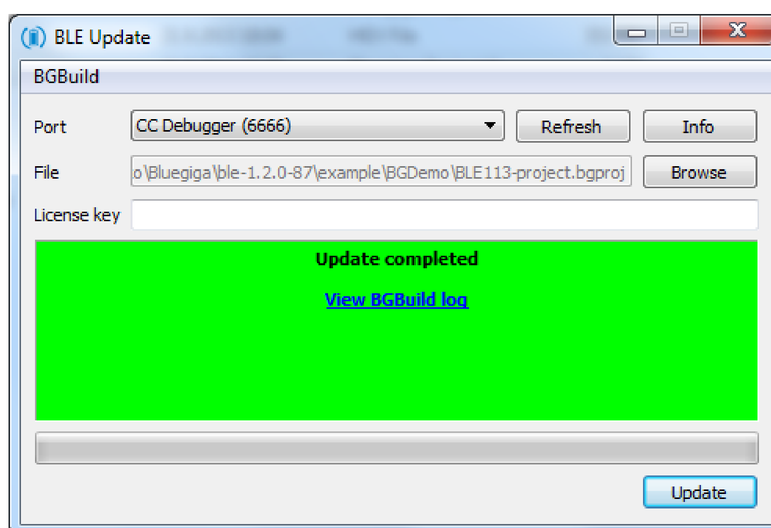


Figure 12: Compile and install with BLE Update tool

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.

