

BLUETOOTH DUAL MODE SOFTWARE

GETTING STARTED GUIDE

Friday, 18 June 2021

Document Revision: 1.5



VERSION HISTORY

Date Edited	Comment
1.0	First version
1.1	Minor changes
1.2	Updated to match software release 1.1.1 build 168
1.3	Bluetooth stack technical features added
1.4	Renamed "Bluetooth Smart Ready" to "Bluetooth Dual Mode", "Bluetooth Smart" to "Bluetooth Low Energy" and "Classic" to "BR/EDR" according to the official Bluetooth SIG nomenclature
1.5	Replaced inappropriate terms in accordance with the latest Bluetooth SIG recommendations.

TABLE OF CONTENTS

1	Introduction	4
2	The Bluegiga <i>Bluetooth</i> Dual Mode software	5
2.1	The Bluegiga <i>Bluetooth</i> Dual Mode stack	5
2.2	Bluetooth Stack Features	6
2.3	The Bluegiga Bluetooth Dual Mode SDK	7
2.4	The BGAPI™ serial protocol API	8
2.5	The BGLIB™ host library API	9
2.6	BGScript™ scripting language API	10
2.7	BGAPI™ vs. BGScript™	11
2.8	Profile Toolkit™	12
2.9	BGBuild compiler	13
2.10	DFU tools	13
2.11	BGTool test application	13
3	Factory default configurations	15
3.1	<i>Bluetooth</i> module factory configuration	15
3.2	Development kit factory configuration	15
4	Getting started with the <i>Bluetooth</i> Dual Mode Software	16
4.1	Installing the <i>Bluetooth</i> Dual Mode SDK	16
4.2	Folder structure	17
4.3	Included tools	17
4.4	Built-in demos in the SDK	18
5	Walkthrough of the BGDemo application	19
5.1	Project configuration	20
5.2	Hardware configuration	21
5.3	<i>Bluetooth</i> services configuration	21
5.4	BGScript code	29
5.5	Compiling BGDemo application	33
5.6	Installing the firmware	34
5.7	Testing BGDemo application	34

1 Introduction

This document explains the architecture and the APIs of the Bluegiga *Bluetooth* Dual Mode Software as well the tools and components included in the *Bluetooth* Dual Mode Software Development Kit.

This document also walks you through the factory demo application preinstalled in the DKBT *Bluetooth* Dual Mode development kit which is also included as one of the demo applications in the SDK.

2 The Bluegiga *Bluetooth* Dual Mode software

This section contains a short description of the Bluegiga *Bluetooth* Dual Mode software, the components, APIs and the tools it includes.

2.1 The Bluegiga *Bluetooth* Dual Mode stack

The main components of the Bluegiga *Bluetooth* Dual Mode and stack are shown in the figure below. The figure shows the layers the *Bluetooth* Dual Mode stack as well also shows the APIs that can be used to interface to the Bluegiga *Bluetooth* Dual Mode Stack.

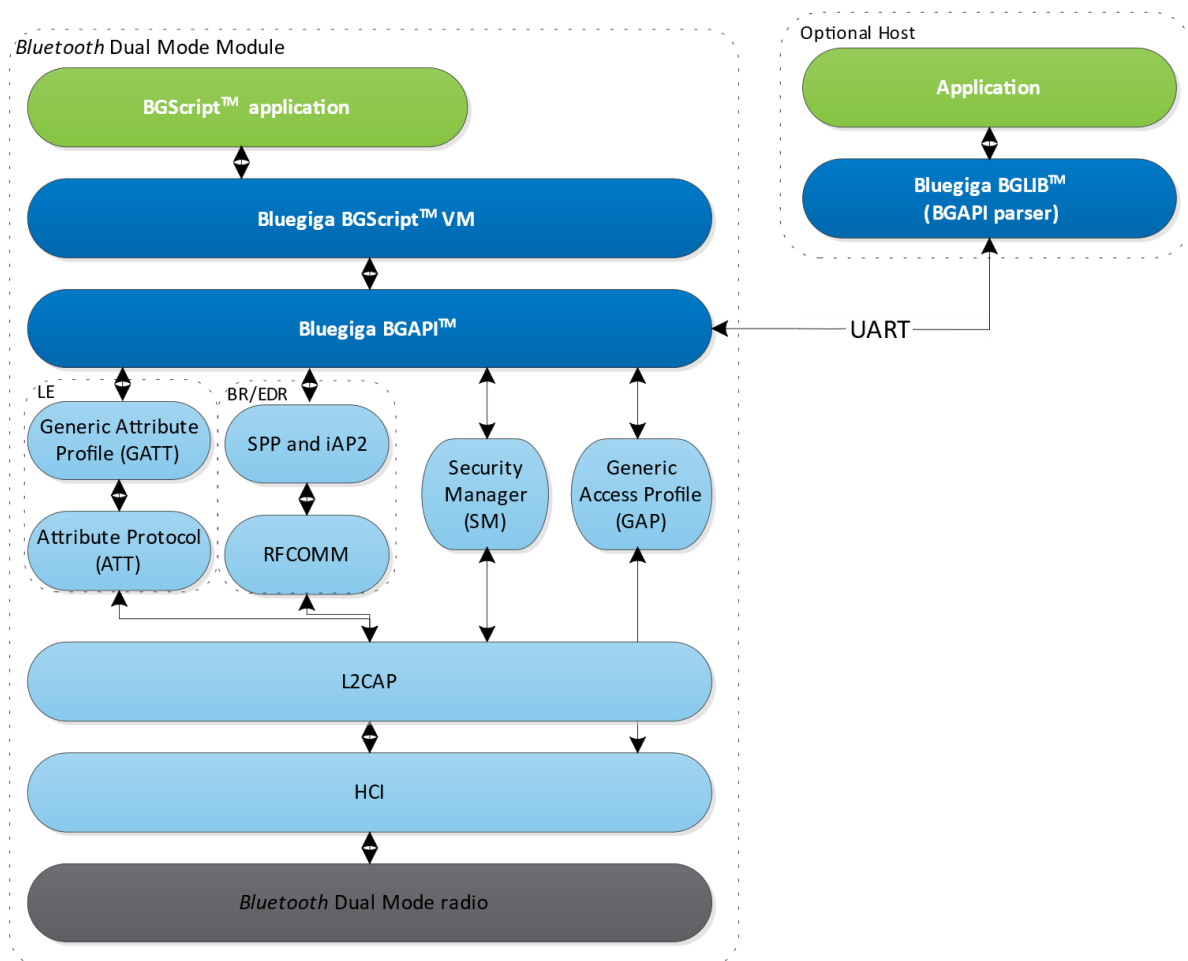


Figure 1: Bluegiga *Bluetooth* Dual Mode Stack and APIs

2.2 Bluetooth Stack Features

Feature	Value
Simultaneous SPP/HID connections	up to 6
Simultaneous SCO connections	0
Simultaneous LE connections	Up to 7
Simultaneous Central / Peripheral	Supported for BR/EDR Not supported for LE BR/EDR and LE connections are independent from each other
Combined BR/EDR and LE connections	Any combination of connections up to 6
Max data rate	1000 kbps over SPP (transparent UART mode) 700 kbps over SPP (BGAPI mode) 200 kbps (iAP2)
Bluetooth security features	Just works for BR/EDR and LE Man-in-the-Middle protection for BR/EDR and LE Legacy Pin code pairing for BR/EDR Out-of-Band pairing for LE
Max bondings	12
Supported Bluetooth BR/EDR profiles	GAP, SPP, DI, GATT over BR, HID device
Supported Bluetooth GATT profiles	Any
Apple iAP support	iAP2
Bluetooth QDIDs	Controller: 58852 Host: 70031 HID over BR: 90001

2.3 The Bluegiga Bluetooth Dual Mode SDK

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

The *Bluetooth* Dual Mode SDK supports multiple development models and the application 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* Dual Mode modules.

