

# **CABLE REPLACEMENT OVER BLE**

## **APPLICATION NOTE FOR BT121**

Friday, 18 June 2021

Document Revision: 1.2



## VERSION HISTORY

Date Edited	Comment
0.90	First draft
1.0	First release
1.1	Renamed "Smart Ready" to "Dual Mode" and "Classic" to "BR/EDR" according to the official Bluetooth SIG nomenclature.
1.2	Replaced inappropriate terms in accordance with the latest Bluetooth SIG recommendations.

## TABLE OF CONTENTS

<b>1</b>	<b>Introduction.....</b>	<b>4</b>
1.1	Uses for Cable Replacement .....	4
1.2	Cable Replacement profile .....	4
<b>2</b>	<b>Prerequisites .....</b>	<b>8</b>
2.1	Installing the <i>Bluetooth</i> Dual Mode SDK .....	8
2.2	Setting up the Server device .....	9
2.3	Setting up the Client Device .....	10
<b>3</b>	<b>Walkthrough of the Cable Replacement Applications.....</b>	<b>10</b>
3.1	Project configuration file .....	10
3.2	Hardware configuration file .....	12
3.3	GATT database configuration .....	13
3.4	BGScript™ code.....	17
3.5	Compiling the applications .....	21
3.6	Installing the firmware .....	22
3.7	BGTool .....	22
3.8	Testing the applications .....	23

# 1 Introduction

This application note discusses how to use the Low Energy (LE) part of the firmware in the BT121 modules in order to make *Bluetooth* LE devices which can act as *serial cable replacement*. This means that data normally carried over a physical serial cable can be instead transmitted over the air using *Bluetooth* radio technology, making the communication between devices wireless.

The reference hardware platform used in the application note is the DKBT *Bluetooth* Dual Mode Development Kit (in short DKBT). You can also use the BT121 *Bluetooth* Dual Mode module, but for development purposes the DKBT hardware provides easier access to various components.

The reference applications discussed in this application note are developed with the Bluegiga BGScript™ scripting language, which means that the applications can run fully independently on the BT121 *Bluetooth* Dual Mode module without the need for an extra host MCU.

This application note relies on information found in the BT121 Data Sheet, DKBT User Guide, and *Bluetooth* Dual Mode API Reference. Please consult those for further details.



This application note assumes you have read and understood the ***Bluetooth Dual Mode Software Getting Started Guide***.

## 1.1 Uses for Cable Replacement

One way to use wireless cable replacement is to place a BT121 *Bluetooth* Dual Mode module at both ends of a link between two serial devices, simply replacing the wired serial link. This is normally achieved with the *Bluetooth* BR/EDR's Serial Port Profile (SPP) but in this application note it will be discussed how this can also be accomplished over a *Bluetooth* Low Energy link.

Another possibility is to connect a serial data sink/source to one BT121 *Bluetooth* Dual Mode module and implement the other end of the link with another *Bluetooth* device, without moving the traffic back to a serial cable link at the other end. For instance, a legacy serial device could use the BT121 to communicate over *Bluetooth* Low Energy with a smartphone application that understands the protocol used by the legacy device.

## 1.2 Cable Replacement profile

A *Bluetooth* Low Energy proprietary profile is implemented in the BT121 *Bluetooth* stack on top of *Bluetooth* Generic Attribute Profile (GATT). This Cable Replacement profile defines the communication protocol for transferring the data over a *Bluetooth* link using Attribute Protocol (ATT) methods.

The Cable Replacement profile must be implemented in both ends of the *Bluetooth* link.

The profile contains two roles:

- **Server:** The server device acts as ATT server and provides the Cable Replacement service for data transfer.
- **Client:** The client device initiates the *Bluetooth* links to the server device, and uses the service provided by the Server.

In a typical use case the BT121 takes the Server role, being connected to a serial data source over its UART port and transparently forwarding data between the port and the Client device. However, the client role is also implemented in the BT121 stack, so it can act as both sides of a Cable Replacement connection.

### 1.2.1 Service

The Cable Replacement profile does not define the service UUID to be used; it can be freely assigned to a suitable value by the Server implementation.

The profile does not as such restrict the number of Cable Replacement services on a Server device. However, the BT121 has one UART and in most cases having a single service makes most sense. The only reason for having two services would be to allocate one service for reliable data transfer and the other for unreliable (see below for details).

The service may contain any number of characteristics, but it must contain exactly one characteristic for transferring data, as described below.

### 1.2.2 Data characteristic

The data is transferred between the Client and the Server using ATT protocol methods on an ATT characteristic designated for data transfer by the Server. The characteristic shall be contained in the service designated by the Server for the Cable Replacement profile.

Note that the Cable Replacement profile does not define the characteristic UUID to be used; it can be freely assigned to a suitable value by the Server implementation.

Depending on Server configuration, the following alternatives for data transfer are possible:

- **Write Request/Response** (client to server), **Value Indication/Confirmation** (server to client): these methods are acknowledged by the recipient's response message, which makes data transfer using these operations reliable; if data is lost, it is detected on the ATT level instead of being silently ignored. The acknowledgements also provide an implicit flow control for the data.
- **Write Command** (client to server), **Value Notification** (server to client): these methods are not acknowledged by the recipient, meaning that data can be silently lost. If these methods are used and reliable data transfer or data flow control is needed, it needs to be implemented on top of the cable replacement profile by the application.

It is the server's responsibility to configure the characteristic appropriately for the application; the GATT properties for the characteristic shall have either Write and Indicate, or Write Without Response and Notify, property bits set, but not both pairs nor one of a pair without other. Also, the Read property is not allowed.

### 1.2.3 Profile operation

The client and the server must follow certain procedures in order to transfer data.

A data transfer session is divided into three distinct phases: connection establishment, actual data transfer, and connection teardown. These are described in detail below.

#### 1.2.3.1 Connection establishment

The typical connection establishment procedure, where the Cable Replacement GATT Client device acts as a LE Central and actively forms a LE connection, while the GATT Server device acts as the LE advertising Peripheral, is described below. Note however that the LE roles can be reversed if needed by the application.

Before a connection is established, the Server device has to be connectable; it may also be discoverable, and it may advertise the Cable Replacement service UUID.

As a prerequisite, since the Cable Replacement service UUID can be freely assigned, the Client shall know the UUID used by the server to be connected to beforehand. It shall also know the UUID of the Data characteristic.

Unless an LE connection already exists between the devices the Client shall initiate the connection. Connection parameters may be set as appropriate.