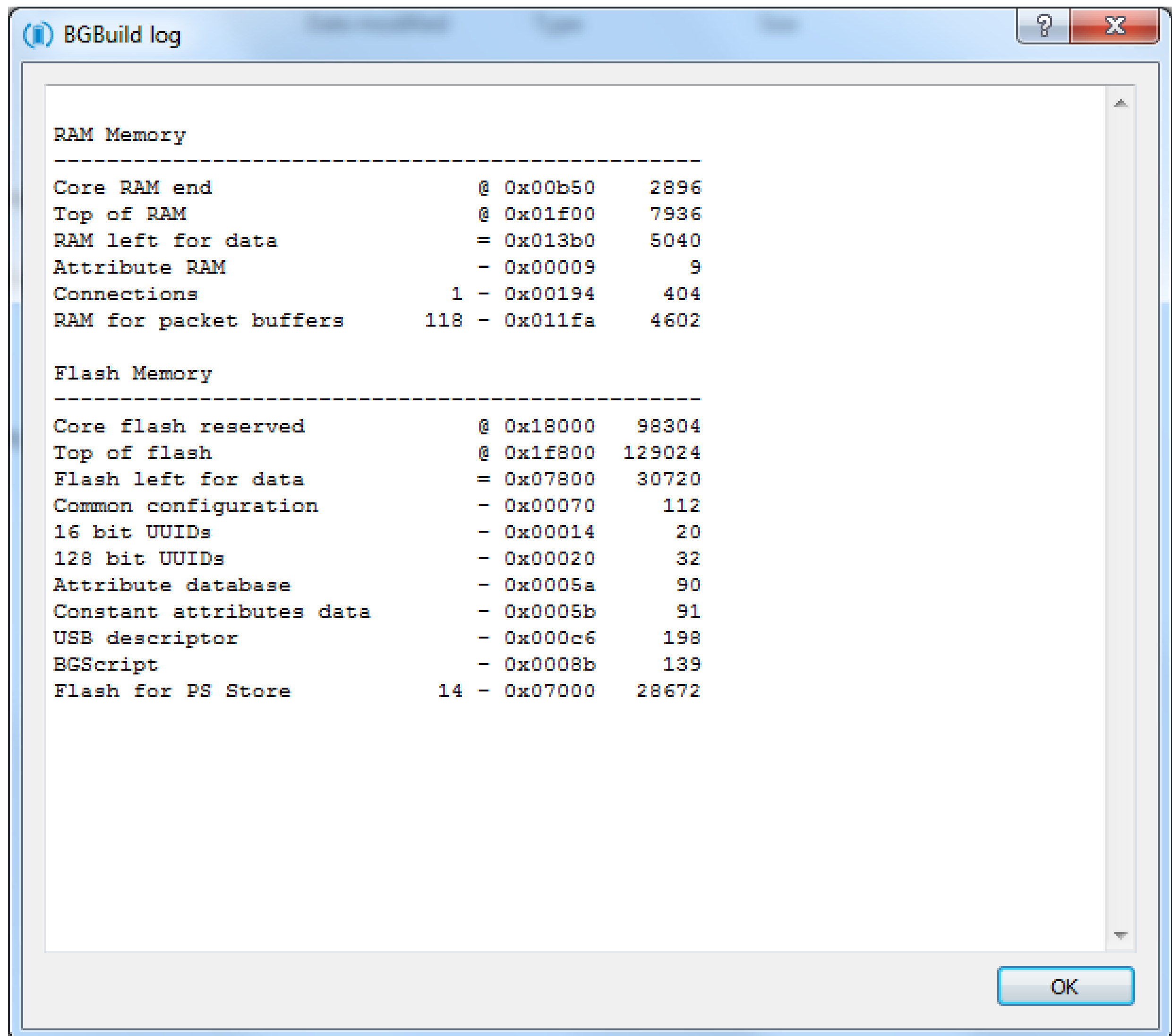


Figure 13: BLE Update build log

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 14: Compiling with BGBuild.exe

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

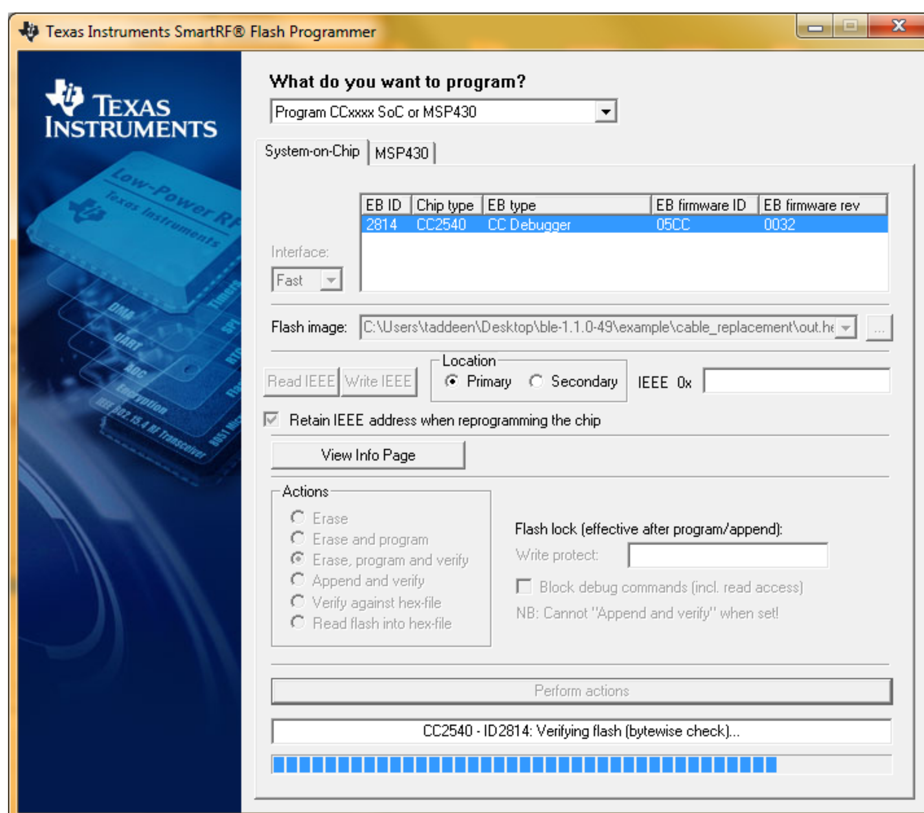


Figure 15: TI's flash programmer tool

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.

5.7 Testing the Glucose Sensor

5.8 Testing with BLEGUI software

This section describes how to test the Glucose Sensor application with BLEGUI software.

5.8.1 Discovering the Glucose Sensor

As soon as the Glucose Sensor is powered on, it starts to advertise itself. At this point, a BLED112 USB dongle can be used to detect the glucose sensor using the BLEGUI software.

Preparations:

1. Connect BLED112 USB dongle to a PC
2. Start BLEGUI software
3. Select the correct COM port from the drop down menu and press **Attach**
4. Execute the **Command – Info** command to make sure the communication works

Discovering the Glucose Sensor:

1. Set desired scan parameters, check **Active Scanning** box and press **Set Scan Parameters** button
2. Select **Generic** scanning mode and **Start** scanning
3. If the glucose sensor is powered on, in range, and not connected, it should appear in the main view.