The SDK also contains documentation, tools for compiling the firmware, installing it into the hardware and lot of example applications speeding up the development process.

The Bluegiga *Bluetooth* Dual Mode SDK includes the following APIs, components and tools:

- **The *Bluetooth* Dual Mode stack** as described in the previous chapter.
- **BGAPI™** is a binary serial protocol API to the Bluegiga *Bluetooth* Dual Mode stack over UART interfaces. BGAPI is target for users, who want to use both *Bluetooth* BR/EDR and LE functionality and use all the features in the *Bluetooth* Dual Mode stack from an external host such as a low power MCU.
- **BGLIB™** is a host library for external MCUs and implements a reference parser for the BGAPI serial protocol. BGLIB is delivered in C source code as part of the *Bluetooth* Dual Mode SDK and it can be easily ported to various processor architectures.
- **BGScript™** interpreter and scripting language allow applications to be developed into the *Bluetooth* Dual Mode modules built-in MCU. It allows simple end user applications or enhanced functionality to be developed directly into the *Bluetooth* Dual Mode module which means no external host MCU is necessarily needed. BGScript applications can be executed at the same time as the BGAPI is used, allowing the possibility to implement some functionality on the *Bluetooth* module and some on the host.
- **Profile Toolkit™** is a simple XML based description language which can be used to easily and quickly develop GATT based service and characteristic databases for the *Bluetooth* Low Energy module.
- **BGBuild compiler** is a free-of-charge compiler that compiles the *Bluetooth* Dual Mode Stack, the BGScript application and the *Bluetooth* GATT services into the firmware binary that can be installed to the *Bluetooth* Dual Mode modules.
- **BGTool** is a graphical user interface application and a developer tool, which allows the *Bluetooth* Dual Mode module to be controller over the host interface using the BGAPI serial protocol. BGTool is a useful tool for testing the *Bluetooth* Dual Mode module and evaluating it's functionality and APIs.
- **DFU Tools** are also included as part of the SDK allowing the firmware to be updated over the UART interfaces.

2.4 The BGAPI™ serial protocol API

The BGAPI is a serial protocol allows external hosts to interface to the *Bluetooth* Dual Mode modules over UART interfaces. The BGAPI serial protocol is a lightweight and well-defined binary protocol which allows command and data to be sent to the *Bluetooth* Dual Mode module and it also provides a way to receive responses and events and data from the module.

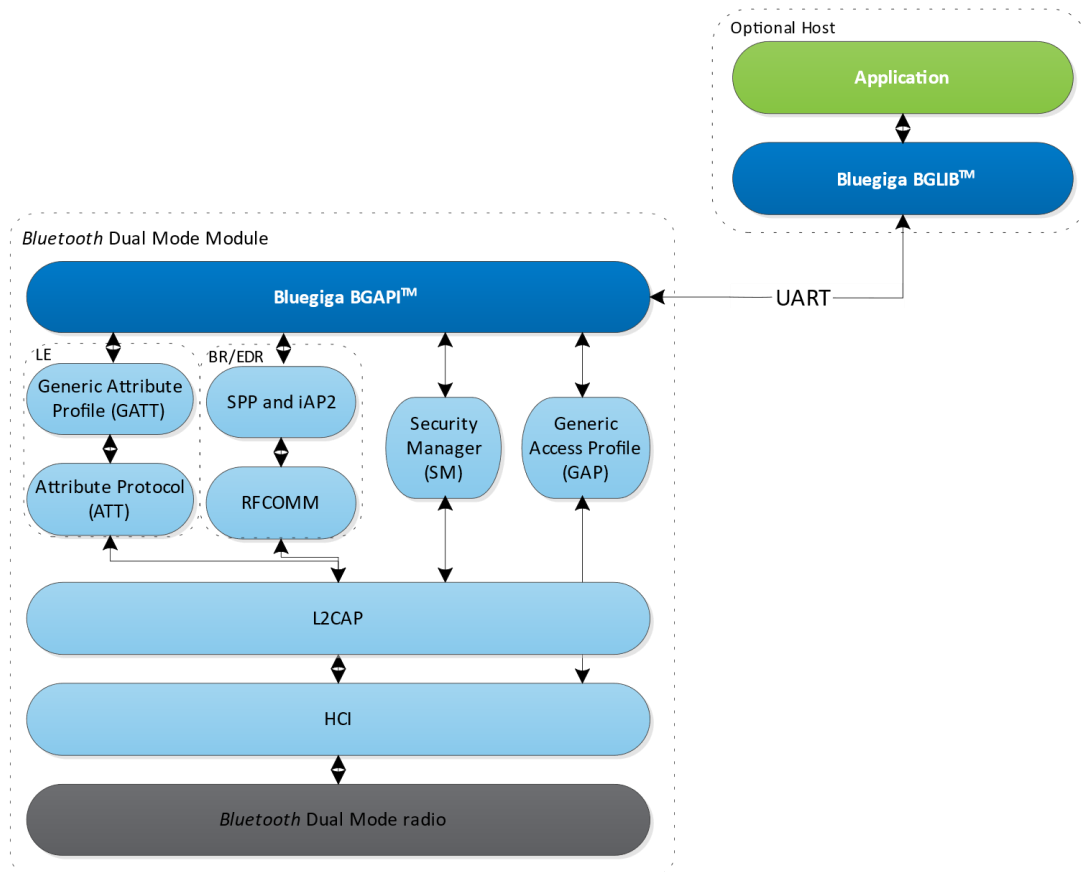


Figure 2: Architecture when using BGAPI and BGLIB

The BGAPI serial protocol provides access for example to the following layers in the *Bluetooth* Dual Mode stack:

- **BT GAP** – BT GAP provides access to basic *Bluetooth* BR/EDR features, like device discovery and connection establishment.
- **LE GAP** - LE GAP provides access to basic *Bluetooth* LE features, like device advertisement, discovery and connection establishment.
- **Security manager** – Security manager is used to configure the local devices security features and establish secure connections
- **RFCOMM** – RFCOMM provides basic serial data transmission over *Bluetooth* RD/EDR.
- **iAP** – iAP provides basic serial data transmission to Apple iOS devices using *Bluetooth* BR/EDR. iAP is only available to Apple MFI licenses and not included in the standard SDK.
- **Hardware** – Access to local hardware features and interfaces like SPI, I2C, GPIO and ADC
- **Persistent Store** - A data storage that allows data to be stored to and read from the internal flash
- **System** - Local device's status and management functions

2.5 The BGLIB™ host library API

BGLIB™ host library is a reference implementation of the BGAPI serial protocol parser, and it's provided in an ANSI C source code in the *Bluetooth Dual Mode SDK*.

BGLIB host library abstracts the complexity of the BGAPI serial protocol and provides instead high level C functions and call-back handlers to the application developer, which makes the application development easier and faster.

BGLIB library can be ported to various host systems ranging from low cost MCUs to devices running Linux, Windows or OSX.

```
/* Function to open RFCOMM connection */
void dumo_cmd_bt_rfcomm_open(    bd_addr address,
                                uint8 streaming_destination,
                                uint8 uuid_len,
                                const uint8* uuid_data);

/* Callback handler*/
struct dumo_msg_bt_rfcomm_open_rsp_t
{
    uint16 result,
    uint8 endpoint
}

/* Response handler */
void dumo_rsp_bt_rfcomm_open(
    const struct dumo_msg_bt_rfcomm_open_rsp_t *msg
)
```

Figure 3: BGLIB API example

2.6 BGScript™ scripting language API

BGScript is a simple BASIC-style programming language that allows end-user applications to be embedded to the Bluegiga *Bluetooth* Dual Mode modules. Although being a simple and easy-to-learn programming language BGScript does provide features and functions to create fairly complex and powerful applications and it provides access to all the same APIs as the BGAPI serial protocol.