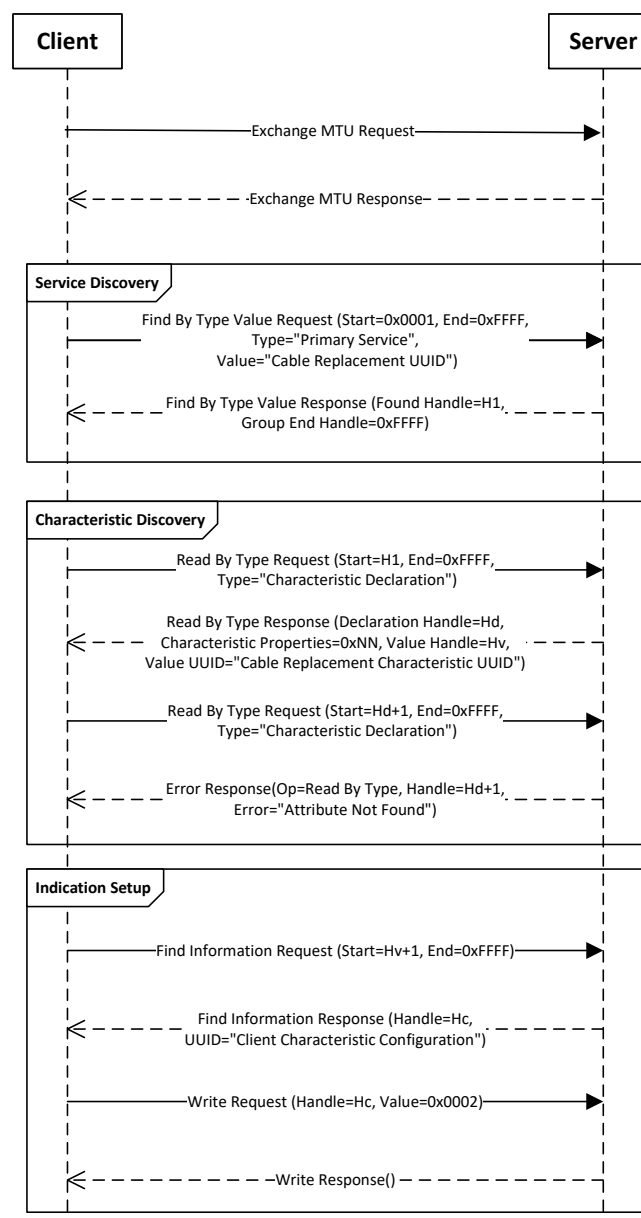
If necessary for the application, the Client device may encrypt the connection before connection establishment is finalized. Note that the Server device may refuse to serve data requests on an unencrypted connection.

The amount of data sent in each ATT method is limited by the maximum ATT MTU. ATT protocol headers reduce the space available for payload by further 3 bytes. Thus, it is recommended to negotiate MTU larger than the default of 23 bytes for improved throughput before connection establishment is finalized.

The Client may discover the Data characteristic handle from the Server, if necessary, using the normal GATT procedures.

The Client shall then write the Client Characteristic Configuration value for the Data characteristic so that it contains either the Notification bit or the Indication bit, depending on the Server configuration. After this is done, the connection is considered established and data may be transferred.

An example message sequence chart below describes the connection establishment. Note that in the example there is only one Cable Replacement service present in the service, and it contains only the Data characteristic and its associated metadata. The chart refers to four different handles: H1 is the Cable Replacement service handle, while Hd, Hv, and Hc are the Data characteristic declaration, value, and configuration handles, respectively. The Data characteristic handles are also used in the following charts.



**Figure 1: Example Connection Establishment Message Sequence**

### 1.2.3.2 Data transfer

As described above, the actual data transfer is done using ATT protocol methods on the Data characteristic. The Client will write to the characteristic on the Server, while the server will indicate/notify the Client.

Note that reading the Data characteristic is not permitted.

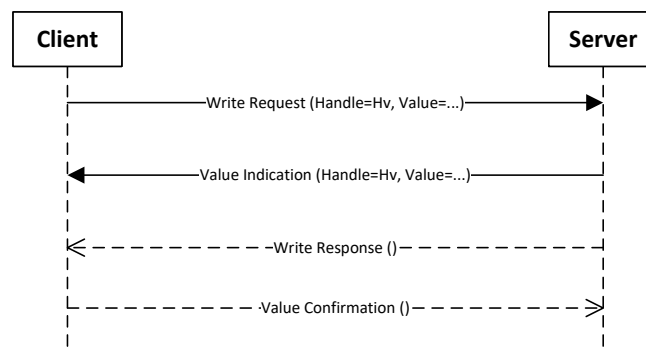
Both the server and the client may refuse to acknowledge data requests on an unencrypted or unauthenticated connection; connection encryption should be set up before data transfer begins.

Data may flow to both directions simultaneously. If acknowledged writes and indications are used, ATT protocol has a limit of one outstanding method at a time for each direction. This means that when acknowledged methods are used, the acknowledgements provide implicit flow control; until the outstanding method is acknowledged, a new one may not be invoked.

Note that the ATT protocol limitation applies to the connection between the devices as a whole, including methods related to other GATT based profiles. Care must be taken if multiple profiles are active at the same time.

If unacknowledged methods are used, data loss may occur if, e.g., all of the recipient's receive buffers are in use. Thus, in this case it is strongly recommended that some method of flow control and transfer reliability is implemented by the application.

The example message sequence below shows acknowledged methods for data transfers to both directions overlapping.



**Figure 2: Example Data Transfer Message Sequence**

### 1.2.3.3 Connection teardown

To end the Cable Replacement session, the Client device shall write the Client Characteristic Configuration value for the Data characteristic so that the bit set at connection establishment is cleared. After this is done, the connection is considered torn down.

The client device may close the LE connection after Cable Replacement connection teardown if required.

The following example message sequence chart shows the connection teardown sequence, which consists only of clearing the indication/notification bit in the Data characteristic client configuration.

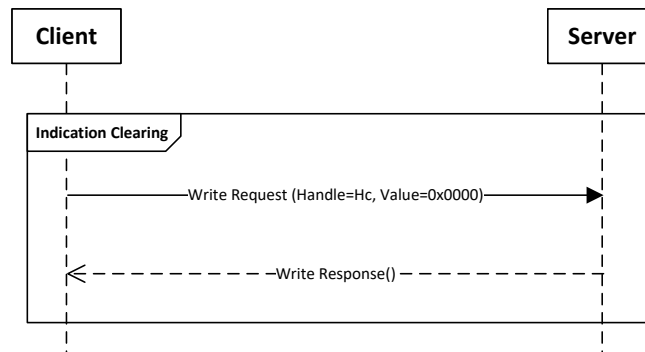


Figure 3: Example Connection Teardown Message Sequence

## 2 Prerequisites

Before you can test the Cable Replacement demo application or start developing your own applications the following prerequisites must be met.