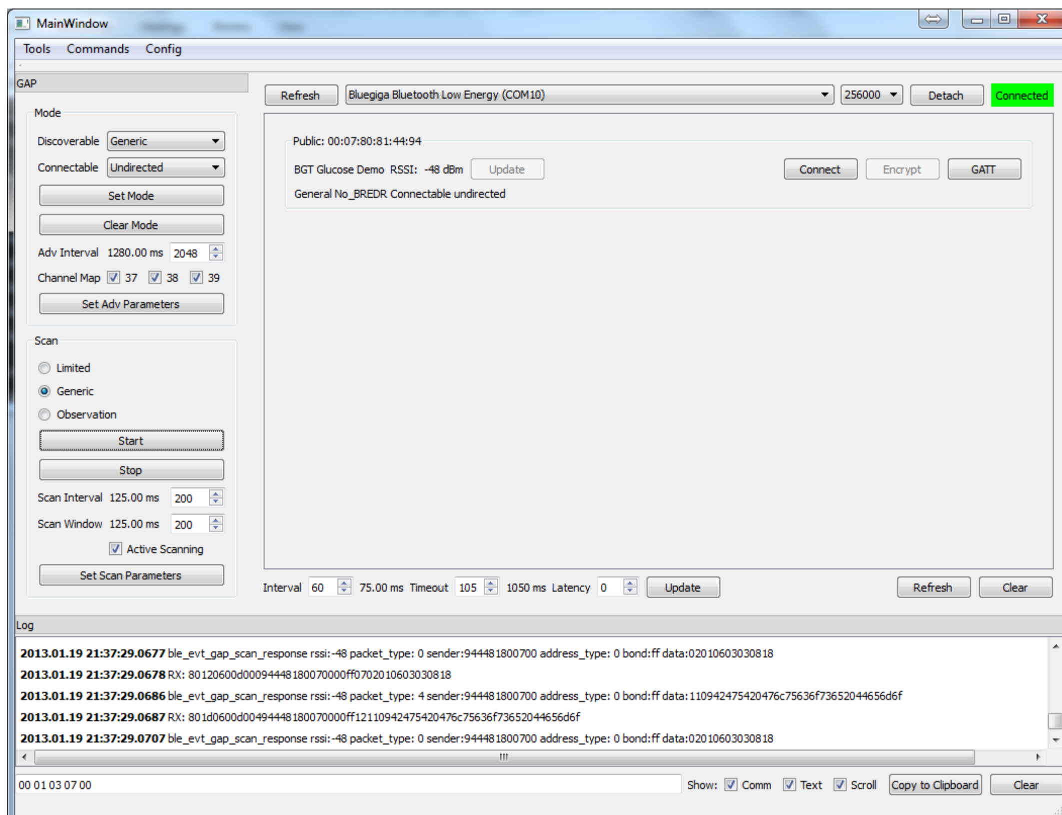


Figure 16: Scanning for the Glucose Sensor

5.8.2 Establishing a *Bluetooth* connection

1. Press the **Connect** button located next to the device you want to connect to.
    a. If the connection is successful, the connect button will change to **Disconnect**.
    b. If the connection fails, an error message will appear in the **Log** view.

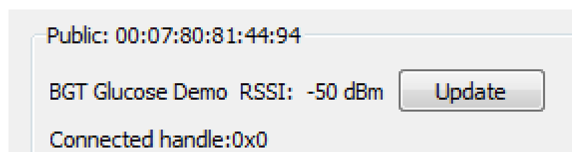


Figure 17: Connected to Glucose Sensor

2. Open the **Tools → Security Manager** dialog and check the "Bondable" option.

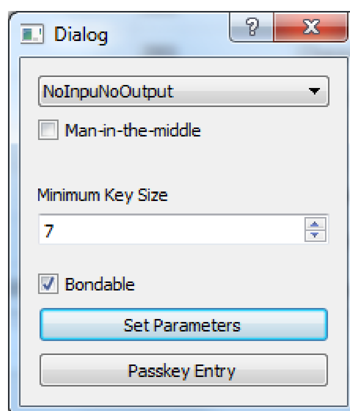


Figure 18: Glucose Profile roles

3. Click the "**Set Parameters**" button in the dialog to enable bonding, then close the dialog.
4. Press the **Encrypt** button located between the **Disconnect** button and **GATT** button. This will **bond** the BLED112 with the BLE112 running the glucose sensor project.
    a. If encryption is successful, you will see the following entry appear in the **Log** view:
        2013.01.19 22:08:49.0912 ble_rsp_sm_encrypt_start handle: 0 result: 0 ['No Error']
    b. If the connection fails, an error message will appear in the **Log** view.

5.8.3 Discovering services

Once you've connected to a device, you can use the ATTribute protocol to discover what services it supports. To discover the services of the glucose sensor:

1. Press the ***GATT*** button of the device you've just connected in order to start the GATT tool.
2. Press ***Service Discover*** button to start a GATT primary service discovery procedure.