BGScript is fully event based programming language and code execution is started when events such as system start-up, *Bluetooth* connection, I/O interrupt etc. occur.

BGScript applications are developed with Bluegiga's free-of-charge *Bluetooth* Dual Mode SDK and the BGScript applications are executed in the BGScript Virtual Machine (VM) that is part of the Bluegiga *Bluetooth* Dual Mode software. The *Bluetooth* Dual Mode SDK comes with all the necessary tools for code editing and compilation and also the needed tools for installing the compiled firmware binary to the Bluegiga *Bluetooth* Dual Mode modules.

BGScript applications can also be used at the same time as BGAPI and they can be for example used to automate some simple actions.

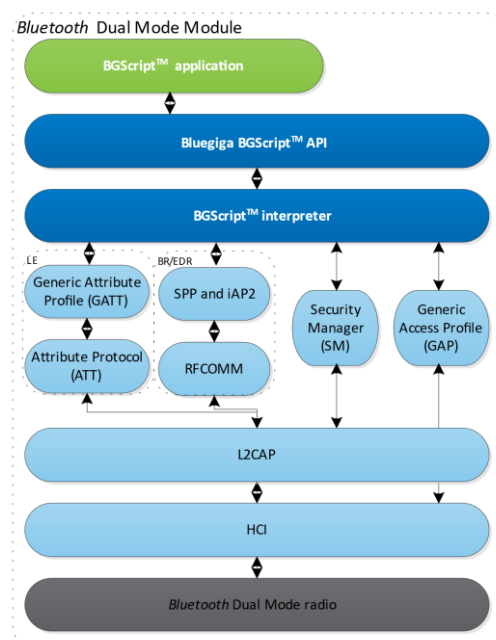


Figure 4: Bluegiga BGScript

```
# System boot event listener - this is executed on module power on
event system_boot(major,minor,patch,build,bootloader,hw)

    # Start a software timer
    call hardware_set_soft_timer(1000,0,0)
end

# System initialized event - generated when the hardware is ready
# to be used
event system_initialized(addr)

    # Start RFCOMM server for SPP
    call bt_rfcomm_start_server(5,2,0)

    # Make device visible via both BT BR/EDR and LE
    call bt_gap_set_mode(1,1,0)
    call le_gap_set_le_mode(2,2)

    #Set local devices friendly name
    call system_set_local_name(21,"Bluegiga BT121 bgdemo")
end
```

Figure 5: A simple BGScript code example

2.7 BGAPI™ vs. BGScript™

This section describes the differences between using BGAPI and BGScript. In brief the difference is:

- BGScript is our custom scripting language used for on-module applications. BGScript applications only run on Bluegiga modules and dongles.
- BGAPI is a custom serial protocol used to externally control the modules over the host interface and BGLIB is an ANSI C reference implementation of the BGAPI serial protocol and only runs outside of our modules and dongles.

So the main difference between BGScript and BGLIB is that BGScript allows you to run an application right on the Bluetooth module, whereas BGLIB uses the BGAPI serial protocol API to send commands and receive events from an external device - typically a microcontroller. Note however that BGScript and BGLIB implement the exact same functionality. Every BGScript command, response, and event has a counterpart in the BGAPI serial protocol and BGLIB host library.

One other thing to keep in mind is that BGScript has some performance penalties compared to external API control due to the fact that BGScript is an interpreted scripting language and requires extra overhead in order to use. It makes the Bluetooth module do the work that could otherwise be done elsewhere. If you are trying to achieve maximum performance or you have an application which is fairly complex (lots of fast timers, interrupts, or communicating with many external sensors over I2C or SPI for example), it is often a good idea to use a small external microcontroller and BGLIB/BGAPI instead.

Question	BGAPI™	BGScript™
An external host needed?	Yes	No
Host interface	UART	No separate host needed
<i>Bluetooth</i> API	BGAPI serial protocol or BGLIB host API	BGScript API
Peripheral interface APIs and support	Host dependent (*)	APIs for UART, SPI, I2C, GPIO, ADC and PS store
Custom peripheral interface drivers	Can be developed to the host	Peripheral drivers are part of the Bluegiga <i>Bluetooth</i> Dual Mode stack
RAM available for application	Host dependent	Application dependent
Flash available for application	Host dependent (**)	~12 kB
Execution speed	Host dependent	Application dependent
Application development SDK	Host dependent + BGAPI and BGLIB	Bluegiga <i>Bluetooth</i> Dual Mode SDK
<i>Bluetooth</i> firmware / application updates	DFU over UART	DFU over UART

Figure 6: BGAPI vs. BGScript

*) The *Bluetooth* Dual Mode modules peripheral interfaces are still available via BGAPI commands and can be used to extend the host MCUs I/Os.

2.8 Profile Toolkit™

The *Bluetooth* Low Energy profile toolkit is a simple set of tools, which can be used to describe GATT based service and characteristic databases used with *Bluetooth* Low Energy and GATT over BR profiles. The profile toolkit consists of a simple XML based description language and templates, which can be used to describe the services and characteristics and their properties in a device's GATT database.

The GATT database developed with the Profile Toolkit is included as part of the device's firmware when the firmware is compiled.

```
<gatt>

  <service uuid="1800">

    <description>Generic Access Service</description>

    <characteristic uuid="2a00">
      <properties read="true" const="true" />
      <value>BT121 Bluetooth SR</value>
    </characteristic>

    <characteristic uuid="2a01">
      <properties read="true" const="true" />
      <value type="hex">0</value>
    </characteristic>

  </service>

</gatt>
```

Figure 7: Profile Toolkit example of GAP service and characteristics

2.9 BGBuild compiler

BGBUILD compiler is a simple compiler that is used to build a firmware images for the *Bluetooth* Dual Mode modules. The BGBuild compiler compiles the *Bluetooth* Dual Mode stack, the GATT database and optionally also a BGScript application into a single firmware binary image that can be installed into a *Bluetooth* Dual Mode module.

2.10 DFU tools

The Device Firmware Update (DFU) protocol is an open serial protocol that can be used to perform field updates to the *Bluetooth* Dual Mode modules. DFU protocol allows any firmware image generated with the BGBuild compiler to be installed into a *Bluetooth* Dual Mode Module.

The *Bluetooth* Dual Mode SDK contains command line binaries versions and source code for the DFU protocol and tools.

DFU protocol and available commands are described in the API reference document.

2.11 BGTool test application

BGTool application can be used to test and evaluate the *Bluetooth* Dual Mode module and issue BGAPI commands to it over UART host interface. It's a useful tool for testing the *Bluetooth* module and its functionality.



Figure 8: BGTool application

3 Factory default configurations

Below are short descriptions what are the default configurations in the modules and development kits are.

