

BGAPI AND BGLIB SPP SERVER EXAMPLE
APPLICATION NOTE FOR BT121 AND DKBT

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

1	Introduction	4
1.1	About the application note.....	4
2	Factory default configuration	5
2.1	BT121 <i>Bluetooth</i> Dual Mode module factory configuration	5
2.2	Development Kit factory configuration	5
3	Getting started with the BT121 <i>Bluetooth</i> Dual Mode Software	6
3.1	Installing the <i>Bluetooth</i> Dual Mode SDK.....	6
3.2	Folder structure	7
3.3	Included tools	7
4	Walkthrough of the BT121 SPP Server demo project	8
4.1	BGAPI project components.....	8
4.2	Module firmware project.....	10
4.3	Host application project and BGLib considerations	16
4.4	Compiling the module firmware project with BGBuild.....	25
4.5	Flashing the new firmware into the <i>Bluetooth</i> Dual Mode hardware	26
4.6	Compiling the host application project with Visual Studio.....	27
4.7	Running the host application on Windows	29
4.8	Testing client connectivity with an Android smartphone	30
5	Summary and BGAPI quick-reference guide	33

1 Introduction

1.1 About the application note

This document discusses the structure and usage of the BGAPI serial protocol and of its companion BGLib library by walking through one of the sample implementations that come with the BT121 Software Development Kit (SDK).

The example project components are designed to be used with the BT121 Module Development Kit and the free “Community” version of Microsoft Visual Studio, but the core implementation works with any ANSI C host platform. Further, since BGAPI itself is a simple binary protocol, it is possible to create or port similar libraries onto any platform that is capable of host-to-module communication over a UART interface.

To follow along with the material in this application note as closely as possible, you should have the following:

- BT121 Module Development Kit
- Microsoft Visual Studio Community Edition or similar
- An Android-based phone with *Bluetooth* 4.0 (or higher) support, running apps such as the “S2 *Bluetooth* Terminal” for *Bluetooth* BR/EDR’s SPP communication or the “BLE Tool” for scanning for BLE advertisements and for connecting to BLE devices

For compiling the host application project, alternative toolchains may be used with small code modifications or Makefile creation. For client-side testing, other smartphone apps or completely different BLE-capable devices may be used with similar results if you follow the same basic steps.



This application note assumes as a prerequisite some familiarity with concepts like BGAPI, BGLib and the module's software architecture. This application note also assumes you have read and understood the *Bluetooth* Dual Mode **Software Getting Started Guide**.

2 Factory default configuration

Below are short descriptions of what the default configurations in the BT121 modules and development kits are.

2.1 BT121 *Bluetooth* Dual Mode module factory configuration

When the bare modules are delivered from the factory, they have the *Bluetooth* Dual Mode software with the following configuration pre-installed:

- Software: Newest stable release of the *Bluetooth* Dual Mode stack – Firmware obtained from the example project files under the directory `\example\bt121\` of the SDK
- Behavior: Boot into idle mode and wait for BGAPI commands from host (module will not be visible in BLE scan nor in BR/EDR inquiries by default)
- Host Interface:
 - BGAPI serial protocol over UART interface
 - UART baud rate: 115200
 - Hardware flow control: enabled
 - Data bits: 8
 - Parity: none
 - Stop bits: 1
- Firmware update interfaces:
 - BGAPI-based DFU over UART enabled
 - ST-based DFU over UART enabled
 - ARM 2-pin Serial Wire Debug (SWD) available

2.2 Development Kit factory configuration

When the BT121 Development Kits are delivered from the factory their modules have the following configuration pre-installed:

- Software: Newest stable release of the *Bluetooth* Dual Mode stack – Firmware obtained from the example project files under the directory `\example\bgdemo\` of the SDK
- Behavior: Boots into advertising mode automatically, for the module to be visible in BLE scan and connectable, and also becomes automatically visible and connectable, while starting the RFCOMM server, in order to allow incoming *Bluetooth* BR/EDR's Serial Port Profile (SPP) connections
- Host interface:
 - None
- Firmware update interfaces:
 - ST- based DFU over UART enabled
 - ARM 2-pin Serial Wire Debug (SWD) available

3 Getting started with the BT121 *Bluetooth* Dual Mode Software

3.1 Installing the *Bluetooth* Dual Mode SDK

To install the latest BT121 *Bluetooth* Dual Mode SDK, please perform the following steps:

1. Go to <https://www.bluegiga.com/en-US/products/bt121-bluetooth-dual-mode/#documentation>
2. Download the *Bluetooth* Dual Mode SDK which is found under the “Software Releases” section
3. Run the installer executable
4. Follow the on-screen instructions to install the SDK