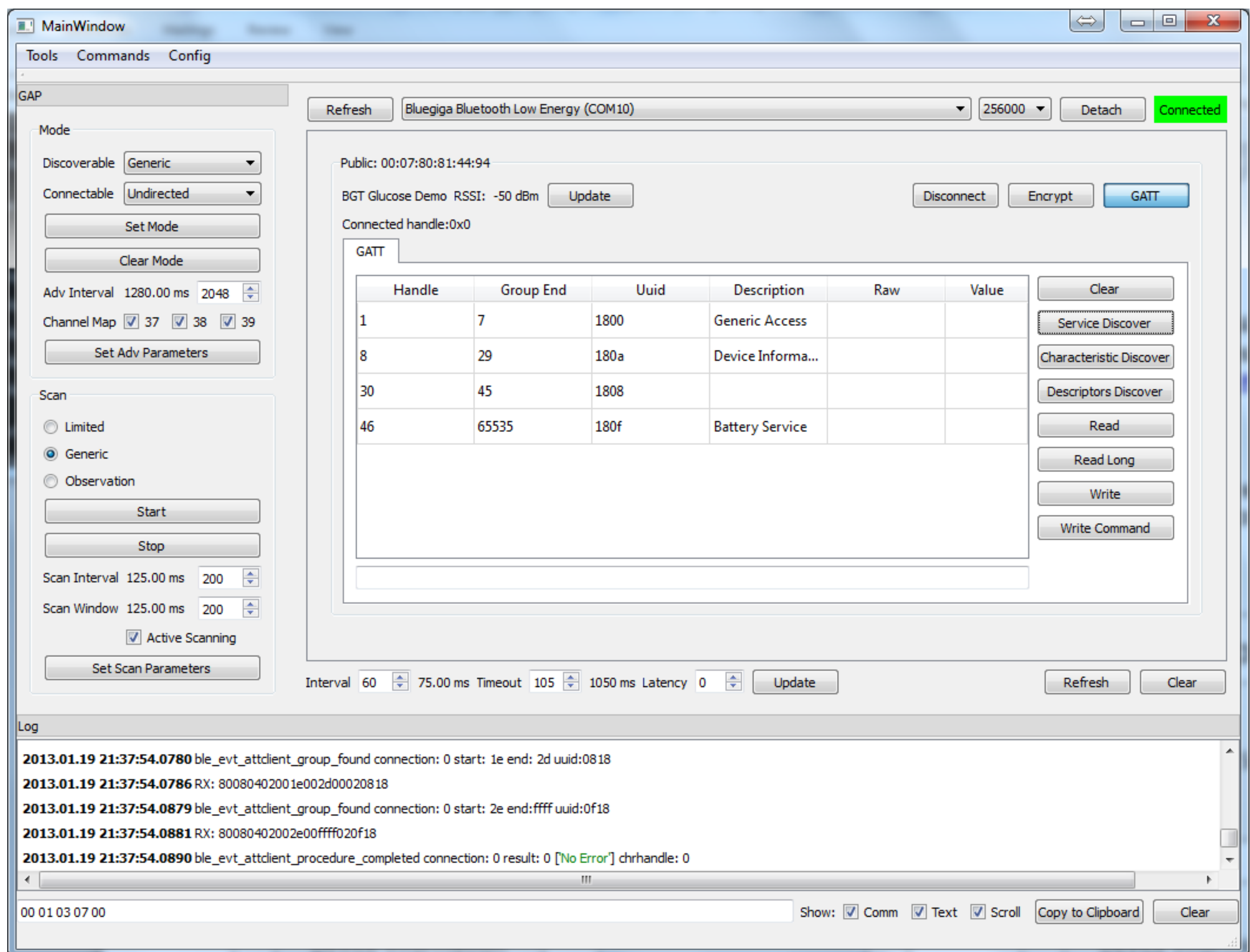


Figure 19: GATT service discovery

The four services defined in the **gatt.xml** should be visible in the GATT tool.

5.8.4 Subscribing to Notifications for Glucose Measurement and Context

The **Glucose Measurement** and **Glucose Measurement Context** attributes cannot be read directly, but only push new data out via **notifications**. Notifications are not enabled by default, and must be enabled explicitly. Notification settings (and indication settings) are set per-client, and will only persist between disconnections if a client has been bonded. Otherwise they must be re-enabled after disconnecting.