1. You have installed *Bluetooth* Dual Mode SDK
2. You have DKBT *Bluetooth* Dual Mode Development kit connected to a serial port data source. This device will act as the Cable Replacement Server.
3. You have a *Bluetooth* LE capable device which can run user applications, for instance a PC, a smartphone, or another DKBT. This device will act as the Client.

The following subsections walk you through the above steps.

### 2.1 Installing the *Bluetooth* Dual Mode SDK

Install the *Bluetooth* Dual Mode SDK by following the on screen instructions in the installer. The SDK installer is available from the Bluegiga web site at <https://www.silabs.com/wireless/bluetooth/bluegiga-classic-legacy-modules/device.bt121-a> under the "Software & Tools" section.

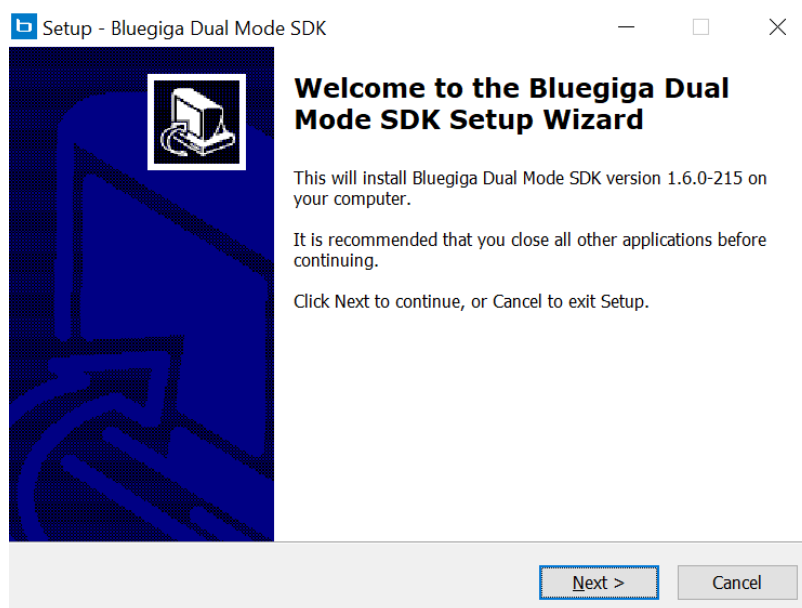


Figure 4: the SDK installer



## 2.2 Setting up the Server device

Below is an overview of the DKBT hardware, showing how the BT121 carrier board connects to the development kit main board which has breakout pins, LCD display, USB UART converter and various other hardware components. You can also use a BT121 *Bluetooth* Dual Mode module by itself, but the development kit provides easy connectivity that helps in getting started, and in this document, we assume it is used.

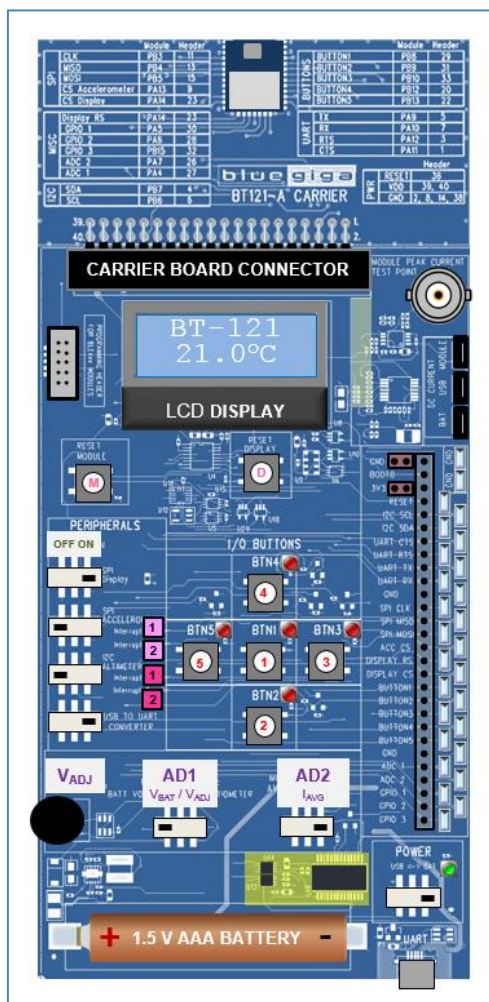


Figure 5: DKBT hardware

### 2.2.1 Serial Port Connection

The connection between a DKBT Development kit with the BT121 carrier board and a device which uses serial port for communication is very straightforward. UART interface pins (UART\_RX, UART\_TX, UART\_RTS and UART\_CTS) are available directly on the DKBT main board. Please consult the BT121 Datasheet and DKBT User Guide for more details.

Serial port speed and flow control will be set in project configuration, described in Chapter 3.1. Using RTS/CTS hardware flow control is strongly recommended. Serial port parameters default to 8N1, but can be set at runtime using BGScript.

After making the connection the Server device is ready to be programmed with the Cable Replacement example server application, as described later in Chapter 3. The example server application will route data from the Cable Replacement connection to the serial connection and vice versa when a Client device is connected to the service.

### 2.2.2 USB Connection

Alternatively, if you do not have a serial device at hand, you can use the USB to UART converter available on the DKBT main board. Connect the kit to a PC using a standard USB cable; the converter will appear as a serial port on the PC. You can use any serial terminal emulator application to set up serial port parameters and transmit data between the BT121 UART and the PC.



Note that the USB to UART converter is enabled with a switch on the DKBT main board.

## 2.3 Setting up the Client Device

This document will assume you will use another DKBT in the Client role. You may also use a BT121 *Bluetooth* Dual Mode module or implement the Cable Replacement profile on a 3<sup>rd</sup> party device (such as a smartphone) but doing so is out of scope for this document.

Connect the Client DKBT to a PC using a USB cable and set up a terminal emulation application as described previously in Chapter 2.2.2.

After making the connection the Client device is ready to be programmed with the Cable Replacement client example application, described in Chapter 3. The example client application will route data from the Cable Replacement connection to the serial connection and vice versa, like the example server application does; the only difference is that the client is the active party in forming the connection.