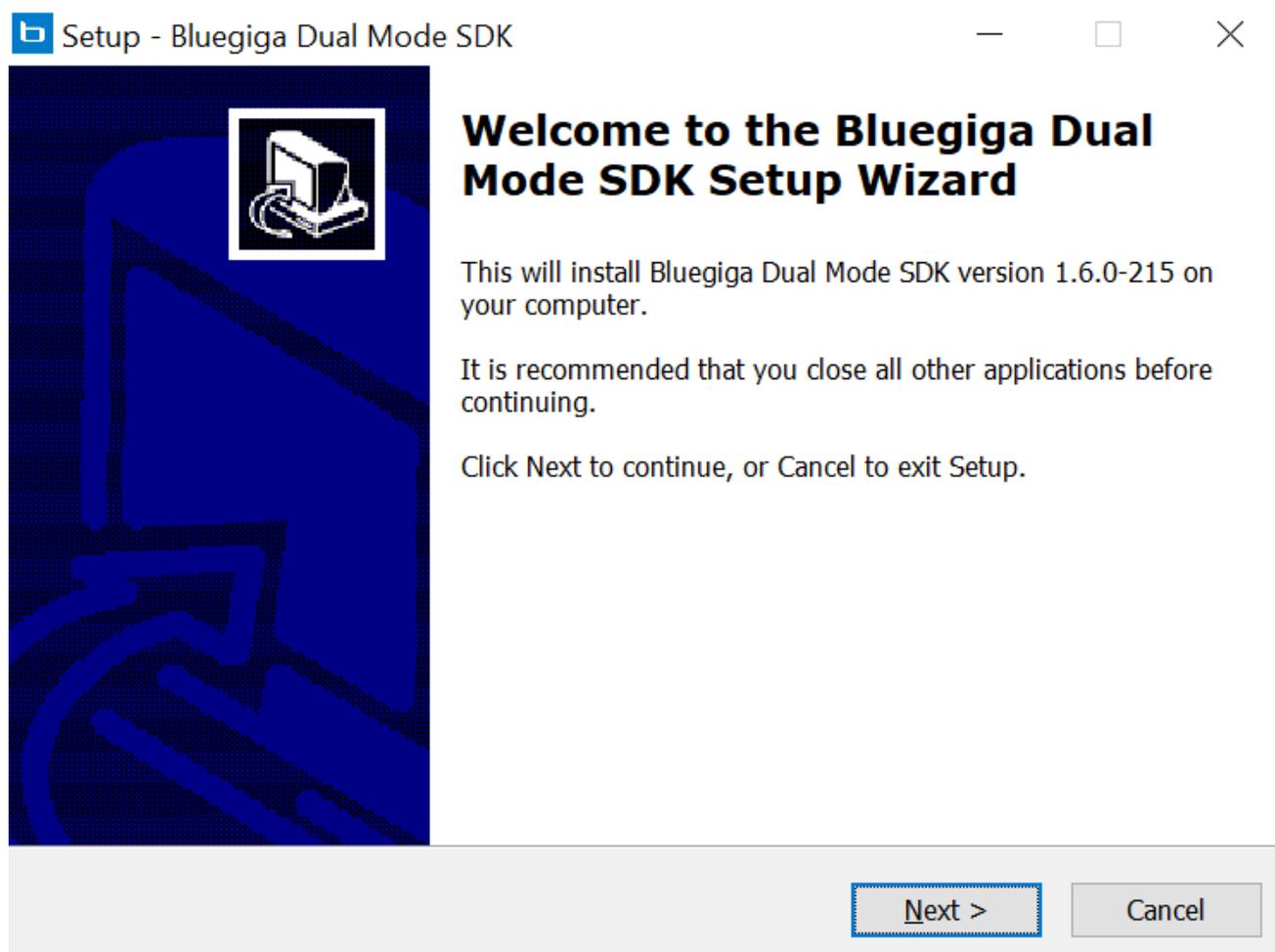


Figure 1: Installing the SDK

3.2 Folder structure

The SDK creates the following folders into the installation directory.

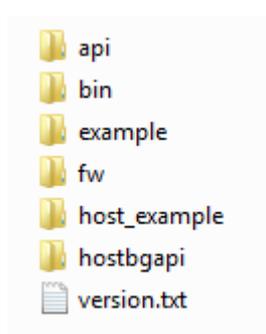


Figure 2: SDK folders

- **API** Includes a raw XML description of the *Bluetooth* stack's API. Could be used to automatically generate other parsers for the API.
- **BIN** This folder includes all the binary applications needed by the SDK.
- **EXAMPLES** This folder includes the BGScript demo applications, profile toolkit examples and the *Bluetooth* module configuration examples, meant to generate example firmwares for the *Bluetooth* module when run through the BGBuild compiler.
- **FW** This folder contains the actual *Bluetooth* Dual Mode stack firmware binaries.
- **HOST_EXAMPLE** This folder contains C source code examples for the host microcontrollers, demonstrating BGLib usage: the project discussed in detail in this application note is found under this directory.
- **HOSTBGAPI** Contains the BGLib library files and an HTML version of the BGAPI, BGScript, and BGLib API reference documentation.

3.3 Included tools

The following tools are installed by the SDK:

- **BGTool** A graphical UI tool for Windows 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 BT121 *Bluetooth* Dual Mode Software, without writing any software first. Within the graphical tool it is also possible to compile and flash firmware onto the modules.
- **bgbuild.exe** A free compiler that is used to build firmware images for the *Bluetooth* Dual Mode modules.
- **bgupdate.exe** A re-flashing tool based on the ST protocol for installing firmware into the BT121 modules over the USB-UART interface of the DKBT Development Kit. This tool, in combination with the Prolific-based USB-UART converter works regardless of what firmware was previously on the module, and is especially helpful in cases where previous firmware configuration has made the BGAPI-based UART DFU interface inaccessible.

4 Walkthrough of the BT121 SPP Server demo project

This section describes in detail all of the parts which make up a BGAPI/BGLib project in general, and specifically the SPP Server demo project that is the focus of this application note. The purpose of the section is to help you understand how the different parts work together, how to navigate and modify the code structure, and ultimately how to build your own BGAPI/BGLib applications with the *Bluetooth* Dual Mode SDK.

The “**spp_server**” example project, found in the SDK under the `\host_example\` directory, and discussed here, implements a *Bluetooth* Dual Mode peripheral device which allows remote BLE Centrals to receive generic advertisement packets and connect to the BT121, and at the same time it makes the module visible to BR/EDR inquiries, and connectable to automatically accept *Bluetooth* BR/EDR’s SPP connections for the exchange of serial data. The project has the following main features:

- Start the RFCOMM server and load the related SPP SDP record
- Set discoverable and connectable modes for *Bluetooth* BR/EDR
- Activate BLE connectable advertising
- Accept all RFCOMM connections by remote devices while taking care of pairing and bonding
- While a RFCOMM connection is ongoing it will print all the bytes received from the remote party and it will automatically send back to sender ASCII characters carrying the information about how many bytes were received
- Auto-resume advertising upon BLE disconnection

If some of these concepts are unfamiliar to you, don’t worry. They are discussed in much more detail in sections below. The behavior of this application is arbitrary in many ways, but it is designed to provide a reference for some of the functions that are common to *Bluetooth* Dual Mode applications.

4.1 BGAPI project components

Unlike BGScript-based projects which are built into single, standalone firmware images, projects which use BGAPI have two separate parts:

1. Module firmware project
 - Hardware configuration (enabling UART with BGAPI)
 - GATT structure definition
 - Definition of *Bluetooth* BR/EDR’s SDP entries
 - Compiled and flashed with *Bluetooth* Dual Mode SDK tools
2. Host application project
 - Application logic
 - BGAPI parser/generator (BGLib code)
 - Host-specific UART RX/TX functions
 - Compiled with host platform toolchain (Visual Studio, gcc, etc.)
 - Flashed if necessary with host MCU flash tools

Either one of these halves alone cannot function completely without the other in place. It is therefore important to understand what each one does and where it must be implemented. The two components for the demo project in this application note come with the BT121 *Bluetooth* Dual Mode SDK, and can be found in these two subfolders where the SDK is installed:

- `\example\bt121\` – Module firmware project
- `\host_example\spp_server\` – Host application project with Visual Studio solution file

The figure below illustrates the demo application architecture.