In order to enable the notifications:

1. Select the Glucose Service (UUID **1808**) in the BLEGUI's GATT view
2. Press **Descriptors Discover** in order to see all characteristics and descriptors in the glucose service
3. Once the descriptors discovery is complete, select the **Client Characteristic Configuration** (UUID: **2902**) value that relates to the **Glucose Measurement** (UUID: **2A18**) and select it
4. In order to enable notifications for the **Glucose Measurement,** enter a "1" in the text box below the GATT table, then click **Write** to write the value to the **Client Characteristics Configuration.**
5. Finally, make sure the write operation is executed properly (see Log)
6. Repeat steps 3 & 4 for the **Glucose Measurement Context** (UUID: **2A34**) **Client Characteristic Configuration** (UUID **2902**) to enable notifications for that attribute as well.

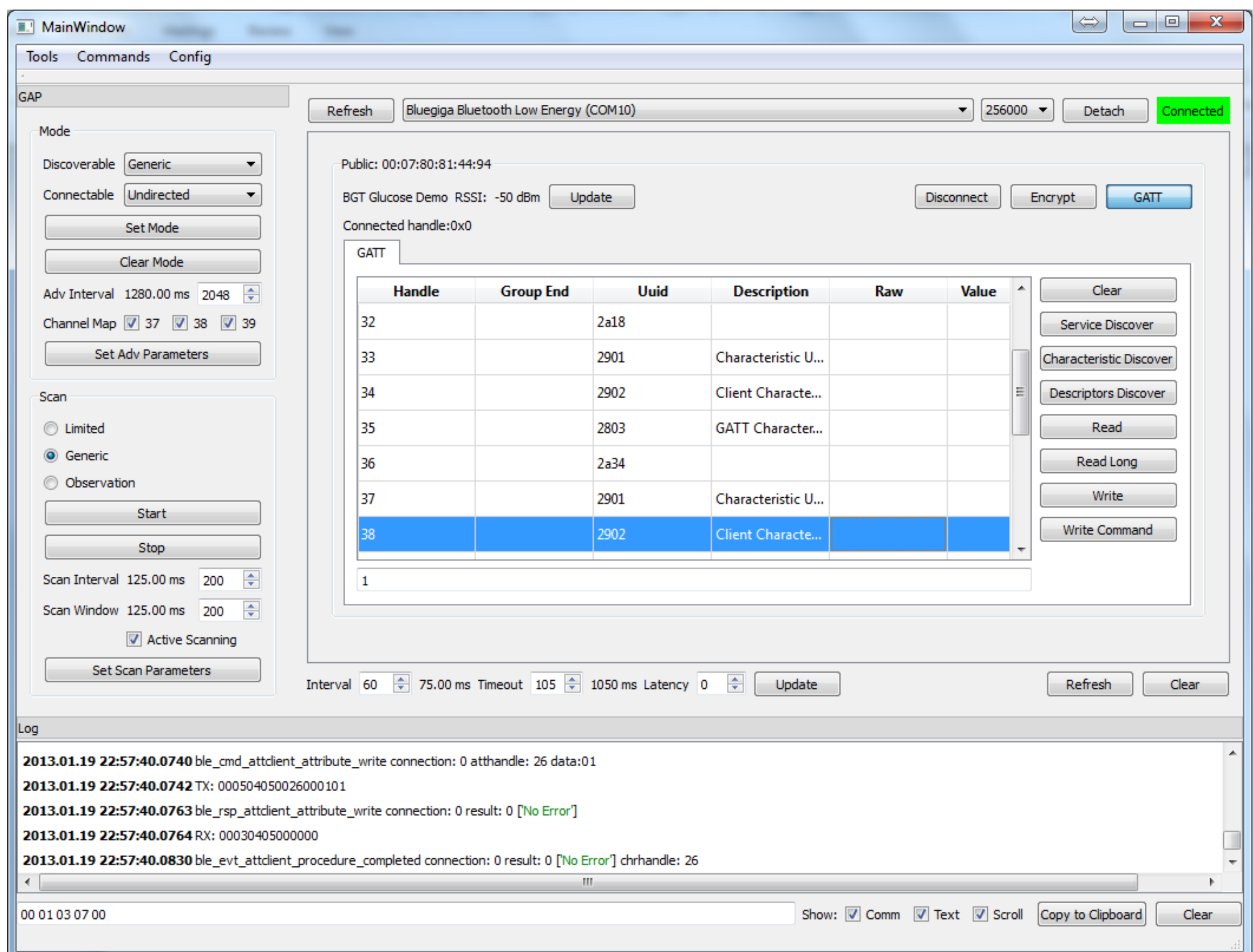


Figure 20: Enabling notifications

5.8.5 Testing Glucose Measurements

Now that we have connected, bonded, and enabled notifications, we can test to make sure the measurements are being taken and sent correctly.