## 3 Walkthrough of the Cable Replacement Applications

This section walks you through the Cable Replacement demo applications that are present in the *Bluetooth* Dual Mode SDK example folder. The purpose of the section is to give you an overview of the *Bluetooth* Dual Mode SDK and how you can start building your own applications.

There are two example applications, one for the Server role and one for the Client role. While there are differences in the application BGScript, as the Client device is the one which actively opens the connection to the Server, the overall architecture is the same in both and the project configuration files are close to identical.

### 3.1 Project configuration file

Building a *Bluetooth* Dual Mode project starts always by making a *project file*, a simple XML file which defines the resources used in the project. The Cable Replacement server project file contents are shown on next page in [Figure 6](#). The Cable Replacement client project file is identical, except for the BGScript and output file definitions.

```

<!-- Project configuration including BT121 device type -->
<project device="bt121">

    <!-- XML file containing GATT service and characteristic definitions both for BLE and GATT over BR -->
    <gatt in="gatt.xml" />

    <!-- Local hardware interfaces configuration file -->
    <hardware in="hardware.xml" />

    <!-- BGScript source code file -->
    <scripting>
        <!-- LE cable replacement server -->
        <script in="server.bgs" />
    </scripting>

    <!-- Firmware output files -->
    <image out="BT121_le_cable_replacement_server.bin" />
</project>

```

**Figure 6: Project file**

An explanation of the project file content is shown in [Table 21](#) below:

<project device="bt121">	This tag starts the project definition and the project file must end in </project> tag. The <b>device</b> is used to define for which <i>Bluetooth</i> module the project is used for.
<gatt in="gatt.xml" />	The <gatt> tag is used to define the XML file containing the GATT service and characteristic database used for both <i>Bluetooth</i> LE and GATT over BR profile. <b>NOTE:</b> This example uses a minimal GATT database.
<hardware in="hardware.xml" />	The <hardware> tag defines the file containing the hardware configuration for interfaces like UART, SPI or I <sup>2</sup> C.
<scripting> <script in="server.bgs" /> </scripting>	Inside the <scripting> tags all the included BGScript code files are defined. Individual script files need to be defined with the <script> tags.
<image out="BT121_le_cable_replacement_server.bin" />	The <image> tag defines the name of the BGBuild compiler output file. The generated .bin file contains the <i>Bluetooth</i> LE stack, the GATT database, the hardware configuration and the BGScript code.

**Table 1: Project file explained**



The full syntax of the project configuration file and more examples can be found from the ***Bluetooth Dual Mode Module Configuration Guide***.

## 3.2 Hardware configuration file

The next logical step is to define the hardware configuration of the *Bluetooth* module: to define which hardware interfaces are enabled and what are their default settings and/or configurations. Cable Replacement server hardware configuration file content is shown below in [Figure 7](#). The Cable Replacement client hardware configuration file is identical.

```
<hardware>

  <!-- Sleep modes disabled -->
  <sleep enabled="false"/>

  <!-- UART enabled @115200bps -->
  <uart baud="115200" flowcontrol="true" bgapi="false" />

  <!-- SPI enabled for dev.kit display -->
  <spi channel="1" alternate="2" clock_idle_polarity="high" clock_edge="1" endianness="msb" divisor="256" />

  <!-- I/O configuration disabled -->
  <port index="0" output="0x4000"/>
  <port index="1" output="0x4000" input="0x2000" interrupts_rising="0x2000"/>

</hardware>
```

Figure 7: Hardware configuration

An explanation of the hardware file content is shown in [Table 2](#) below:

<code>&lt;sleep enabled="false" /&gt;</code>	This tag defines whether the MCU sleep modes (low power modes) are disabled.
<code>&lt;uart baud="115200" flowcontrol="true" bgapi="false" /&gt;</code>	The <code>&lt;uart&gt;</code> tag is used to define the UART interface settings and whether it is enabled. In the example project the UART is used for Cable Replacement data exchange. BGAPI serial protocol is disabled as the project uses a BGScript application to implement the application logic.
<code>&lt;spi channel="1" alternative="2" clock_idle_polarity="high" clock_edge="1" endianness="msb" divisor="256" /&gt;</code>	The <code>&lt;spi&gt;</code> tag defines whether the SPI interface is enabled or not, as well as the used SPI interface (of the two available) and its settings. In the example project SPI interface is used to communicate with the display on the DKBT development kit.
<code>&lt;port index="0" output="0x4000" /&gt;</code> <code>&lt;port index="1" output="0x4000" input="0x2000" interrupts_rising="0x2000" /&gt;</code>	The <code>&lt;port&gt;</code> tag is used to define the I/O port default configuration and states. On the BT121 module port index 0 refers to pins named PA and port index 1 to pins named PB. In the example project pin 15 is configured as output for both ports 1 and 2 and they are used to control the state of the display on the DKBT development kit.

Table 2: Hardware configuration file explained



The full syntax of the hardware configuration file and more examples can be found in the ***Bluetooth Dual Mode Module Configuration Guide***.

### 3.3 GATT database configuration

The next step is to configure the GATT database entries used by the device. This is again done by editing the GATT database XML file you have defined in the project file.

The Server GATT database is described in detail below. It is a simple one, containing entries for Generic Access, Device Information, and Cable Replacement profiles.

The Client GATT database is even simpler, containing only entries for Generic Access and Device Information profiles.

#### 3.3.1 Bluetooth Generic Access Profile service entry

Generic Access Profile (GAP) service and its mandatory characteristics, Device Name and Appearance, are configured in the GATT database XML file as shown below. For more information on Generic Access Profile, refer to the *Bluetooth SIG* Core Specification document.

```
<!-- Generic Access Service -->
<!-- https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.generic -->
<service uuid="1800">

    <description>Generic Access Service</description>

    <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth -->
    <characteristic uuid="2a00">
        <properties read="true" const="true" />
        <value>LE Cable Replacement Server</value>
    </characteristic>

    <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth -->
    <characteristic uuid="2a01">
        <properties read="true" const="true" />
        <value type="hex">0</value>
    </characteristic>

</service>
```

Figure 8: GATT database entry for GAP service

The table below explains the GAP service entry in detail.