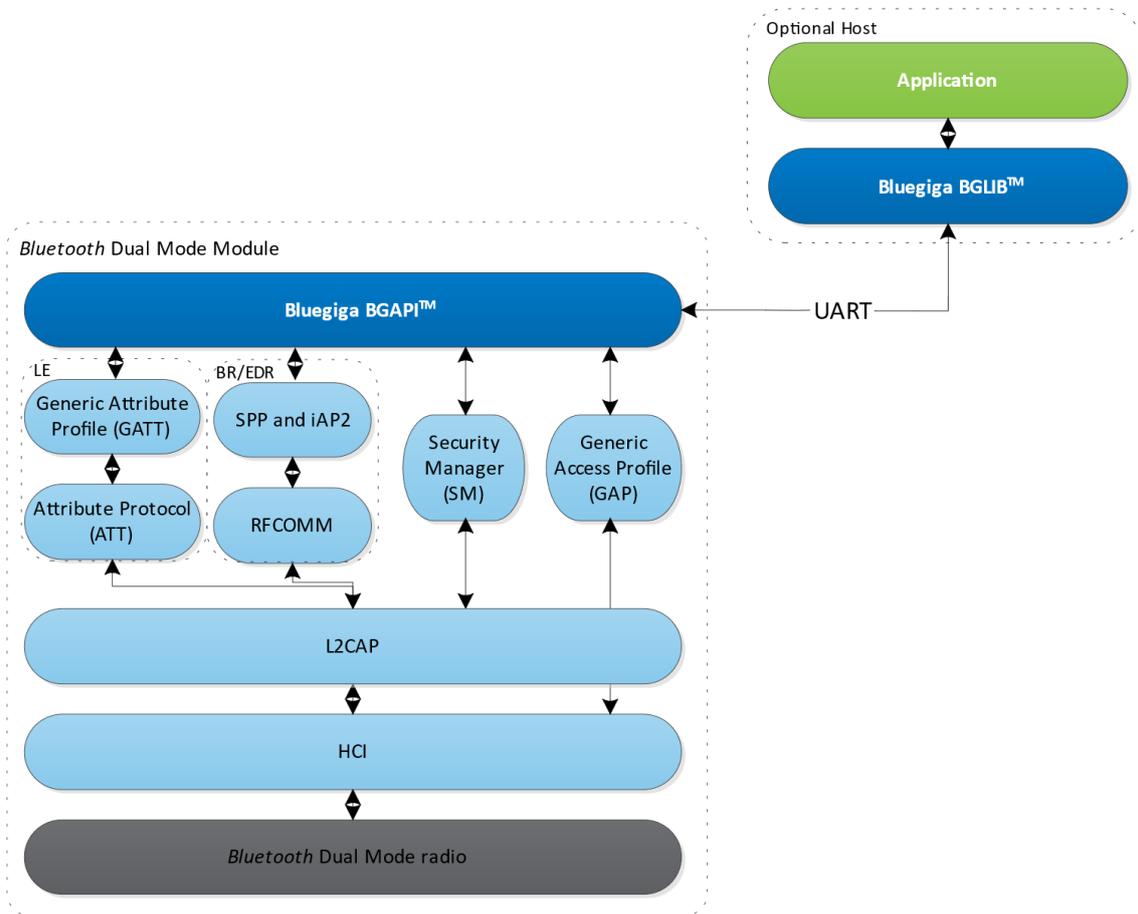


Figure 3: BGAPI/BGLib project application architecture

4.2 Module firmware project

Building a *Bluetooth* Dual Mode project starts always by making a project file, which is an XML file that defines the resources used in the project. In the case of this project, such definition is entered into the “**project.xml**” file:

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2
3 <!-- Project configuration including BT121 device type -->
4 <project device="bt121">
5
6   <!-- XML file containing GATT service and characteristic defir
7   <gatt in="gatt.xml" />
8
9   <!-- Local hardware interfaces configuration file -->
10  <hardware in="hardware.xml" />
11
12  <!-- Local SDP entries for Bluetooth BR/EDR -->
13  <sdp>
14    <entry file="did.xml" autoload="true"/>
15    <entry file="spp.xml" id="2"/>
16  </sdp>
17
18  <!-- Firmware output files -->
19  <image out="BT121_BGAPI_UART115k.bin" />
20
21 </project>
22

```

Figure 4: Project file

It is common for project definitions to be split up into multiple files, such as one for the GATT structure, one for the hardware configuration, and so on. However, note that this is not required. Below is a table explaining the syntax of the project.xml file:

<project device="bt121">	This tag starts the main project definition. The device attribute is used to define which <i>Bluetooth</i> module the project will be compiled for.
<gatt ... />	The <gatt> tag is used to define the file containing the local GATT structure on this device.
<hardware in="hardware.xml" />	The <hardware> tag is used to point to the file which defines the hardware configuration for interfaces like UART, SPI or I ² C, and GPIO.
<sdp> ... </sdp>	The <sdp> section is used to point to the SDP definition file(s) for supported <i>Bluetooth</i> BR/EDR profiles. It also configures the automatic loading of the desired SDP records at boot. Unique IDs for each SDP entry can be configured for later use, for example when SDP records have to be dynamically loaded at runtime.
<image out="BT121_BGAPI_UART115k.bin" />	The <image> tag defines the name of the BGBuild compiler output file. The generated firmware image file with extension .bin contains the <i>Bluetooth</i> Dual Mode stack, the GATT database, the hardware configuration and the BGScript code (optional).

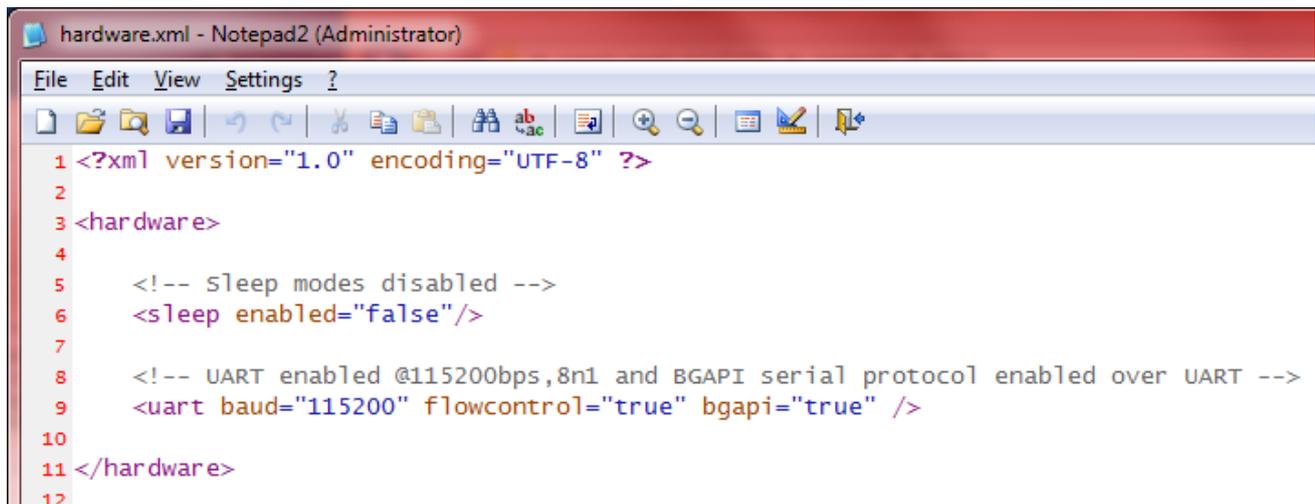
Table 1: project.xml explained



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

4.2.1 Hardware configuration

The hardware configuration in the module-side component of a BGAPI/BGLib project is critical, since it must define the UART peripheral interface used for BGAPI control from the external host. Without this, it is not possible to use BGAPI externally at all. The hardware definition from the hardware.xml is reproduced below:



```

1 <?xml version="1.0" encoding="UTF-8" ?>
2
3 <hardware>
4
5     <!-- sleep modes disabled -->
6     <sleep enabled="false"/>
7
8     <!-- UART enabled @115200bps,8n1 and BGAPI serial protocol enabled over UART -->
9     <uart baud="115200" flowcontrol="true" bgapi="true" />
10
11 </hardware>
12

```

Figure 5: Hardware configuration

Explanation of the syntax is in the table below:

<pre><uart baud="115200" flowcontrol="true" bgapi="true" /></pre>	<p>The <uart> tag is used to enable UART peripherals and to configure the UART port settings such as baud rate and flow control. This tag also controls whether the given UART port is to expose the BGAPI layer as a host control point.</p> <p>This project configures the UART for 115200 baud, 8/N/1, flow control enabled, and BGAPI enabled. These parameters must be kept in mind for the host application design.</p>
<pre><sleep enabled="false"/></pre>	<p>The <sleep> tag is used to either prevent sleep modes to be entered ("false", like in this demo project), or to enable all power modes ("true"), where the selection of the most appropriate mode is done automatically by the firmware at any time.</p>

Table 2: hardware.xml explained



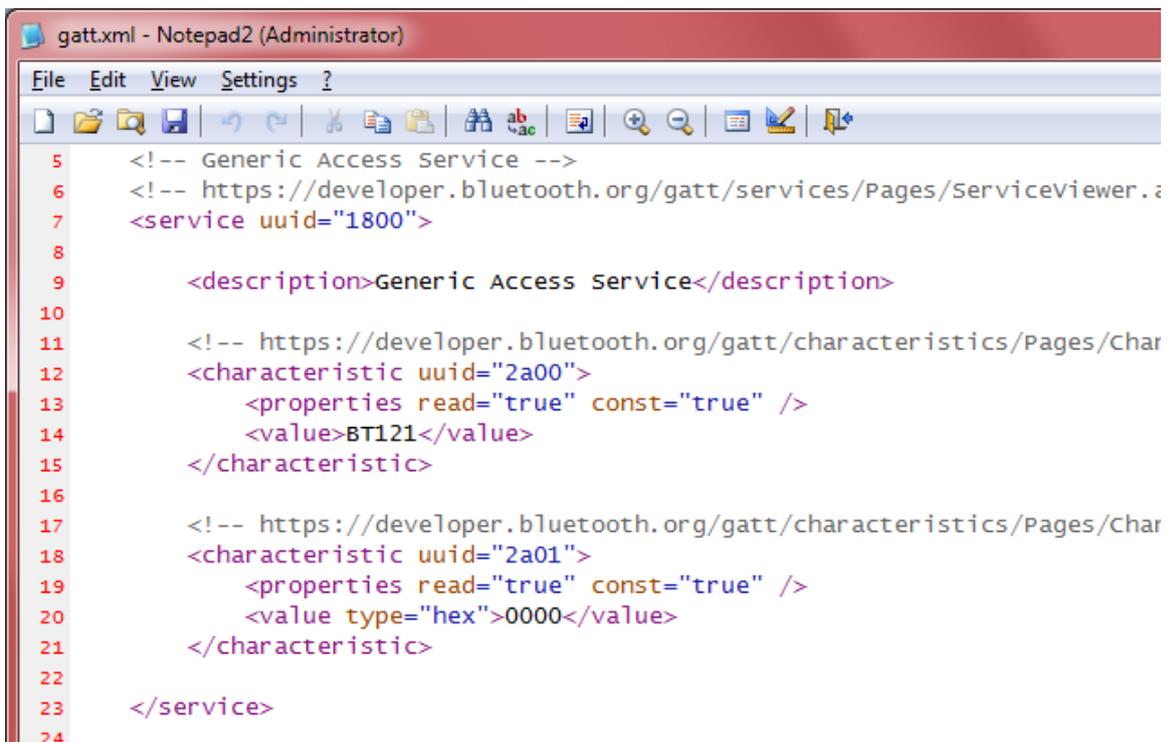
The full syntax of the project configuration file and more examples can be found in the **Bluetooth Dual Mode Configuration Guide**

4.2.2 GATT services configuration

The *Bluetooth* GATT services made available on the device are defined using the XML-based language shown below in a file which is then compiled with the BGBuild compiler as part of the firmware. The GATT database in this application simply includes two officially adopted services (Generic Access and Device Information). Profile behavior and specifications are shortly introduced later in this chapter.

More detailed information about official adopted profiles can be found from *Bluetooth* SIG web page: <https://developer.bluetooth.org/gatt/profiles/Pages/ProfilesHome.aspx>

First, take a look at the **Generic Access** service and characteristics:



```
gatt.xml - Notepad2 (Administrator)
File Edit View Settings ?
5 <!-- Generic Access Service -->
6 <!-- https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer. :
7 <service uuid="1800">
8
9     <description>Generic Access Service</description>
10
11     <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/Char
12 <characteristic uuid="2a00">
13     <properties read="true" const="true" />
14     <value>BT121</value>
15 </characteristic>
16
17     <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/Char
18 <characteristic uuid="2a01">
19     <properties read="true" const="true" />
20     <value type="hex">0000</value>
21 </characteristic>
22
23 </service>
24
```

Figure 6: Generic Access Protocol (GAP) service



The full syntax of the GATT configuration file and more examples can be found from the **BLUETOOTH DUAL MODE PROFILE TOOLKIT DEVELOPER GUIDE**

Every *Bluetooth* Dual Mode peripheral implementing a GATT server contains this service, which includes the device name and appearance.

The figure 6 shows part of the GATT database used in the demo application and how it looks in the Profile Toolkit XML format. An explanation of the syntax can be found in the table on the next page.

<pre><service uuid="1800" /></pre>	<p>The <service> tag is used to define a start of a GATT service. The <i>uuid</i> attribute defines the 16-bit or 128-bit UUID used by the service. In this case, the 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><characteristic uuid="2a00"> <properties read="true" const="true" /> <value>BT121</value> </characteristic></pre>	<p>The <characteristics> tag starts the definition of a characteristic, by which the actual data is 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 <properties> 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 and what security needs to be in place to access the characteristic. An optional const 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 <value> tags. For non-const values, all data must be initialized at runtime.</p>
<pre><characteristic uuid="2a01"> <properties read="true" const="true" /> <value type="hex">0000</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 type="hex". If omitted, the default value type is UTF-8.</p>

Table 3: GAP services and characteristics explained

Next, take a look at the **Device Information** service and the two characteristics that it contains:

```

25  <!-- Device Information Service -->
26  <!-- https://developer.bluetooth.org/gatt/services/Pages/Serviceviewer.a
27  <service uuid="180A">
28
29      <description>Device Information Service</description>
30
31      <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/Char
32      <characteristic uuid="2A29">
33          <properties read="true" const="true" />
34          <value>Silicon Labs</value>
35      </characteristic>
36
37      <!-- https://developer.bluetooth.org/gatt/characteristics/Pages/Char
38      <characteristic uuid="2A24">
39          <properties read="true" const="true" />
40          <value>BT121</value>
41      </characteristic>
42
43  </service>
44

```

Figure 7: Device Information service

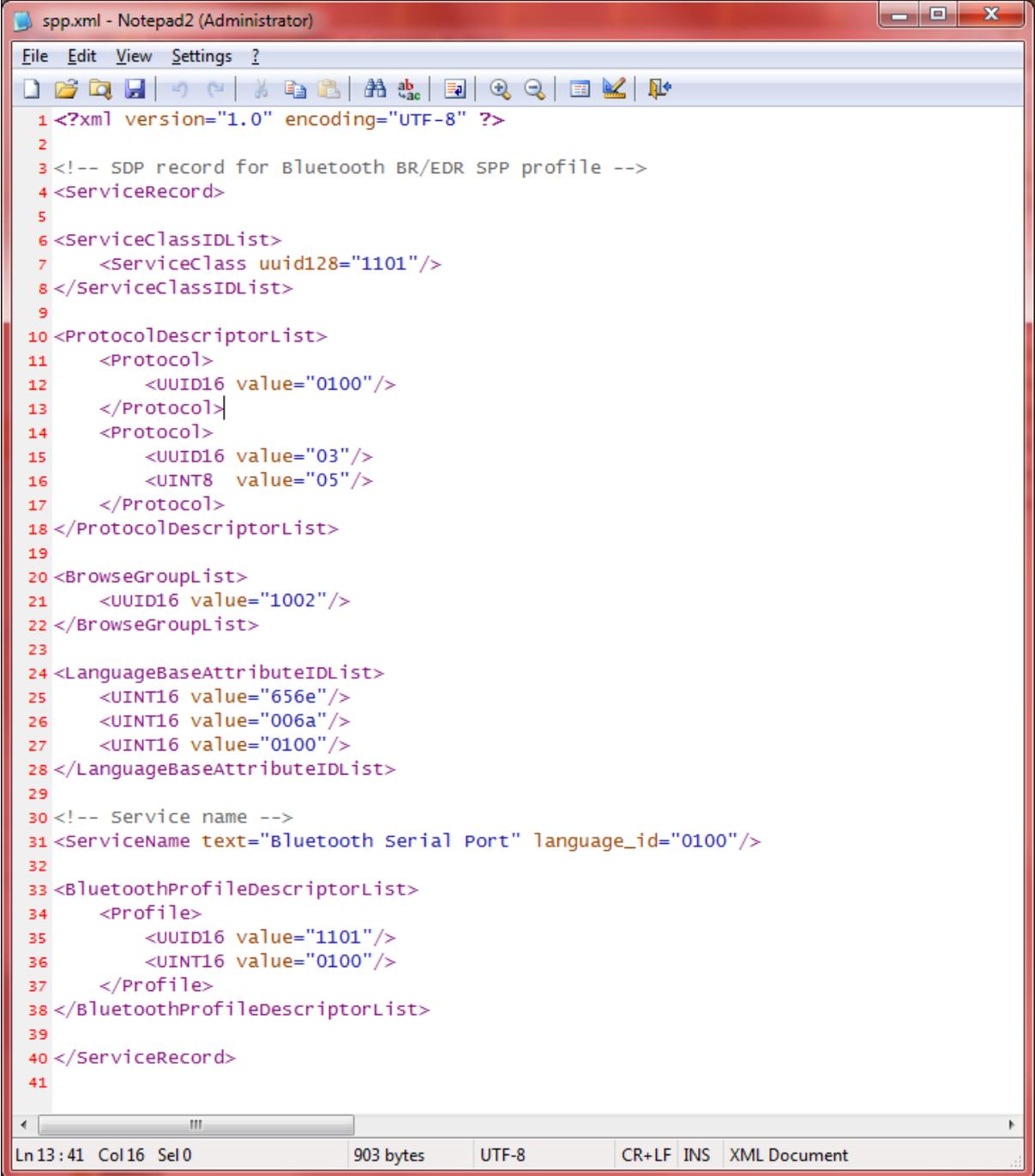
This service is optional for GATT servers, and if included it may contain many optional characteristics, such as manufacturer name and model number string (as shown in figure 7), firmware version, serial number, and others. For more information on what is available here, visit the [Device information service](#) page on the *Bluetooth* SIG website.

4.2.3 SDP entries

The project definition file might also include a section dedicated to *Bluetooth* BR/EDR's SDP (Service Discovery Protocol) entries.

In this demo project, such section is used to point to two XML-based definition files for the SPP (Serial Port Profile) and the DID (Device ID) SDP records, which are required for the correct functionality of the example firmware. Additionally, by the use of the tag *autoload="true"* the firmware is instructed to automatically make the DID record available at boot to any remote device scanning for local services.

The next figure shows the *spp.xml* file which defines how the SDP record for the SPP should be constructed.