In order to test measurements:

1. Press the **P0_0** button on the DKBLE112 (or, if not using the DK, momentarily bring **P0_0** HIGH).
2. You should see two new values come in below in the **Log** view, and:
   - One new value will appear in the "**Raw**" column of the **2A18** attribute (measurement).
     Example: 1b0400d9070718101e00040014a7110000
   - One new value will appear in the "**Raw**" column of the **2A34** attribute (context).
     Example: 5f04000313e10224001e640205e01de200
3. Verify from Log view that no error is received.

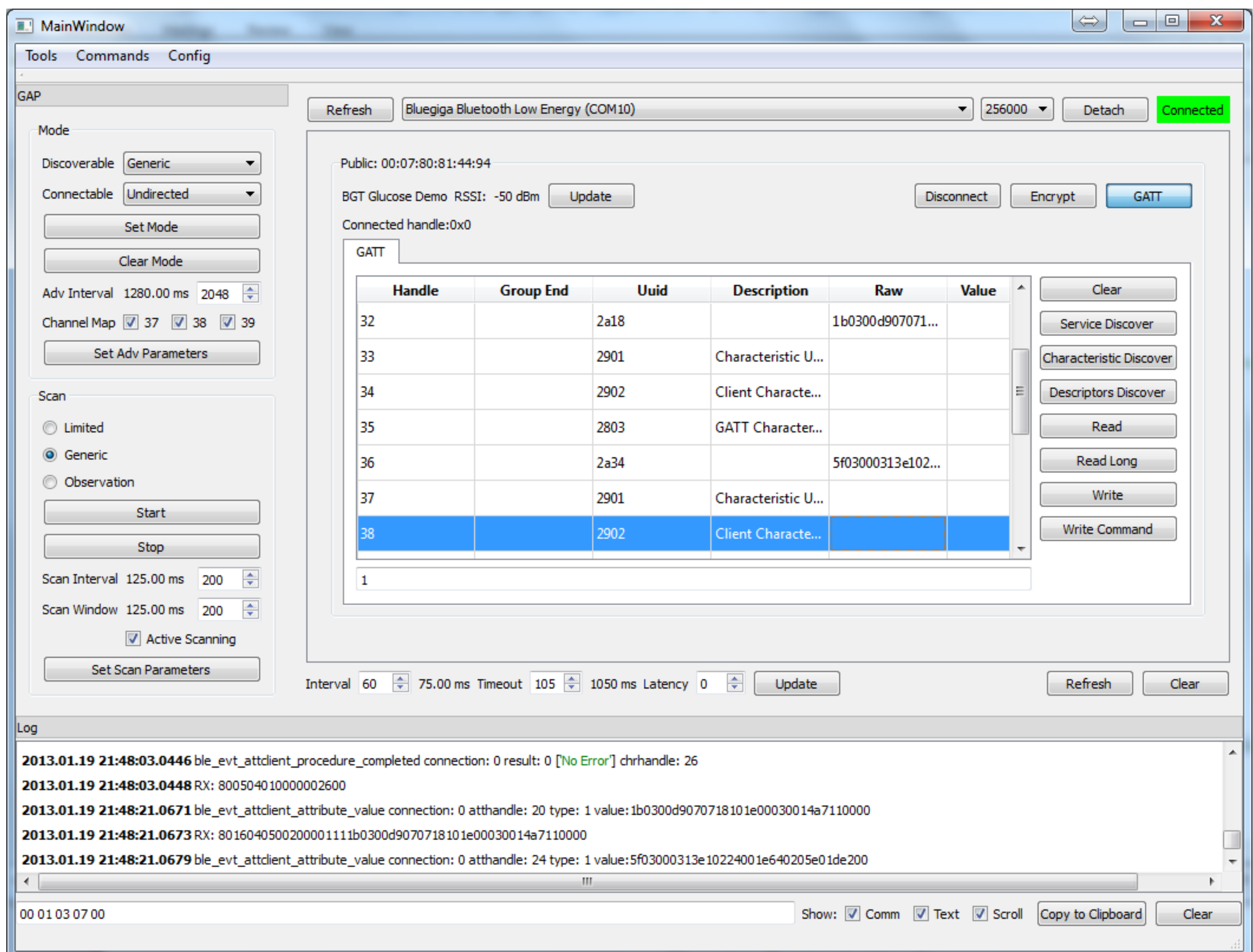


Figure 21: Receiving glucose measurements and context updates

5.8.6 Testing the Record Access Control Point with BLEGUI

The Record Access Control Point is used to access any stored measurements on the glucose sensor. After pressing the P0_0 button a few times to take sample measurements, you can send a “Retrieve All Records” command to test the record access method. It is difficult to see multiple records come in within the BLEGUI interface since they are repeatedly **notified** into the same attribute sequentially, but you can verify in the log that you receive multiple records.