<pre>&lt;service uuid="1800"&gt;   &lt;description&gt;Generic Access Service&lt;/description&gt;   ... &lt;/service&gt;</pre>	<p>This defines the UUID of the service. For GAP the UUID is assigned by <i>Bluetooth SIG</i>, must be as shown here and <b><u>must not be changed</u></b>.</p> <p>The service definition contains inside it characteristic definitions which are described below.</p>
<pre>&lt;characteristic uuid="2A00"&gt;   &lt;properties read="true" const="true"/&gt;   &lt;value&gt;LE Cable Replacement Server&lt;/value&gt; &lt;/characteristic&gt;</pre>	<p>The Device Name characteristic is mandatory. Its UUID is assigned by <i>Bluetooth SIG</i>, must be as shown here and <b><u>must not be changed</u></b>. Value may be freely defined.</p> <p>The characteristic properties define how the characteristic can be accessed, and should not be changed for this characteristic.</p>
<pre>&lt;characteristic uuid="2A01"&gt;   &lt;properties read="true" const="true"/&gt;   &lt;value type="hex"&gt;0&lt;/value&gt; &lt;/characteristic&gt;</pre>	<p>The Appearance characteristic is mandatory. Its UUID is assigned by <i>Bluetooth SIG</i>, must be as shown here and <b><u>must not be changed</u></b>. Value may be freely chosen from the values defined by <i>Bluetooth SIG</i>; refer to the <i>Bluetooth Assigned Numbers</i> document.</p> <p>The characteristic properties define how the characteristic can be accessed, and should not be changed for this characteristic</p>

Table 3: GAP service entry explained



For typical use, do not modify GAP service entry except for the Device Name and Appearance values.

### 3.3.2 Bluetooth Device Information service entry

Device Information (DI) service and its example characteristics, Manufacturer Name, Model Number, and System ID, are configured in the GATT database as shown below. For more information on Device Information service, refer to the *Bluetooth SIG Device Information Service Specification* document.

```
<!-- Device Information Service -->
<!-- https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.device_i
<service uuid="180A">

  <description>Device Information Service</description>

  <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth
  <characteristic uuid="2A29">
    <properties read="true" const="true" />
    <value>Silicon Labs</value>
  </characteristic>

  <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth
  <characteristic uuid="2A24">
    <properties read="true" const="true" />
    <value>BT121</value>
  </characteristic>

  <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicViewer.aspx?u=org.bluetooth
  <characteristic uuid="2A23">
    <properties read="true" const="true" />
    <value type="hex">000780FFFE000047</value>
  </characteristic>

</service>
```

Figure 9: GATT database entry for DI service

The table below explains the DI service entry in detail.

<pre>&lt;service uuid="180A"&gt;   &lt;description&gt;Device Information Service&lt;/description&gt;   ... &lt;/service&gt;</pre>	<p>This defines the UUID of the service. For DIS the UUID is assigned by <i>Bluetooth SIG</i>, must be as shown here and <b><u>must not be changed</u></b>.</p> <p>The service definition contains inside it characteristic definitions which are described below.</p>
<pre>&lt;characteristic uuid="2A29"&gt;   &lt;properties read="true" const="true"/&gt;   &lt;value&gt;Silicon Labs&lt;/value&gt; &lt;/characteristic&gt;</pre>	<p>The Manufacturer Name characteristic UUID is assigned by <i>Bluetooth SIG</i>, must be as shown here and <b><u>must not be changed</u></b>. Value may be freely defined.</p> <p>The characteristic properties define how the characteristic can be accessed, and should not be changed for this characteristic.</p>
<pre>&lt;characteristic uuid="2A24"&gt;   &lt;properties read="true" const="true"/&gt;   &lt;value&gt;BT121&lt;/value&gt; &lt;/characteristic&gt;</pre>	<p>The Model Number characteristic UUID is assigned by <i>Bluetooth SIG</i>, must be as shown here and <b><u>must not be changed</u></b>. Value may be freely defined.</p> <p>The characteristic properties define how the characteristic can be accessed, and should not be changed for this characteristic</p>
<pre>&lt;characteristic uuid="2A23"&gt;   &lt;properties read="true" const="true"/&gt;   &lt;value type="hex"&gt;000780FFFE000047&lt;/value&gt; &lt;/characteristic&gt;</pre>	<p>The System ID characteristic UUID is assigned by <i>Bluetooth SIG</i>, must be as shown here and <b><u>must not be changed</u></b>. Value may be freely defined but must consist of vendor OUI identifier and a device specific identifier as defined by the DI service specification.</p> <p>The characteristic properties define how the characteristic can be accessed, and should not be changed for this characteristic</p>

**Table 4: DI service entry explained**

### 3.3.3 Cable Replacement service entry

Cable Replacement service entry is also configured in the GATT database. The service may contain any number of characteristics but for clarity it is recommended that it contains only one, the characteristic used for data transfer.

Note that this service entry is needed only for the device acting as the *GATT* server.

```
<!-- Example cable replacement service -->
<!-- Service and characteristic UUID can be freely chosen -->
<service uuid="f6ec37db-bda1-46ec-a43a-6d86de88561d" advertise="true">

  <description>LE Cable Replacement Service</description>

  <characteristic uuid="af20fbac251849989af7af42540731b3" id="xle_data">
    <description>Data characteristic</description>

    <!-- One and only one of the following properties specification has to be uncommented -->

    <!-- Reliable data transfer uses ATT write requests and value indications -->

    <!-- Properties for reliable unencrypted data transfer -->
    <!--
    <properties write="true" indicate="true"/>
    -->

    <!-- Properties for encrypted data transfer -->
    <properties
      write="true" encrypted_write="true"
      indicate="true" encrypted_notify="true"/>

    <!-- Unreliable data transfer uses ATT write commands and value notifications -->

    <!-- Properties for unreliable unencrypted data transfer -->
    <!--
    <properties write_no_response="true" notify="true"/>
    -->

    <!-- Properties for encrypted data transfer -->
    <!--
    <properties
      write_no_response="true" encrypted_write="true"
      notify="true" encrypted_notify="true"/>
    -->

    <!-- Note: "user" type has to be specified; length is not needed -->
    <value type="user" />
  </characteristic>
</service>
```

Figure 10: GATT database entry for Cable Replacement service



The table below explains the Cable Replacement service entry in detail.