```
1 <?xml version="1.0" encoding="UTF-8" ?>
2
3 <!-- SDP record for Bluetooth BR/EDR SPP profile -->
4 <ServiceRecord>
5
6 <ServiceClassIDList>
7   <ServiceClass uuid128="1101"/>
8 </ServiceClassIDList>
9
10 <ProtocolDescriptorList>
11   <Protocol>
12     <UUID16 value="0100"/>
13   </Protocol>
14   <Protocol>
15     <UUID16 value="03"/>
16     <UINT8 value="05"/>
17   </Protocol>
18 </ProtocolDescriptorList>
19
20 <BrowseGroupList>
21   <UUID16 value="1002"/>
22 </BrowseGroupList>
23
24 <LanguageBaseAttributeIDList>
25   <UINT16 value="656e"/>
26   <UINT16 value="006a"/>
27   <UINT16 value="0100"/>
28 </LanguageBaseAttributeIDList>
29
30 <!-- Service name -->
31 <ServiceName text="Bluetooth serial port" language_id="0100"/>
32
33 <BluetoothProfileDescriptorList>
34   <Profile>
35     <UUID16 value="1101"/>
36     <UINT16 value="0100"/>
37   </Profile>
38 </BluetoothProfileDescriptorList>
39
40 </ServiceRecord>
41
```

Figure 8: *spp.xml*

The table below shows the configurable options in the SPP configuration file. Notice that this information, and the corresponding information related to the DID record, is also found in the *BLUETOOTH* Dual Mode SOFTWARE GETTING STARTED GUIDE

<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 should not be changed.</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.</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 "Bluetooth Serial Port" string to contain something else.</p>

Table 4: spp.xml explained



For typical use, do not modify SPP profile's SDP configuration file.

4.2.4 Output firmware image configuration

The project definition file ends with one final tag, which is used to specify the name of the binary output file which will result from the compilation process.