In order to test record access:

1. Select the **Record Access Control Point** (UUID **2A52**) attribute.
2. Type “0101” in the text box below the GATT table, then click the **Write** button.
3. You should see a set of notified values appear in the **Log** window.
4. If you see an error in the Log, verify that your “**Bondable**” setting is correct within the **Tools → Security Manager** dialog, and verify that connection is in fact encrypted by clicking the “**Encrypt**” button again.

You can test “Retrieve First Record” by using “0105” in step 2, or “Retrieve Last Record” by using “0106”.

5.9 Testing with iPhone or iPad

There is an app created by Nordic Semiconductor for Bluetooth *Smart* enabled iOS devices (iPhone 4S/5, iPad 3/4/Mini), which can be used to test the glucose sensor.

This section briefly describes how to test the glucose sensor BGScript project using an iPhone.

5.9.1 Getting the App

You can find the app in iTunes here: https://itunes.apple.com/us/app/nrfready-utility/id497679111

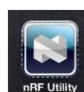
Alternatively, you can open the App Store on your iPhone and search for “nrfready” to find the app. Note that while the app will run perfectly well on an iPad, you may need to **specifically search for iPhone apps**, because it isn’t currently built as a universal app.

5.9.2 Testing the App

Once you’ve installed the App to your iPhone or iPad, the following steps are required:

1. Power on the glucose sensor (for example DKBLE112)
2. If using the DKBLE112 with built-in peripherals, prepare the board as follows:
    - Set **POTENTIOMETER** switch to **ON**
    - Set **ACCELEROMETER** switch to **OFF**
    - Set **DISPLAY** switch to **ON**
    - Set **RS232** switch to **ON** (if using UART debug output from the on-board DB9 port)
    - Press the **RESET DISPLAY** button, then the **RESET BLE112** button.
    - You should now see “Glucose Demo / Advertising ?” on the LCD:

Figure 22: “Advertising” display on DKBLE112 LCD

3. Start the **nRF Utility** iPhone application: 

4. Dismiss the disclaimer that appears.
5. Select the "**Bluetooth Smart**" item from the main welcome screen.
6. Tap the "**BGM**" icon from the list of services to enter the Blood Glucose Monitor screen.

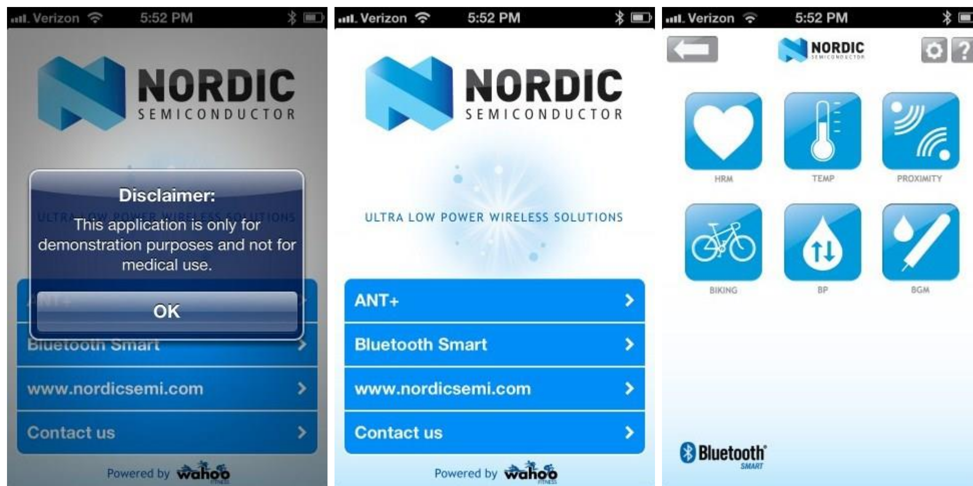


Figure 23: nRF Application Startup

7. Tap the gray "gear" icon near the top right corner to enter the Settings screen.
8. Slide the "**Wildcard**" option to **ON**.
9. Tap the gray "back arrow" icon near the top left, then click the "**CONNECT**" button.