3.1 *Bluetooth* module factory configuration

When the modules are delivered from the factory they have the following configuration preinstalled:

- BGAPI serial protocol over UART interface
- UART baud rate: 115200
- Hardware flow control: enabled
- Data bits: 8
- Parity: none
- Stop bits: 1

3.2 Development kit factory configuration

When the DKBT development kits are delivered from the factory they have the following configuration preinstalled:

- Built in demo application developed with BGScript scripting language
- BGAPI serial protocol disabled
- UART is enabled and DFU updates

4 Getting started with the *Bluetooth* Dual Mode Software

4.1 Installing the *Bluetooth* Dual Mode SDK

In order to install the *Bluetooth* Dual Mode SDK please perform the following steps:

1. Go to https://siliconlabs.force.com/apex/SL_CommunitiesSelfReg and create yourself an account if you do not one already
2. Go to https://www.silabs.com/support/resources.p-wireless_bluetooth-classic_bluegiga-legacy-modules_bt121-a and download the *Bluetooth Dual Mode Software and SDK*
3. Unzip the SDK and start the installer EXE
4. Follow the on-screen instructions to install the SDK

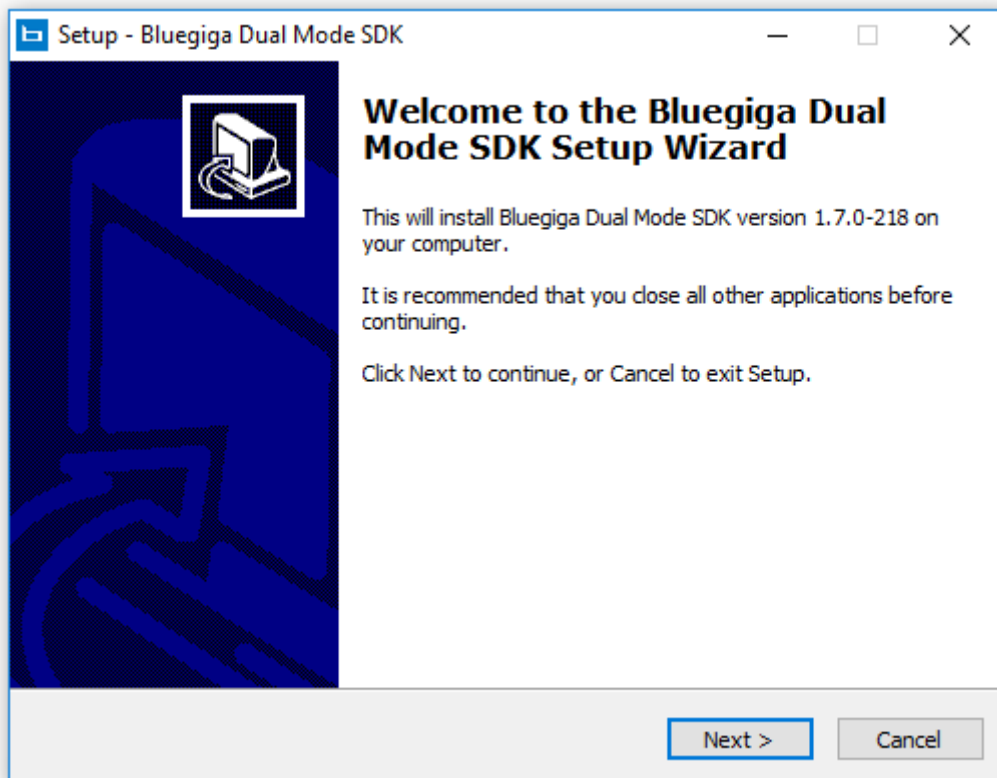


Figure 9: SDK Installer

4.2 Folder structure

The SDK creates the following folders into the installation directory.

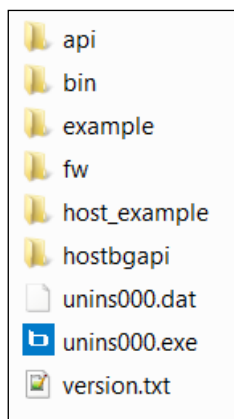


Figure 10: SDK folders

- **API** This folder includes a raw XML description of the *Bluetooth* stack's API. This file is not necessarily needed for anything, but could be used for example to automatically generate parsers for the API.
- **HOSTBGAPI** This folder includes a HTML file describing the API as well the BGLIB parsers .H file, which needs to be included in the projects implementing BGAPI library.
- **HOSTBGEXAMPLE** This folder contains code examples for external hosts and they use the BGAPI serial protocol and the BGLIB library.
- **BIN** This folder includes all the binary applications needed by the SDK.
- **EXAMPLE** This folder includes the BGScript demo applications, profile toolkit examples and the Bluetooth module configuration examples, which all generate a firmware for the *Bluetooth* module if run through the BGBuild compiler
- **FW** This folder contains the actual Bluetooth Dual Mode stack firmware binaries.

4.3 Included tools

The following tools are installed by the SDK:

- **BGTool.exe** A graphical UI tool that can be used to control the *Bluetooth* Dual Mode module over UART/RS232 using the BGAPI serial protocol. The tool is useful for quickly trying the features and capabilities of the *Bluetooth* Dual Mode software, without writing any software.
- **BGBuild.exe** BGBUILD compiler is a simple compiler that is used to build a firmware images for the Bluetooth Dual Mode modules.
- **BGUpdate.exe** A command line tool implementing DFU protocol over UART/RS232 and which can be used to update the firmware to a *Bluetooth* Dual Mode module

4.4 Built-in demos in the SDK

There are various example and demos included in the SDK and they also get updated with the newer version of the SDK. At the time of the release the following examples are included in the SDK:

4.4.1 BT121

This is the factory default firmware and configuration for the BT121 *Bluetooth* Dual Mode modules. This means that the modules when delivered from the factory have this firmware in them.

The configuration of the module is as defined in chapter 3.1.

4.4.2 BGDemo

BGDemo is the application pre-installed demo application on the DKBT development kit when they are shipped from the factory. It will make the *Bluetooth* module visible to both *Bluetooth* BR/EDR and Low Energy devices and it can be connected from both.

The demo will expose the temperature readings from the I2C altimeter via the *Bluetooth* Low Energy connection using the Health Thermometer Profile. The demo enables also serial data exchange over *Bluetooth* Serial Port Profile (SPP).

4.4.3 iAPDemo

This application demonstrates Apple iAP profile functionality on the DKBT development kit. It will make the *Bluetooth* module visible to both Apple iOS device and *Bluetooth* Low Energy devices and it can be connected from both.

The demo will expose the temperature readings from the I2C altimeter via the *Bluetooth* Low Energy connection using the Health Thermometer Profile. The demo enables also data exchange over the Apple iAP profile for iOS devices with an iAP capable application.

iAPDemo is not included in the standard *Bluetooth* Dual Mode SDK since iAP is only available for Apple MFI licenses and requires a version of the *Bluetooth* Dual Mode SDK with iAP add-ons.

For more information please contact us.

5 Walkthrough of the BGDemo application

This section walks you through the demo application that is pre-installed on the DKBT development kits when they are shipped from the factory. The purpose of the section is to give you first overview of the *Bluetooth* Dual Mode SDK and how you can start building your own applications.

BGDemo is the application pre-installed demo application on the DKBT development kit when they are shipped from the factory. It will make the *Bluetooth* module visible to both *Bluetooth* BR/EDR and Low Energy devices and it can be connected from both.

The demo will expose the temperature readings from the I2C altimeter via the *Bluetooth* Low Energy connection using the Health Thermometer Profile. The demo enables also serial data exchange over *Bluetooth* Serial Port Profile (SPP).

The figure below illustrates the BGDemo application architecture.