```
<image out="BT121_BGAPI_UART115k.bin" />
```

The file that is produced here can be flashed into a BT121 module using the bgupdate.exe executable from the \bin\ directory of the SDK or via the "Upload tool" section of the GUI-based BGTool application. With SDKs earlier than the version 1.1.0 these methods require the Prolific-based USB-UART converter which is found in the main board of the evaluation kit. Alternatively, the firmware image file can be uploaded to the module using the dedicated BGAPI-based DFU commands over the host UART interface, where the usage of the BGAPI serial protocol must be enabled in the current module's firmware.

This wraps up the discussion of the module firmware project definition.

4.3 Host application project and BGLib considerations

The second major component to any BGAPI/BGLib project is the portion which runs on the host platform outside of the module, and which communicates with the module using the BGAPI protocol over the UART interface.

4.3.1 Project source file overview

The host application project that come with the SDK is written in C and is specifically prepared for the Microsoft Visual Studio Community Edition environment. The code can be ported or in some cases directly compiled with other toolchains, but this document focuses on the Visual Studio project.

The host-side source files for this project are found in the `host_example\spp_server\` subfolder where you have installed the BT121 *Bluetooth* Dual Mode SDK, and comprise only a few source and header files along with the Visual Studio project definition:

Name	Date modified	Type	Size
Debug	04-Sep-15 09:34	File folder	
main.c	03-Aug-15 12:24	C File	9 KB
spp_server.sdf	04-Sep-15 09:40	SQL Server Comp...	31 744 KB
spp_server.sln	03-Aug-15 12:24	Microsoft Visual S...	1 KB
spp_server.vcxproj	04-Sep-15 09:33	VC++ Project	5 KB
uart.h	03-Aug-15 12:24	C/C++ Header	2 KB
uart_win.c	03-Aug-15 12:24	C File	4 KB

Figure 9: List of Visual Studio project files

Here is a quick summary of the relevant files:

main.c	Main application logic to control the BT121 module, and BGLib initialization and setup code.
uart.h	Function declarations for simple UART port initialization and RX/TX data transfers common to most platforms and compatible with BGLib methods and callbacks.
uart_win.c	Implementations for the functions defined in uart.h which are appropriate for a Windows environment.
spp_server.sln spp_server.sdf spp_server.vcxproj	Visual Studio project and solution definition files. If you have Visual Studio 2013 or newer installed, you should be able to open the solution (.sln) file directly and start working with the project.

Table 5: list and explanation of the host project files

Only the **main.c** file contains application-specific code in this case. The rest of the code is managed by the IDE or generic for any BGLib-based project on a Windows platform.

4.3.2 BGLib support file overview

While the files in the previous section contain all of the application logic, the actual BGLib implementation code which contains the BGAPI parser and packet generation functions is found under the `\hostbgapi\` location from the SDK:

Name	Date modified	Type	Size
 apitypes.h	02-Sep-15 11:40	C/C++ Header	1 KB
 bg_errorcodes.h	02-Sep-15 11:40	C/C++ Header	23 KB
 dumo.html	02-Sep-15 11:40	Firefox HTML Doc...	363 KB
 dumo_bglib.h	02-Sep-15 11:40	C/C++ Header	181 KB

Figure 10: List of supporting BGLib include files

The SDK's specific arrangement of files is one possible way that the BGAPI protocol can be used, but it is also possible to create your own library code which implements the protocol correctly with a different code architecture. The only requirement here is that the chosen implementation must be able to create BGAPI command packets correctly and send them to the module over UART, and similarly be able to receive BGAPI response and event packets over UART and process them into whatever function calls are needed to trigger the desired application behavior.

The header files contain primarily *#define*'d compiler macros and named constants which correspond to all of the various API methods and enumerations that you may need to use. The `dumo_bglib.h` file also contains function declarations for the basic packet reception, processing, and transmission functions. All functions defined here use only ANSI C code, to help ensure maximum cross-compatibility on different platforms.

Structure packing and byte order note



The SDK's provided BGLib implementation makes heavy use of direct mapping of packet payload structures onto contiguous blocks of memory, to avoid additional parsing and RAM usage. This is accomplished with the **PACKSTRUCT** macro used extensively in the BGLib header files. It is important to ensure that any ported version of BGLib also correctly packs structures together (no padding on multi-byte struct member variables) in order to achieve the correct operation.

The BGAPI protocol uses little-endian byte ordering for all multi-byte integer values, which means directly mapped structures will only work if the host platform also uses little-endian byte ordering. This covers most common platforms today, but some big-endian platforms exist and are actively used today (Motorola 6800 and 68k, AVR32, and [others](#)). Platforms which require native big-endian byte ordering cannot use the SDK's included BGLib implementation.

4.3.3 BGLib memory requirements

The amount of memory required at the host by the BGLib highly depends on how the BGLib itself is being used. In fact, the BGLib consists of macros that will consume memory only when invoked. The amount consumed by a macro depends on:

- Processor architecture
- Compiler optimization settings and the compiler being used
- Number of parameters

Following are some example figures for a Cortex M0 code when compiled with 'arm-none-eabi-gcc -c -mcpu=cortex-m0 -mthumb -Os main.c' at the command line, where size was verified via 'nm --print-size --size-sort --radix=d main.o'

- One zero-parameters macro → 34 bytes
- One two-parameters macro → 38 bytes
- One three-parameters macro → 42 bytes
- One two-parameters + one three-parameters macros → 70 bytes
- One two-parameters + one three-parameter + one zero-parameters macros → 90 bytes

In addition, macros do use the memcpy function which would also consume some memory unless needed by other parts of the software as well.

As a practical example for reference, given the same conditions as above, here is the memory requirement at the host when the code example under discussion in this application note (in the SDK under \host_example\spp_server) is used:

- Commands in use:
 - dumo_cmd_system_hello
 - dumo_cmd_system_reset
 - dumo_cmd_bt_rfcomm_start_server
 - dumo_cmd_bt_gap_set_mode
 - dumo_cmd_le_gap_set_mode
 - dumo_cmd_sm_set_bondable_mode
 - dumo_cmd_sm_read_bonding
 - dumo_cmd_endpoint_send
 - dumo_cmd_endpoint_close
 - dumo_cmd_le_gap_set_mode
- Memory requirement with all the above commands in use: approximately 280 bytes

4.3.4 Functional application overview

Before we dig into the host-side code in **main.c**, let's look at what the application is designed to do, and how this is accomplished at a high level. Leaving aside the specifics of how BGAPI communication is handled behind the scenes, the code implements the following behavior:

1. On start-up, the application opens the serial interface defined by the user and verifies that BGAPI communication over the UART interface is proper by issuing the **"dumo_cmd_system_hello()"** API command: as soon as the module successfully responds, the application sends the **"dumo_cmd_system_reset(0)"** API command to reboot the module into a known state. The BGLib implementation translates the latter command into the 5-byte API packet as documented in the API reference material: [**20 01 01 01 00**]. Also all other activity is triggered by incoming API events and responses, so this first response is key, and in case of failure the program is exited. When a reset successfully occurs, the stack on the module will generate the **system_boot** and the **system_initialized** events, which will be sent out of the module's UART port for the host to receive and process.
2. When the **system_initialized** event occurs, indicating that radio portion is fully initialized, the application prints the local mac address which is carried in the event, then starts the RFCOMM server with the command **"dumo_cmd_bt_rfcomm_start_server(2,0)"** where 2 is the ID for the SDP entry as defined in the project.xml file
3. Upon successful start of the RFCOMM server, indicated by the module with the response to the **rfcomm_start_server** command above, the module is made visible to BR/EDR inquiries and connectable via the command **"dumo_cmd_bt_gap_set_mode(1, 1, 0)"**
4. If the above step is successful, next action by the application is to also start BLE advertisements in connectable mode, to allow also remote BLE devices to discover the module and connect to it. This is achieved by using the command **"dumo_cmd_le_gap_set_mode(2, 2)"** (Notice that BLE connections are not under the scope of this application note, however it is still possible to open a BLE connection to module and browse the GATT database, for example to read the device name, manufacturer and model.)
5. When visibility and connectability modes are set for both LE and BR/EDR the application will enable bondable mode with the command **"dumo_cmd_sm_set_bondable_mode(1)"**, which is required at least for the BR/EDR part, and then it will remain waiting for incoming connections. User is notified by this status with the message "Waiting for connection...\n"
6. Upon bonding and/or connection the application will inform the user via messages such as "Bonding from <mac> successful\n " and "SPP connection from <mac> at endpoint <id>\n"
7. In case of an SPP connection the remote side can start to send data, and any such data is exposed to the user after the string "Received data from endpoint <id> " at the shell prompt. At the same time the application will recognize the amount of bytes just received from the remote side and will pass this amount as a response back to the sender with the following message: "Received <amount> bytes\n"
8. The last part of the application's code takes care of recognizing when a *Bluetooth* connection is closed, either in case of BR/EDR via the event **"dumo_evt_endpoint_closing"** or in case of LE via the event **"dumo_evt_le_connection_closed"**. Follow-up actions are respectively to close the RFCOMM endpoint to free it for future use with BGAPI command **"dumo_cmd_endpoint_close"**, or to restart advertisements with a similar **"dumo_cmd_le_gap_set_mode(2, 2)"** as launched at initialization time, given that advertisement mode is always stopped upon LE connection establishment.

All of the API methods and parameters mentioned above and used in the code are described in detail in the BT121 *Bluetooth* Dual Mode API documentation.

4.3.5 Initializing BGLib for sending commands

The **main.c** file uses two important macros, which are defined in the main header file, to set up and initialize the BGLib functionality provided in the SDK's implementation. The first one defines the function prototypes and buffers and state-tracking variables used by the parser and generator:

```
BGLIB_DEFINE();
```

The second one assigns a platform-specific function to BGLib's internal function pointers for UART communication:

```
BGLIB_INITIALIZE(on_message_send);
```

The “**on_message_send**” function is implemented in **main.c**, and controls how data moves from the host to the module over UART. This or similar functions are necessary in *every* instance where BGLib is used. The parameter list for such function is simple:

```
static void on_message_send(uint8 msg_len, uint8* msg_data, uint16 data_len, uint8* data)
```

4.3.6 Incoming messages

As it is implementation specific how UART is accessed, BGLIB does not itself handle reading packets sent by module to the host.

Process to handle incoming packets is as follows:

1. Read **BGLIB_MSG_HEADER_LEN** amount of data to **buffer**
2. Read **BGLIB_MSG_LEN(buffer)** amount of data to buffer after the header
3. Use **BGLIB_MSG_ID(buffer)** to get the message ID, as defined in the form:

```
dumo_evt_system_boot_id
```

Use **BGLIB_MSG(buffer)** to access the payload part of the message. This gives a pointer to the C structure that contains all the messages as union. For example, to access *system boot event* parameter *major* then following code could be used:

```
BGLIB_MSG(buffer)->evt_system_boot.major
```

As a summary, the **BGLIB_DEFINE** and **BGLIB_INITIALIZE** macros are defined along with related macros in the **dumo_bglib.h** file. You must ensure that both of these macros are used in your application code to prepare the environment correctly. Without them, you will encounter many compile errors.

4.3.7 Understanding basic packet structures

Every API packet regardless of type follows the same structure, as defined in the API reference material:

- Header (4 bytes)
 - Technology type and high-length byte
 - Low-length byte
 - Command class byte
 - Command ID byte
- Payload (0 or more bytes)
 - ...depends on packet type

Within BGLib, every incoming packet (responses and events) is referenced using the **dumo_cmd_packet** struct pointer to allow access to the relevant data. This structure contains a **header** member (**uint32** mapped across all 4 header bytes), followed by a union of every possible payload structure. Your code can access data using the correct payload type structure and its members.

The basic format to remember is this:

```
BGLIB_MSG(buffer)->packet_name.member_name
```

4.3.8 Event and response detection and handling

The **main()** function within **main.c** contains all of the application logic which drives this demo. If you are familiar with the way BGScript code is arranged, you might notice that each **case** statement within the **switch** block corresponds to one single API event in exactly the same way that BGScript code uses **event** handlers.