Figure 24: nRF Application Connection

10. You should now be connected, but not yet bonded with the BLE112. If you are using the DKBLE112, the LCD should show the following:

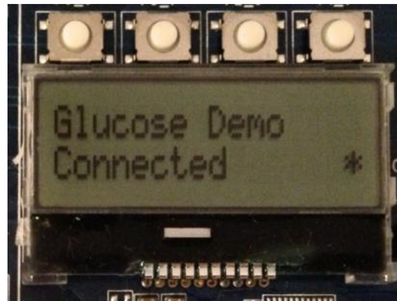


Figure 25: “Connected” display on DKBLE112 LCD

11. If you have not already bonded with the BLE112 running the glucose sensor demo, you should see a **Bluetooth Pairing Request** confirmation appear. Click the “**Pair**” button to bond. The DKBLE112 (if present) will now show “**Encrypted**”, like this:

Figure 26: App Bonding “Encrypted” display on DKBLE112 LCD

Note: if the pairing request does not appear at this point, you may need to do one or more of the following operations:

- Go to the Bluetooth area of the Settings app on your iPhone and “Forget” the **BGT Glucose Demo** device, if present.
- Turn Bluetooth off and then back on again on your iPhone.
- Completely power-cycle your iPhone (not usually required).
- Click the **P0_1** button on the DKBLE112 board to reset all settings and erase bonding entries.
- Reflash the glucose sensor project onto your BLE112 (also clears bonding entries).

12. Press the **P0_0** button on the DKBLE112 to trigger a glucose reading. The app should show a new record every time you press the button. You can affect the glucose concentration value by changing the potentiometer. For a thorough test, make sure you trigger at least a few records.

13. Tap the "**First**" button to retrieve the first stored record. Typically, the sequence number will be 0.
14. Tap the "**Last**" button to retrieve the last stored record.
15. Tap the "**All**" button to retrieve all records stored.

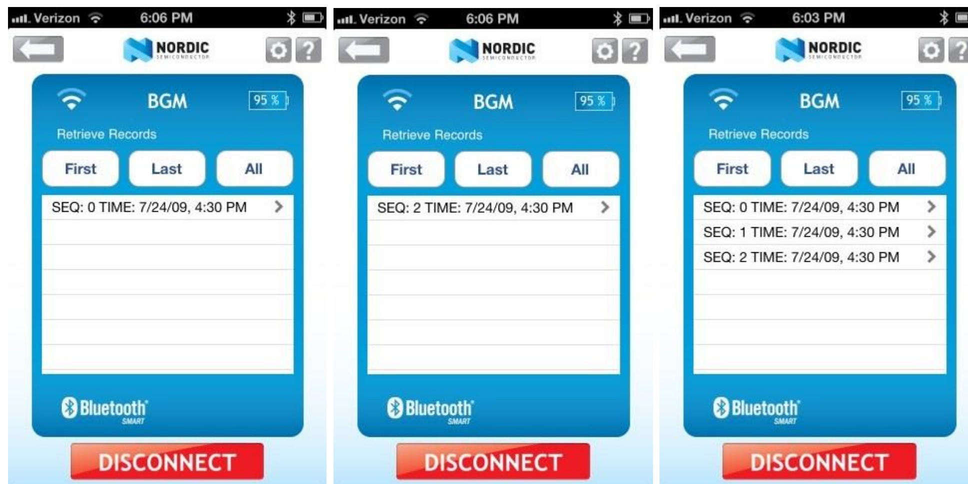


Figure 27: Stored Record Retrieval with nRF App

16. Tap on any record to show the record detail:

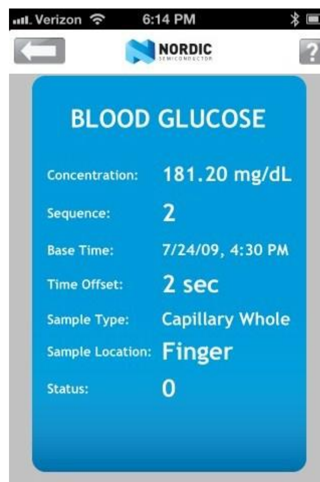


Figure 28: Glucose Record Detail View in nRF App

6 External resources

- *Bluetooth* 4.0 software development kit is available at : www.bluegiga.com
- BLE112 and DKBLE112 hardware documentation is available at : www.bluegiga.com
- Heart Rate Profile can be downloaded from: Heart Rate Profile
- *Bluetooth* SIG's developer portal: https://developer.*Bluetooth*.org/

Simplicity Studio

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

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

Disclaimer
Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required, or Life Support Systems without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications.

Trademark Information
Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, ClockBuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, Gecko OS, Gecko OS Studio, ISOmodem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.

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

http://www.silabs.com