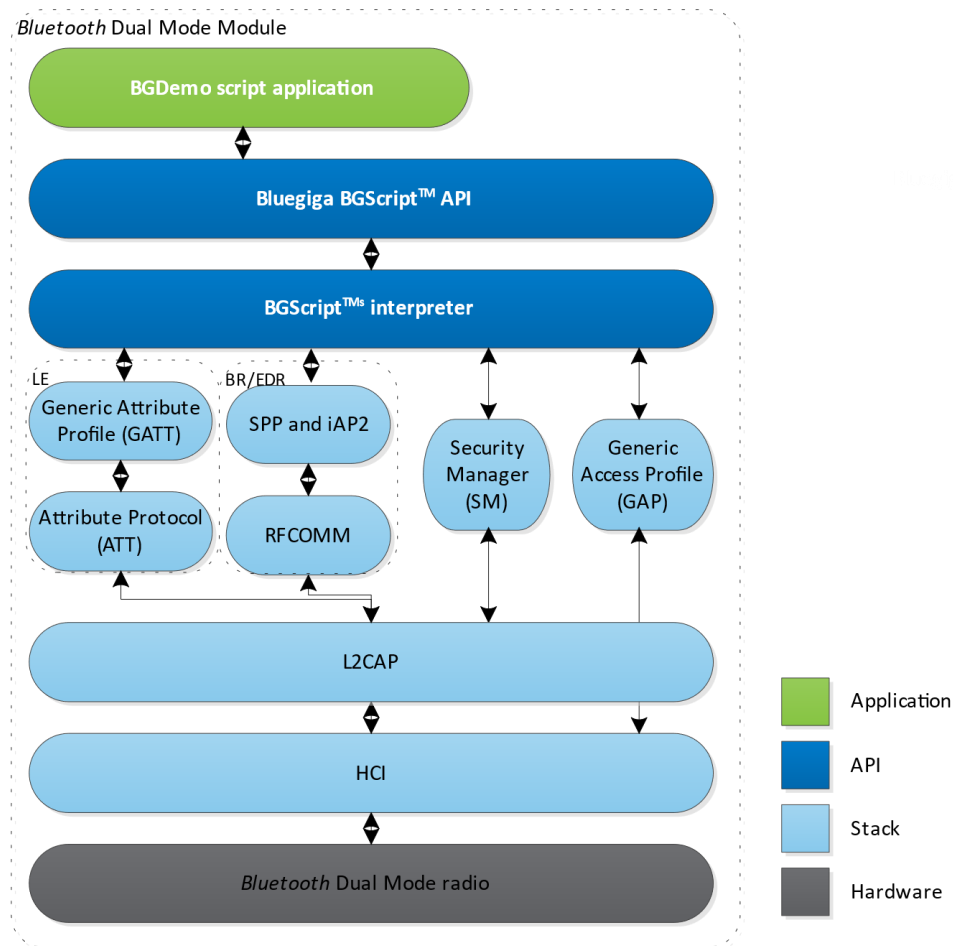


Figure 11: BGDemo architecture

5.1 Project configuration

Building a *Bluetooth* Dual Mode project starts always by making a project file, which is a simple XML file which defines the resources used in the project.

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

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

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

  <!-- Local SDP entries for Bluetooth BR/EDR -->
  <sdp>
    <entry file="DID.xml" autoload="true"/>
    <entry file="SPP.xml" id="2"/>
  </sdp>

  <!-- BGScript source code file -->
  <scripting>
    <script in="bgdemo.bgs" />
  </scripting>

  <!-- Firmware output files -->
  <image out="BT121_BGDemo.bin" out_hex="BT121_BGDemo.hex" />

</project>
```

Figure 12: Project file

<code><project device="bt121"></code>	This tag starts the project definition and the project file must end in <code></project></code> tag. The <code>device</code> is used to define for which <i>Bluetooth</i> module the project is used for.
<code><gatt in="gatt.xml" /></code>	The <code><gatt></code> tag is used to define the XML file containing the GATT service and characteristic database used for BLE and GATT over BR profile.
<code><hardware in="hardware.xml" /></code>	The <code><hardware></code> tag define which file contains the hardware configuration for interfaces like UART, SPI or I ² C.
<code><sdp></code> <code> <entry file="DID.xml" autoload="true"/></code> <code> <entry file="SPP.xml" id="2"/></code> <code></sdp></code>	The <code><sdp></code> tag is used to define the Bluetooth BR/EDR Service Discover Profile (SDP) entries or <i>Bluetooth</i> profiles used by the project. The <code><entry></code> tags are used to define the actual XML files for each SDP entry and also unique IDs for each SDP entry which can later on be used in the BGScript code to load the SDP entries.
<code><scripting></code> <code> <script in="bgdemo.bgs" /></code> <code></scripting></code>	Inside the <code><scripting></code> tags all the included BGScript code file(s) are defined. Individual script file(s) need to be defined with the <code><script></code> tags.
<code><image out="BT121_BGDemo.bin" /></code>	The <code><image></code> tag defines the name of the BGBuild compiler output file. The generated .bin file contains the <i>Bluetooth</i> Dual Mode stack, the GATT database, the SDP entries, the hardware configuration and the BGScript code.



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

5.2 Hardware configuration

The next logical step is to define the hardware configuration of the *Bluetooth* module and defined which hardware interfaces are enabled and what are their default configuration.

```
<hardware>

  <!-- Sleep modes enabled -->
  <sleep enabled="true"/>

  <!-- UART enabled @ 115200bps. BGAPI disabled as this is a BGScript project -->
  <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"/>

  <!-- I2C enabled for altimeter -->
  <i2c channel="1" pullup="true" />

</hardware>
```

Figure 13: Hardware configuration

<sleep enabled="true" />	This tag defines whether the MCU sleep modes (low power modes) are enabled or now.
<uart baud="115200" flowcontrol="true" bgapi="false" />	The <uart> tag is used to define if the UART interface is enabled or not as well the UART interface settings. In the BGDemo example project UART is used for SPP data exchange and status messages. BGAPI serial protocol is disabled as this example uses a BGScript application to implement the application logic.
<spi channel="1" alternative="2" clock_idle_polarity="high" clock_edge="1" endianness="msb" divisor="256" />	The <spi> tag defines if the SPI interface is enabled or not as well the used SPI interface (two available) and it's settings. In the BGDemo example SPI interface is used to communicate with the display on the DKBT development kit.
<port index="0" output="0x4000" /> <port index="1" output="0x4000" />	The <port> 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 BGDemo example 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.
<i2c channel="1" pullup="true"/>	The <i2c> tag is used to enable and configure the I2C interface. In the BGDemo example the I2C is used to communicate with the altimeter on the DKBT development kit.



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

5.3 Bluetooth services configuration

Next step is to configure the *Bluetooth* GATT services and *Bluetooth* BR/EDR profiles used by the device. This is again done by editing the XML files you have defined in the project file.

5.3.1 *Bluetooth* GATT database

The GATT database defines the services and characteristics, which are used for both *Bluetooth* Low Energy and GATT over BR profiles and it defines the structure and presentation format of the data exposed by the device.

The GATT database is again an XML encoded file built with the *Bluetooth* Low Energy profile toolkit which when is compiled with the BGBuild compiler as part of the firmware.