<pre>&lt;service uuid="f6ec37db-bda1-46ec-a43a-6d86de88561d" advertise="true"&gt;   &lt;description&gt;LE Cable Replacement Service&lt;/description&gt;   ... &lt;/service&gt;</pre>	<p>This defines the UUID of the service. Any 128 bit UUID may be chosen, as long as the client using the service is aware of it. The UUID shown here is for example only.</p> <p>The service definition contains inside it a data transfer characteristic definition which is described below.</p>
<pre>&lt;characteristic uuid="af20fbac251849989af7af42540731b3" id="xle_data"&gt;   &lt;description&gt;Data characteristic&lt;/description&gt;   &lt;properties write="true" encrypted_write="true" indicate="true" encrypted_notify="true" /&gt;   &lt;value type="user" /&gt; &lt;/characteristic&gt;</pre>	<p>The UUID for the characteristic used for data transfer can be any 128 bit UUID, as long as the client using the service is aware of it. The UUID shown here is for example only.</p> <p>Characteristic <b>id</b> attribute defines a symbolic name which can be used in BGScript for the characteristic.</p> <p>The characteristic properties define how the characteristic can be accessed. As described in Chapter 1.2.2, choice can be made between acknowledged and unacknowledged ATT methods. It is done as follows:</p> <ul style="list-style-type: none"> <li>• <b>write="true" indicate="true"</b>: acknowledged write requests and value indications are used for data transfer</li> <li>• <b>write_no_response="true" notify="true"</b>: write commands and value notifications which are not acknowledged are used for data transfer</li> </ul> <p>Both of the above specify plaintext data transfer. To protect the data from eavesdropping, add to the above one of the following:</p> <ul style="list-style-type: none"> <li>• <b>encrypted_write="true" encrypted_notify="true"</b>: requires encryption for data communication</li> <li>• <b>authenticated_write="true" authenticated_notify="true"</b>: requires both encryption and authentication (i.e., authenticated link key) for data communication</li> </ul> <p>The characteristic value must be as shown here and <b><u>must not be changed</u></b>.</p>

**Table 5: Cable Replacement service entry explained**

## 3.4 BGScript™ code

This section explains the most relevant sections of the BGScript™ code used in the cable replacement client and server applications and explains how the applications work.

### 3.4.1 Event handlers

BGScript applications are event-driven: calls to event handler procedures are triggered by various events such as scheduled timer expiry, hardware events, or when some communication with other devices takes place. Thus the application code is structured to respond to the events which are expected to happen during the application's runtime, and in the following the description of the application behavior concentrates on explaining the various event handlers.

Besides the event handlers, there are some helper procedures, but these are only used in the context of the event handlers, never individually.

### 3.4.2 Endpoints

Another central concept in the applications is the *endpoint*. An endpoint is an abstraction of a data source/sink, and in the example applications we use three of them:

- UART endpoint, which represents the BT121 serial interface
- Cable Replacement endpoint, which represents the Cable Replacement data channel
- Drop endpoint, which simply discards all data sent to it

While a script can write data directly to an endpoint, the example applications couple endpoints together by setting the UART endpoint to stream data to the Cable Replacement, and vice versa, when the Cable Replacement connection establishment is done. This frees the application from handling the data itself: it is done by the stack automatically.

### 3.4.3 Server application

In the following we walk through the events handled in the server application roughly in the order they are executed.

The application code can be found in the BGScript file **server.bgs**, in the same directory where the other project files are; for brevity it is not reproduced here.

#### 3.4.3.1 System boot

The **system\_boot** event is generated when power is applied to the *Bluetooth* module and it's the starting point for the code execution.

In the example Cable Replacement server application the system boot event handler initializes certain variables and configures the display hardware using the *display\_hardware\_init()* procedure.



No *Bluetooth* commands can be used in the boot event as the radio has not been initialized.

#### 3.4.3.2 System Initialized

Once the *Bluetooth* radio has been initialized the **system\_initialized** event is generated indicating that the radio is ready to be used. The *Bluetooth* address of the radio is given as an input parameter to the event handler.

In this event handler the code:

- Configures Security Manager:
  - a. Sets the device I/O capabilities for the Secure Simple Pairing (SSP) to **NoInputNoOutput** so that the **Just Works** pairing mechanism will be selected, avoiding the user to have to enter or compare a passkey, and
  - b. indicates that man-in-the-middle protection is not required; this and the above configuration are entered by *call sm\_configure(0,3)*
  - c. Additionally allows bonding attempts by *call sm\_set\_bondable\_mode(1)*

This is sufficient for demonstrating encrypted data transfer; note that in a more demanding setup the code would need to configure SM in a stricter fashion.

- Configures *Bluetooth* communication parameters:
  - a. *Bluetooth* LE default connection parameters are set by calling *le\_gap\_set\_conn\_parameters()* so that the shortest allowed connection interval is used. This increases throughput in a noisy environment as a broken connection event will not cause a long break in communication, at the expense of increased power usage.

- b. ATT MTU is set to the maximum allowed by calling *gatt\_set\_max\_mtu()*, to increase throughput as ATT headers incur less overhead proportionally, and more data can be sent in one ATT method/acknowledgement roundtrip. In order to use LE data packets optimally, the MTU should be a multiple of 27 bytes (the size of an LE data packet) minus 4 bytes needed for L2CAP layer headers. For instance, setting MTU to 50 would ensure that two full LE data packets would be sent per ATT request (as long as there is enough data).
- Prepares the DKBT display for writing by calling the *display\_set\_data\_mode()* procedure, and shows the device's own address on it by writing to the display buffer and calling the *display\_update()* procedure.
- Configures *endpoints*:
  - a. Configures the UART to drop incoming data (until a Cable Replacement connection is formed) by calling *endpoint\_set\_streaming\_destination()* with the destination set to 31, which is the ID of a special endpoint which drops any data sent to it
  - b. Calls the *le\_serial\_wait\_for\_connection()* procedure where a Cable Replacement server side endpoint waiting for incoming connections is set up with a *le\_serial\_listen()* call, and
- The device is made connectable and advertisement of the Cable Replacement service over *Bluetooth* is started in *le\_advertise()* procedure by calling *le\_gap\_set\_mode()*.

After the above setup completes the Server is ready to handle incoming Cable Replacement connections.

#### 3.4.3.3 LE connection opened

When a Client device opens a *Bluetooth* LE ACL connection, the **le\_connection\_opened** event is triggered. It is given various connection related parameters as input, but in this example the code merely shows the Client's *Bluetooth* address on the DKBT display.

#### 3.4.3.4 Cable Replacement connection opened

Once the Client device has performed the connection establishment as described in Chapter 1.2.3.1, the **le\_serial\_opened** event is triggered. At this point, the cable replacement endpoint which was created during system initialization changes state from listening to connected and becomes ready for data transfer.