```
<!-- Generic Access Service -->
<service uuid="1800">

    <description>Generic Access Service</description>

    <!-- Device name -->
    <characteristic uuid="2a00">
        <properties read="true" const="true" />
        <value>BT121 HTM+SPP Demo</value>
    </characteristic>

    <!-- Appearance -->
    <characteristic uuid="2a01">
        <properties read="true" const="true" />
        <value type="hex">0768</value>
    </characteristic>

</service>
```

Figure 14: GAP service



The figure above shows only partial code. See the BGScript file for more information.

The [Figure 14](#) shows part of the GATT database used in BGDemo and how it looks like in the Profile Toolkit. The figure just contains the GAP service and the explanation of the syntax can be found below.

<pre><service uuid="1800" /></pre>	<p>The <code><service></code> tag is used to define a start of a GATT service. The UUID defines the 16-bit or 128-bit UUID used by the service.</p> <p>In this case UUID 1800 refers to the GAP service defined by the <i>Bluetooth</i> SIG and details of which can be found from <i>Bluetooth</i> developer site here.</p>
<pre><description> Generic Access Service </description></pre>	<p>The <code><description></code> tag is just an informative definition in the GATT file and is not used for anything other than just a comment describing what the service is.</p>
<pre><characteristic uuid="2a00"> <properties read="true" const="true" /> <value>BT121 HTM+SPP Demo</value> </characteristic></pre>	<p>The <code><characteristics></code> tag starts the definition of a characteristic, so the actual data exposed by the device. Again the characteristic UUID must be defined and every characteristic needs to have a unique 16-bit or 128-bit UUID. In this case 2a00 refers to device name characteristic, which can be found from here.</p> <p>The <code><properties></code> tag defines what the characteristics access and security properties are, meaning how it can be accessed (read, write etc.) by a remote <i>Bluetooth</i> device what security needs to be in place to access the characteristic. An optional <code>const</code> parameter can also be used to define the value to be constant and non-editable.</p> <p>In case of constant values the actual value of the characteristic can be defined inside the <code><value></code> tags.</p>
<pre><characteristic uuid="2a01"> <properties read="true" const="true" /> <value type="hex">0768</value> </characteristic></pre>	<p>The second characteristic (appearance) is defined in the same manner and has the same properties as the device name. The only difference is that the characteristic is in hex format instead of UTF-8 as defined with <code>type="hex"</code>.</p>

Figure 15: GAP service explained



The full syntax of the hardware configuration file and more examples can be found from the ***Bluetooth Low Energy Profile Toolkit Developer Guide***.

Below if an explanation of the Health Thermometer service used also in the BGDemo application.

```
<!-- Health Thermometer Service -->
<service uuid="1809" advertise="true">

    <description>Health Thermometer Service</description>

    <!-- Temperature Measurement -->
    <characteristic uuid="2a1c" id="xgatt_temperature_celsius">
        <!-- Read property is not included in the HTM service specification -->
        <!-- It's been added so the data can be read from Android devices -->
        <properties indicate="true" read="true" />
        <value type="hex">0000000000</value>
    </characteristic>
</service>
```

Figure 16: Health Thermometer Service

<pre><service uuid="1809" advertise="true"/></pre>	<p>UUID 1809 is assigned for the HTM service and the details of the service can be found here.</p> <p>advertise="true" means that the UUID of the service is automatically included in the BLE advertisement data. The benefit of this is that a device making a device discovery can learn the supported service directly from the advertisement data without making a service discovery.</p>
<pre><characteristic uuid="2a1c" id="xgatt_temperature_celsius"> <properties indicate="true" read="true" /> <value type="hex">0000000000</value> </characteristic></pre>	<p>UUID 2a1c is assigned for the temperature measurement characteristic and its details can be found here.</p> <p>id="xgatt_temperature_celsius" is used to assign a unique ID for the characteristic, which can be later on used in the BGScript application to read or write the characteristic's value.</p> <p>This characteristic has two properties enabled, which are indicate and read. Read property is not included in the official HTM service specification, but due to the some Android applications' limitations it's included in the project, so the value can be accessed from Android as well.</p> <p>The value is initialized with zeros but the 0000000000 also initializes the characteristic's length to 5 bytes.</p>

Figure 17: HTM service explained

5.3.2 Bluetooth SPP SDP entry

For *Bluetooth* BR/EDR the SDP entries also need to be configured. The included SDP entries are defined in the project configuration file as shown in chapter 5.1. In addition, the actual XML file needs to exist and in the examples provided these are named SPP.xml in the project configuration file. An example of an SPP.xml file is shown below.

```
<!-- SDP record for Bluetooth BR/EDR SPP profile -->
<ServiceRecord>

  <ServiceClassIDList>
    <ServiceClass uuid128="1101"/>
  </ServiceClassIDList>

  <ProtocolDescriptorList>
    <Protocol>
      <UUID16 value="0100"/>
    </Protocol>
    <Protocol>
      <UUID16 value="03"/>
      <UINT8 value="05"/>
    </Protocol>
  </ProtocolDescriptorList>

  <BrowseGroupList>
    <UUID16 value="1002"/>
  </BrowseGroupList>

  <LanguageBaseAttributeIDList>
    <UINT16 value="656e"/>
    <UINT16 value="006a"/>
    <UINT16 value="0100"/>
  </LanguageBaseAttributeIDList>

  <!-- Service name -->
  <ServiceName text="Bluetooth Serial Port" language_id="0100"/>

  <BluetoothProfileDescriptorList>
    <Profile>
      <UUID16 value="1101"/>
      <UINT16 value="0100"/>
    </Profile>
  </BluetoothProfileDescriptorList>

</ServiceRecord>
```

Figure 18: SDP record for a single SPP profile

The table below shows the configurable options in the SPP configuration file (SPP.xml).

<pre><ServiceClassIDList> <ServiceClass uuid128="1101"/> </ServiceClassIDList></pre>	<p>This defines the UUID of the <i>Bluetooth</i> profile. For <i>Bluetooth</i> Serial Port Profile the UUID must be 1101 and <u>should not be changed</u>.</p>
<pre><BrowseGroupList> <UUID16 value="1002"/> </BrowseGroupList></pre>	<p>This section defines if this SDP entry is visible in the SDP browse group. Typically, you should not change this, but for some special applications you might want to disable the browse group visibility.</p>
<pre><ProtocolDescriptorList> <Protocol> <UUID16 value="0100"/> </Protocol> <Protocol> <UUID16 value="03"/> <UINT8 value="05"/> </Protocol> </ProtocolDescriptorList></pre>	<p>value="0100" means this profile is based on top of RFCOMM. value="03" means the next parameter defines the assigned RFCOMM channel value="05" defines the RFCOMM channel assigned for the profile</p> <p>Note: You can only change the RFCOMM channel number, <u>but you must take care that when in your application code you load the RFCOMM server you use exactly the value defined in this XML file.</u></p>
<pre><ServiceName> text="Bluetooth Serial Port" language_id="0100" </ServiceName></pre>	<p>This defines the service name for the given UUID. If you want to rename the service you can modify the <i>Bluetooth Serial Port</i> to contain something else.</p>



For typical use, do not modify SPP profiles in the SDP configuration file.

5.3.3 Bluetooth Device Information Profile SDP entry

Similarly for the Device Information Profile there is a corresponding XML file, named DID.xml in the project configuration files of the examples provided. This SDP entry is mandatory and describes certain characteristics of the module such as Vendor ID, Product ID, Version etc.