This is reflected in the code by coupling of the UART and Cable Replacement endpoints, so that each feeds data to the other. This is done by calling *endpoint\_set\_streaming\_destination()* twice, once for UART to Cable Replacement streaming and once for the reverse direction. Also, the hardware display is updated to show the connection status.

After the endpoint coupling the data can be transparently relayed from the UART interface to the *Bluetooth* connection and vice versa; this is done by the *Bluetooth* Dual Mode stack and the BGScript application does not need to be involved in any way.

#### 3.4.3.5 Cable Replacement endpoint closing

Once the client disconnects the generic **endpoint\_closing** event is triggered for the Cable Replacement endpoint. At this point the following happens:

- The hardware display is updated to reflect the new state,
- The Cable Replacement endpoint is cleaned up by calling *endpoint\_close()*,
- The UART endpoint is configured to again drop data, and finally
- The *le\_serial\_wait\_for\_connection()* is called again, so that a new Cable Replacement server side endpoint is created and set to listen for the next incoming connection.

Note that you cannot recycle the old Cable Replacement endpoint for the next connection: it should be cleaned up (although stack does do that automatically, but after a delay) and a new one must be created in order to handle the next connection.

#### 3.4.3.6 LE Connection closing

If the client device closes also the LE ACL connection, **le\_connection\_closed** event will be generated. In this case, the remote client address is cleared from the hardware display and, more importantly, device is again made connectable and *Bluetooth* advertisement of the Cable Replacement service is restarted by setting the device's LE GAP mode with a call to *le\_advertise()* procedure.

#### 3.4.4 Client Application

In the following we walk through the events handled in the client application roughly in the order they are executed.

The application code can be found in the BGScript file **client.bgs**, in the same directory where the other project files are; for brevity it is not reproduced here.

##### 3.4.4.1 System boot

The **system\_boot** event handler is very similar to the server application's corresponding event handler.

##### 3.4.4.2 System Initialized

In this event handler the code:

- Configures Security Manage in the same way as the server application does
- Configures *Bluetooth* communication parameters in the same way as the server application does
- Prepares the DKBT display for writing, and shows the device's own address on it
- Configures the UART to drop incoming data (until a Cable Replacement connection is formed); note that the Cable Replacement endpoint is not yet created
- Calls the *le\_scan()* procedure to start scanning for advertising devices within range

After the above setup completes the Client will be looking for available servers by scanning.

##### 3.4.4.3 LE scanning results

Once the stack has discovered a device which advertises itself, it provides the scanning results to the application in the **le\_gap\_scan\_response** event. In this event handler the code looks for the Cable Replacement service UUID, and if it is found, stops scanning by calling *le\_gap\_end\_procedure()* and attempts to open a LE connection to the device with a *le\_gap\_open()* call.

Note that the scan response handler checks only the first service in the Service Class List advertisement data; this is sufficient for the demo, but a full application would need to check the complete list.

##### 3.4.4.4 LE connection opened

When the LE ACL connection is formed, the **le\_connection\_opened** event is triggered. In this example the code shows the Client's *Bluetooth* address on the DKBT display and then tries to set up connection encryption by calling *sm\_increase\_security()*. If the devices have not been previously bonded, this will result in a bonding attempt; in this example, since neither device requires man-in-the-middle protection and have set their I/O capabilities to NoInputNoOutput, the bonding should be done automatically.

#### 3.4.4.5 Bonding failure

In the case bonding fails for some reason, the event handler for **sm\_bonding\_failed** is called. There, the DKBT display is updated to show an error code and the LE connection is closed. However, this should not happen with the example applications.

#### 3.4.4.6 LE connection parameters notification

In the case bonding succeeds a **le\_connection\_parameters** event indicating current connection parameters will be sent by the stack, with the security parameter set to a non-zero value. This will notify the client application that the connection is now secure, and it will create its Cable Replacement endpoint in the event handler by calling *le\_serial\_open()*.

Initially the endpoint is not yet connected – that will happen after the connection establishment procedure has been done – so the endpoint is not yet coupled with the UART endpoint at this stage.

#### 3.4.4.7 Cable Replacement connection opened

Once the Client device has performed the connection establishment as described in Chapter 1.2.3.1, the **le\_serial\_opened** event is triggered as with the server. This works in the client the same way as it does in the server, and couples the Cable Replacement and the UART endpoints together.

After the endpoint coupling the data can be relayed between the serial connection and the *Bluetooth* connection by the *Bluetooth* stack; the BGScript application does not need to be involved in any way.

#### 3.4.4.8 Cable Replacement endpoint closing

In this example the client does not actively close the Cable Replacement endpoint, but it may happen, e.g., due to link loss. If this happens the generic **endpoint\_closing** event is triggered for the Cable Replacement endpoint. At this point the following happens:

- The hardware display is updated to reflect the new state,
- The Cable Replacement endpoint is cleaned up,
- The UART endpoint is configured to again drop data

#### 3.4.4.9 LE Connection closing

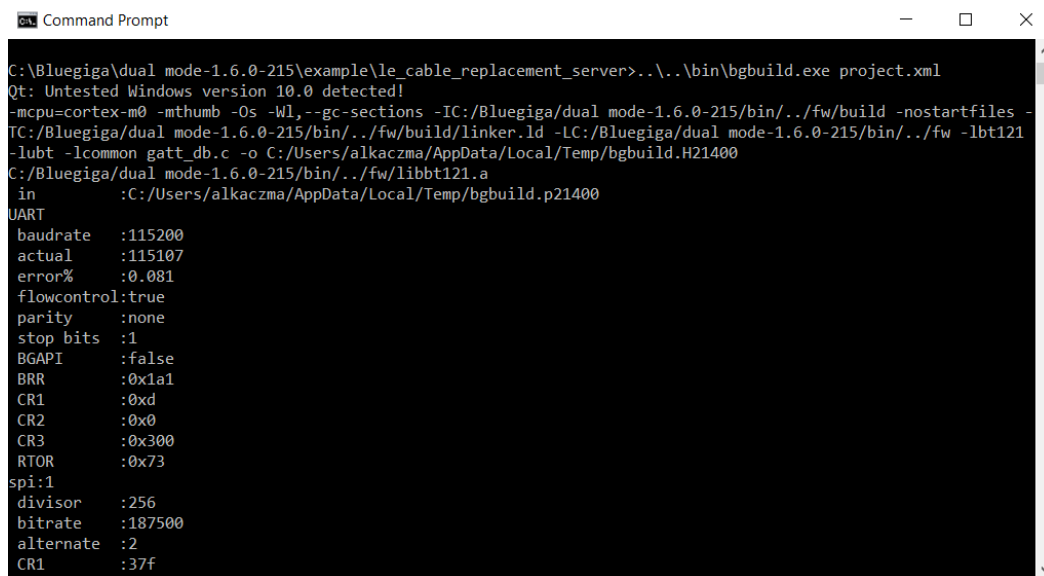
If the LE ACL connection is lost, **le\_connection\_closed** event will be generated. In this case, the server address is cleared from the hardware display and, more importantly, device restarts scanning by calling *le\_scan()* again.