```
<!-- SDP record for Bluetooth BR/EDR DI profile -->
<ServiceRecord>

<LanguageBaseAttributeIDList>
  <UINT16 value="656e"/>
  <UINT16 value="006a"/>
  <UINT16 value="0100"/>
</LanguageBaseAttributeIDList>

<ServiceClassIDList>
  <ServiceClass uuid16="1200"/>
</ServiceClassIDList>

  <UINT16 value="0200"/> <!-- SpecificationID -->
  <UINT16 value="0103"/> <!-- 1.3 -->

  <UINT16 value="0201"/> <!-- VendorID -->
  <UINT16 value="0047"/> <!-- Bluegiga's ID -->

  <UINT16 value="0202"/> <!-- ProductID -->
  <UINT16 value="1234"/> <!-- dummy -->

  <UINT16 value="0203"/> <!-- Version -->
  <UINT16 value="0000"/> <!-- 0 -->

  <UINT16 value="0204"/> <!-- Primary record -->
  <BOOL value="1"/> <!-- true -->

  <UINT16 value="0205"/> <!-- VendorIDSource-->
  <UINT16 value="0001"/> <!-- Bluetooth SIG -->

</ServiceRecord>
```

Figure 19: Bluetooth Device Information Profile

The table below describes the configuration options used in the example on previous page.



Do not change the order of the configuration parameters.

<pre><UINT16 value="0200"/> <UINT16 value="0103"/></pre>	<p><u>This MUST not be changed.</u></p>
<pre><UINT16 value="0201"/> <UINT16 value="0047"/></pre>	<p>0201 refers to vendor ID parameter and you can change the 0047 to your own vendor ID if you have one assigned from USB Implementers Forum or <i>Bluetooth</i> SIG. If you do not have your own vendor ID you can keep using 0047 <u>unless you are making MFI compliant devices in which can you must have your own ID.</u></p>
<pre><UINT16 value="0202"/> <UINT16 value="1234"/></pre>	<p>0202 refers to product ID parameter and if you have decided to use your own vendor ID you can also use your own product ID as well and change 1234 to something else.</p> <p>If case you are using the default vendor ID, this value must not be changed.</p>
<pre><UINT16 value="0203"/> <UINT16 value="0000"/></pre>	<p>0202 refers to product version and you can replace the value 0000 with your own version number.</p>
<pre><UINT16 value="0204"/> <UINT16 value="1"/></pre>	<p><u>This MUST not be changed.</u></p>
<pre><UINT16 value="0205"/> <UINT16 value="0001"/></pre>	<p>0205 refers to the source of the vendor ID and it must tell if your own vendor ID is from <i>Bluetooth</i> SIG or USB Implementers Forum</p> <p>0000: Source of vendor ID is USB Implementers Forum</p> <p>0001: Source of vendor ID is <i>Bluetooth</i> SIG</p>



For typical use, do not modify the SDP configuration file of the DI profile.

5.4 BGScript code

This section explains the most relevant sections of the BGScript code used in the BGDemo application and explains how the application works.

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 BGDemo the boot event:

1. Writes data (the welcome message) to UART endpoint
2. Configures the GPIO pin, which are used to control the display, directions and set's the initial states
3. Initializes the displays settings. See the displays data sheet for more details.
4. Configures both UART and SPI data to be dropped (discarded) the data by default as the UART data is only used when SPP connection is active. The SPI data is discarded, because the display sends responses to the commands, which are ignored in this application.

```
# Boot event listener - Generated when the module is started
event system_boot(major,minor,patch,build,bootloader,hw)

    rep=-1

    # Write welcome message to UART interface at boot
    call endpoint_send(0,20,"BT121 HTM+SPP Demo\r\n")

    # Set the display to command mode
    # Configure Port 0 pin 14 as output
    call hardware_configure_gpio(0,14,hardware_gpio_output,hardware_gpio_float)
    # Configure Port 1 pin 14 as output
    call hardware_configure_gpio(1,14,hardware_gpio_output,hardware_gpio_float)
    # Write I/O port statuses
    call hardware_write_gpio(0,$4000,$4000)
    call hardware_write_gpio(0,$4000,$0)
    call hardware_write_gpio(1,$4000,$0)

    # Initialize the display
    # Change display cursor position
    call endpoint_send(3,11,"\x30\x30\x30\x39\x14\x56\x6d\x70\x0c\x06\x01")

    # By default stream data to DROP endpoint meaning the data will be ignored
    call endpoint_set_streaming_destination(3,31)
    call endpoint_set_streaming_destination(0,31)

    # Initialize display data
    display(0:16)= " Bluegiga BT121 "
    display(40:16)= " 20.C  UART  "
    display(43:1)=$df # degree
end
```

Figure 20: System boot



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

Once the *Bluetooth* radio has been initialized an event is generated, which indicates the radio is ready to be used.

In this event handler the code:

1. Configures the friendly name for *Bluetooth* BR/EDR (BLE device name is configured in the GATT)
2. Starts RFCOMM server for incoming SPP connections
3. Configures the security manager for Bluetooth Just Works pairing mode
4. Starts RFCOMM server which enables SPP connections and adds SPP into the SDP record
 - a. Parameter 5 refers to the RFCOMM channel and must match the value used in SPP's SDP configuration file
 - b. Parameter 2 refers to the SDP entry used for the RFCOMM and must match the value used in project configuration file
 - c. Parameter 0 defines the endpoint (in this case UART) where the data from the SPP connection is routed by default.
 - d. Finally, the code starts a software timer and updates the text on the display

```
# System initialized event listener - Generated when the module is ready to be used
event system_initialized(addr)

# Set local Bluetooth friendly name for Bluetooth classic
call system_set_local_name(18,"BT121 HTM+SPP Demo")

# Configure Bluetooth Security Manager
# No MITM required, no input/output -> Just Works pairing mode
call sm_configure(0,3)
# Enable bonding mode
call sm_set_bondable_mode(1)

# Start RFCOMM (SPP) server at RFCOMM channel 5, with SDP ID 2
call bt_rfcomm_start_server(5,2,0)

# Set Bluetooth classic mode to visible and connectable
call bt_gap_set_mode(1,1,0)

# Start Bluetooth LE advertisements and enable connections
call le_gap_set_mode(2,2)

# Set software timer to tick each second
call hardware_set_soft_timer(1000,0,0)

# Set the display to data mode
call hardware_write_gpio(1,$4000,$4000)

# Write data to display
call endpoint_send(3,80,display(0:80))
end
```

Figure 21: System initialized event



Verify the **bt_rfcomm_start_server** command uses the same parameters used in the project and SDP configuration files.

In order to detect the SPP connection establishment and disconnections two endpoint event handlers are needed. The first event handler show below is used to detect an established SPP connection and it:

1. Sends a status message to UART when connection is established
2. Sends a status message to *Bluetooth* when connection is established
3. Configures the UART into transparent mode, so data sent to UART is transparently forwarded to the SPP connection.

In order to catch the disconnection event, the second event listener is needed, which:

1. Confirms the endpoint closing event
2. Sends a status message to UART when connection is closed

```
# Event listener for endpoint status changes
# These events are generated for example when SPP connection is opened
event endpoint_status(endpoint,type,destination,flags)

    # Endpoint status has changed to RFCOMM and become active
    # This means SPP connection has been received
    if (type = endpoint_rfcomm)  && (flags&ENDPOINT_FLAG_ACTIVE)

        # Store endpoint ID
        rep=endpoint

        display(40+13:2)="BT"

        # Write a message to UART interface to indicate SPP connection
        call endpoint_send(0,15,"SPP CONNECTED\r\n")

        # Send a welcome message to Bluetooth SPP connection
        call endpoint_send(endpoint,29,"Connected to BT121 SPP DEMO\r\n")

        # Configure data streaming from UART to SPP connection - enable transparent SPP mode
        call endpoint_set_streaming_destination(0,endpoint)
    end if
end

# This event is generated when an endpoint is closing
# for example when SPP connection is closed
event endpoint_closing(reason,endpoint)

    # Close endpoint also in the module
    call endpoint_close(endpoint)

    # If RFCOMM endpoint is closing, which means SPP connection is closed
    # Write a message to UART
    if(endpoint = rep)
        display(40+13:2)="  "
        call endpoint_send(0,18,"SPP DISCONNECTED\r\n")
    end if
end
```

Figure 22: SPP connection and disconnection event handlers

The software timer launched in the initialization event is used to read the temperature from the I2C altimeter/temperature sensor. The code does the following items:

1. Reads the temperature from the sensor
2. Converts the reading to the format required by the HTM specification
3. Writes the data to local GATT database so it can be read by a remote device
call gatt_server_write_attribute_value(...)
4. Sends an indication of the value change to all remote devices listening for indications
call gatt_server_send_characteristic_notification(...)
5. Converts the temperature to ASCII
6. Check to which direction SPP data is being transmitted
7. Updates the information to the display