### 3.5 Compiling the applications

The server example application can be compiled with the BGBuild compiler as follows:

- Open a command prompt (cmd.exe)
- Change the working directory to <SDK installation directory>\example\le\_cable\_replacement\_server
- Run bgbuild.exe with the project XML file as an argument: ..\..\bin\bgbuild.exe project.xml

The client example application can be compiled in the same manner in the directory containing the client application project (<SDK installation directory>\example\le\_cable\_replacement\_client).

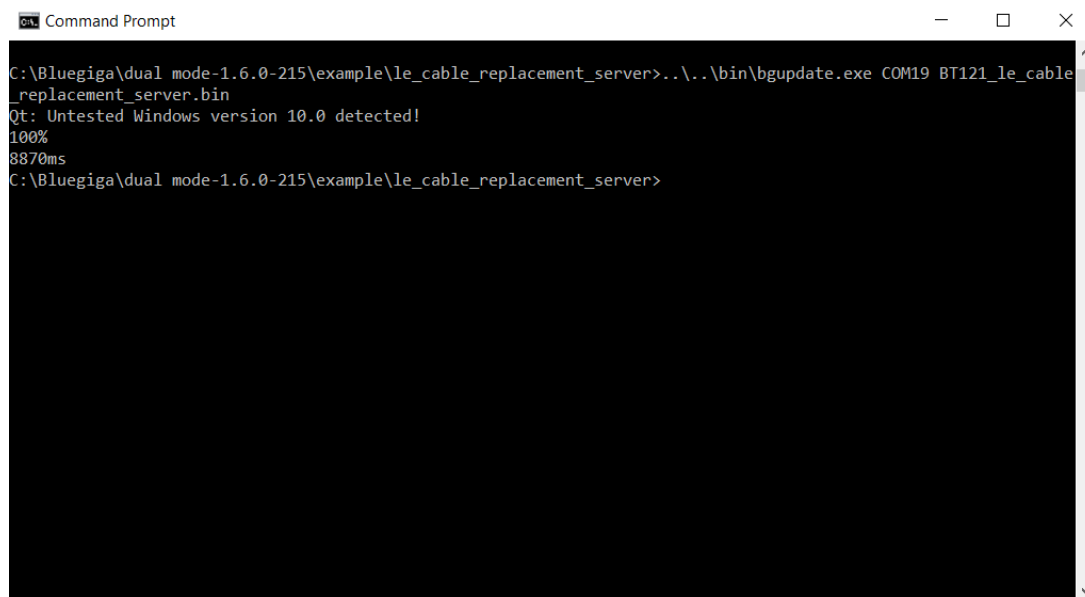


```
C:\Bluegiga\dual mode-1.6.0-215\example\le_cable_replacement_server>..\bin\bgbuild.exe project.xml
Qt: Untested Windows version 10.0 detected!
-mcpu=cortex-m0 -mthumb -Os -Wl,--gc-sections -IC:/Bluegiga/dual mode-1.6.0-215/bin/./fw/build -nostartfiles -
TC:/Bluegiga/dual mode-1.6.0-215/bin/./fw/build/linker.ld -LC:/Bluegiga/dual mode-1.6.0-215/bin/./fw -lbt121
-lubt -lcommon gatt_db.c -o C:/Users/alkaczma/AppData/Local/Temp/bgbuild.H21400
C:/Bluegiga/dual mode-1.6.0-215/bin/./fw/libbt121.a
in :C:/Users/alkaczma/AppData/Local/Temp/bgbuild.p21400
UART
baudrate :115200
actual :115107
error% :0.081
flowcontrol:true
parity :none
stop bits :1
BGAPI :false
BRR :0x1a1
CR1 :0xd
CR2 :0x0
CR3 :0x300
RTOR :0x73
spi:1
divisor :256
bitrate :187500
alternate :2
CR1 :37f
```

Figure 11: Compiling the server example

## 3.6 Installing the firmware

The firmware binary can be installed to the DKBT development kit with the BGUpdate tool. It will require the COM port you have connected the DKBT to and the firmware image as arguments. Do this once for each device, programming one with the firmware containing the server application and the other with the firmware containing the client application.



```
C:\Bluegiga\dual mode-1.6.0-215\example\le_cable_replacement_server>..\bin\bgupdate.exe COM19 BT121_le_cable
replacement_server.bin
Qt: Untested Windows version 10.0 detected!
100%
8870ms
C:\Bluegiga\dual mode-1.6.0-215\example\le_cable_replacement_server>
```

Figure 12: Firmware update with BGUpdate

## 3.7 BGTool

Alternatively, you can use the graphical BGTool for building and installing the firmware. Note however that as the applications assign the BT121 UART for serial data, BGAPI is disabled in the hardware configuration and you cannot use BGTool to give BGAPI commands to the devices after programming them.

## 3.8 Testing the applications

Once both devices have been programmed, they should connect automatically and show the connection status on their respective displays.

If you have already connected the Server device to an actual serial data source and there is, e.g., a PC application for communicating with that data source, you can connect the Client device to a PC and configure the application to use the serial port the Client device is connected to.

Alternatively, you can for instance connect both devices to a PC using the USB serial converters present on the DKBT main boards and use a terminal emulation application such as TeraTerm on both devices' serial ports to verify data sent from one appears on the other.

# Smart. Connected. Energy-Friendly.



**IoT Portfolio**  
[www.silabs.com/products](http://www.silabs.com/products)



**Quality**  
[www.silabs.com/quality](http://www.silabs.com/quality)



**Support & Community**  
[www.silabs.com/community](http://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.

**Note: This content may contain offensive terminology that is now obsolete. Silicon Labs is replacing these terms with inclusive language wherever possible. For more information, visit [www.silabs.com/about-us/inclusive-lexicon-project](http://www.silabs.com/about-us/inclusive-lexicon-project)**

## 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

[www.silabs.com](http://www.silabs.com)