```
# Software timer listened - generated when software timer runs out
event hardware_soft_timer(handle)

    # Start a single measurement from I2C thermometer
    call hardware_write_i2c(1,$C0,2,$0226)
    call hardware_stop_i2c(1)
    # Wait for the measurement result
    f=0
    while f=0
        call hardware_write_i2c(1,$C0,1,$26)
        call hardware_read_i2c(1,$C0,1)(r,l,b)
        call hardware_stop_i2c(1)
        if (b&2)=0
            f=1
        end if
    end while
    # Read temperature from I2C
    call hardware_write_i2c(1,$C0,1,$04)
    call hardware_read_i2c(1,$C0,1)(r,l,b)
    call hardware_stop_i2c(1)

    # Build HTM service's temperature reading characteristic
    ...

    # Write attribute to local GATT data base
    call gatt_server_write_attribute_value(xgatt_temperature_celsius,0,5,tmp(0:5))(r)
    # Send indication to all "listening" clients
    # 0xFF as connection ID will send indications to all connections
    call gatt_server_send_characteristic_notification($ff, xgatt_temperature_celsius,5,tmp(0:5))(r)

    # Temperature to ASCII
    ...

    # Update TX&RX to display to which direction the SPP data is sent
    ...

    # Update display
    call endpoint_send(3,80,display(0:80))
end
```

Figure 23: Software timer event handler



The figure above shows only partial code. See the BGScript file for more information.

One more event listener is used in the BGDemo application to detect closed/lost BLE connections and to enable advertisements again.

```
# BLE disconnect event listener
event le_connection_closed(reason,connection)

    # BLE connection closed, restart advertisement
    call le_gap_set_mode(2,2)
end
```

Figure 24: BLE disconnection event listener

5.5 Compiling BGDemo application

The demo application can be compiled with the BGBuild compiler.

The syntax is:

Usage: *bgbuild.exe [options] input*

BGBuild tool for Bluetooth Dual Mode

Options:

-?, -h, --help	Displays this help.
-v, --version	Displays version information.
-g, --gattonly	Only create GATT c-file.
-r, --root <buildtools>	Override default build tools location
-s, --scompiler <scriptcompiler>	Path to script compiler

Arguments:

input	Project file to build.
-------	------------------------

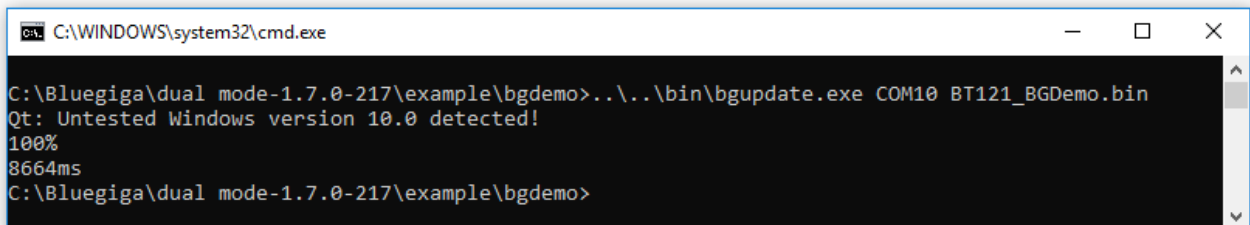
```
C:\WINDOWS\system32\cmd.exe
C:\Bluegiga\dual mode-1.7.0-217\example\bgdemo>..\bin\bgbuild.exe project.xml
Qt: Untested Windows version 10.0 detected!
-mcpu=cortex-m0 -mthumb -Os -Wl,--gc-sections -IC:/Bluegiga/dual mode-1.7.0-217/bin
../fw/build -nostartfiles -TC:/Bluegiga/dual mode-1.7.0-217/bin/..fw/build/linker
.ld -LC:/Bluegiga/dual mode-1.7.0-217/bin/..fw -lbt121_tiny -lubl -lcommon gatt_db
.c -o C:/Users/jemareci/AppData/Local/Temp/bgbuild.H31116
C:/Bluegiga/dual mode-1.7.0-217/bin/..fw/libbt121_tiny.a
in :C:/Users/jemareci/AppData/Local/Temp/bgbuild.p31116
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
I2C:1
prescaler :2
divider :40
alternate :1
```

Figure 25: Compiling the BGDemo

5.6 Installing the firmware

The firmware binary can be installed to a DKBT development kit or a BT121 module with the BGUpdate tool. The syntax is:

usage: *bgupdate.exe COM-port .BIN-file*

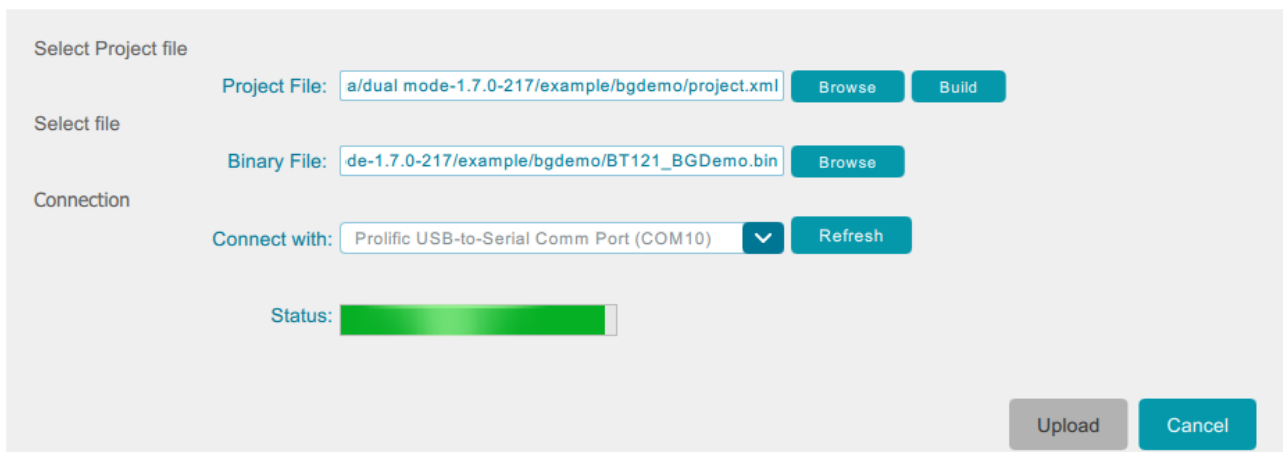


```
C:\WINDOWS\system32\cmd.exe

C:\Bluegiga\dual mode-1.7.0-217\example\bgdemo>..\bin\bgupdate.exe COM10 BT121_BGDemo.bin
Qt: Untested Windows version 10.0 detected!
100%
8664ms
C:\Bluegiga\dual mode-1.7.0-217\example\bgdemo>
```

Figure 26: Firmware update with BGUpdate

Alternatively, you can use the **Upload Tool** tab in the BGTool to upload the firmware to the development kit or Bluetooth module.



Select Project file

Project File:

Select file

Binary File:

Connection

Connect with:

Status:

Figure 27: Firmware update with BGUpdate

5.7 Testing BGDemo application

You can find the test instructions from the **DKBT Development Kit Quick Start**.

Smart. Connected. Energy-Friendly.



IoT Portfolio
www.silabs.com/products



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.

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

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