```

* The main program is a simplistic loop where most of the time is
* spent waiting for blocking uart_rx() call to return.
*/
while(1)
{
    /**
     * Read enough data from serial port for the BGAPI message header.
     */
    ret = uart_rx(BGLIB_MSG_HEADER_LEN, buffer);
    ....
    pck=BGLIB_MSG(buffer);
    switch(BGLIB_MSG_ID(buffer))
    {
        case dumo_rsp_system_hello_id:
            /**
             * BGAPI communication channel works, reset the module.
             */
            dumo_cmd_system_reset(0);
            break;
        case dumo_evt_system_boot_id:
            printf("dumo_evt_system_boot_id()\n");
            break;
        case dumo_evt_system_initialized_id:
            printf("dumo_evt_system_initialized_id()\n");
            /**/
            printf("Address:");
            print_address(pck->evt_system_initialized.address);
            printf("\n");
            /*start rfcomm server, use id 2 for sdp entry*/
            dumo_cmd_bt_rfcomm_start_server(2,0);
            break;
            case dumo_rsp_bt_rfcomm_start_server_id:
                ....
    }
}

```

Figure 11: main program's while loop

This kind of program flow keeps all relevant BLE module interaction code in the same place, and also makes it easy to port from on-module BGScript directly to off-module BGAPI/BGLib, in case your development process evolves this way. However, while this example also puts the full case handling code inline inside the **switch** block, it is perfectly acceptable to break that code out into user-defined function calls, or whatever is preferable to enhance code readability and organization.

4.3.9 Sending commands

In addition to catching and handling API events and responses to commands, it is equally important to be able to send the commands. BGLib also makes this easy with built-in commands.

Consider the code below in which the application enables visibility and connectability modes after receiving confirmation that the RFCOMM server has started:

```
....
dumo_cmd_bt_rfcomm_start_server(2,0);
break;
case dumo_rsp_bt_rfcomm_start_server_id:
if (pck->rsp_bt_rfcomm_start_server.result == 0)
{
printf("SPP server started\n");
printf("Setting discoverable, connectable & bondable on\n");
/*Set Bluetooth BR/EDR mode to connectable*/
dumo_cmd_bt_gap_set_mode(1, 1, 0);
}
else
{
printf("SPP server start failed, errorcode %d\n", pck->rsp_bt_rfcomm_start_server.result);
}
break;
```

Figure 12: capturing responses and events then issuing commands

Note specifically the **dumo_cmd_bt_rfcomm_start_server** call and how its return value is stored in the **rsp_bt_rfcomm_start_server.result** variable. This variable is accessed via the **pck** pointer which is defined in the **main.c** file as below:

```
struct dumo_cmd_packet* pck;
```

Every **dumo_cmd_...** function is followed by a corresponding response packet from the module. As mentioned in the end of chapter 4.3.7 above, you can access this data by using the correct packet name and dotted structure member variable.

The application code in **main.c** uses the **pck** pointer to track the events or the responses and is reused for example each time a new command is sent.

Packet pointers and memory persistence



You can use as many packet pointers as you want, but since they are pointers to byte arrays elsewhere in memory, make sure you do not expect them to retain information any longer than the BGAPI protocol data flow allows. In other words, you should assume that any incoming event or response data is not permanent, and may be cleared or overwritten by new incoming data.

If you need to retain any specific packet data for longer periods of time, you should make sure that you copy the relevant values into other data structures.

4.4 Compiling the module firmware project with BGBuild

The firmware application can be compiled with the **BGBuild** compiler using the command prompt.

To start, open a Command Prompt window and change directory to the “**examplebt121**” subfolder where the BT121 SDK is installed. Then, run “**..\..\bin\bgbuild.exe project.xml**” as shown in the screenshot below.

```
Command Prompt
C:\Bluegiga\dual mode-1.6.0-215\example\bt121>..\bin\bgbuild.exe
Qt: Untested Windows version 10.0 detected!
Usage: ..\..\bin\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.
  -m, --mapfile            Generate map file.
  -w, --write-sdp-files    Write SDP entries to .sdp files.
  -r, --root <buildtools> Override default build tools location
  -s, --scompiler <scriptcompiler> Path to script compiler

Arguments:
  input                    Project file to build.

C:\Bluegiga\dual mode-1.6.0-215\example\bt121>..\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/dakarole/AppData/Local/Temp/bgbuild.H19308
C:/Bluegiga/dual mode-1.6.0-215/bin/./fw/libbt121.a
in :C:/Users/dakarole/AppData/Local/Temp/bgbuild.p19308
UART
baudrate :115200
actual :115107
error% :0.081
flowcontrol:true
parity :none
stop bits :1
BGAPI :true
BRR :0x1a1
CR1 :0xd
CR2 :0x0
CR3 :0x300
RTOR :0x1c200
PORTA
MODER :feabffff
OTYPER :0
PUPDR :0
ODR :0
AFRL :0
AFRH :11110
PORTB
MODER :ffffff
OTYPER :0
PUPDR :0
ODR :0
AFRL :0
AFRH :0
EXTI
EXTICR1 :0
EXTICR2 :0
EXTICR3 :0
EXTICR4 :0
EMR :0
IMR :0
RTSR :0
FTSR :0
LE serial and RFCOMM bridging Disabled :

BOOTLOADER:
in :C:/Bluegiga/dual mode-1.6.0-215/bin/./fw/boot.juo
bootdfu :C:/Bluegiga/dual mode-1.6.0-215/bin/./fw/boot2.juo

Memory used:
SW :114312
HW :143
Files :371
Total :114866
Max :122880
Free :8014/122880(7%)
ALL OK

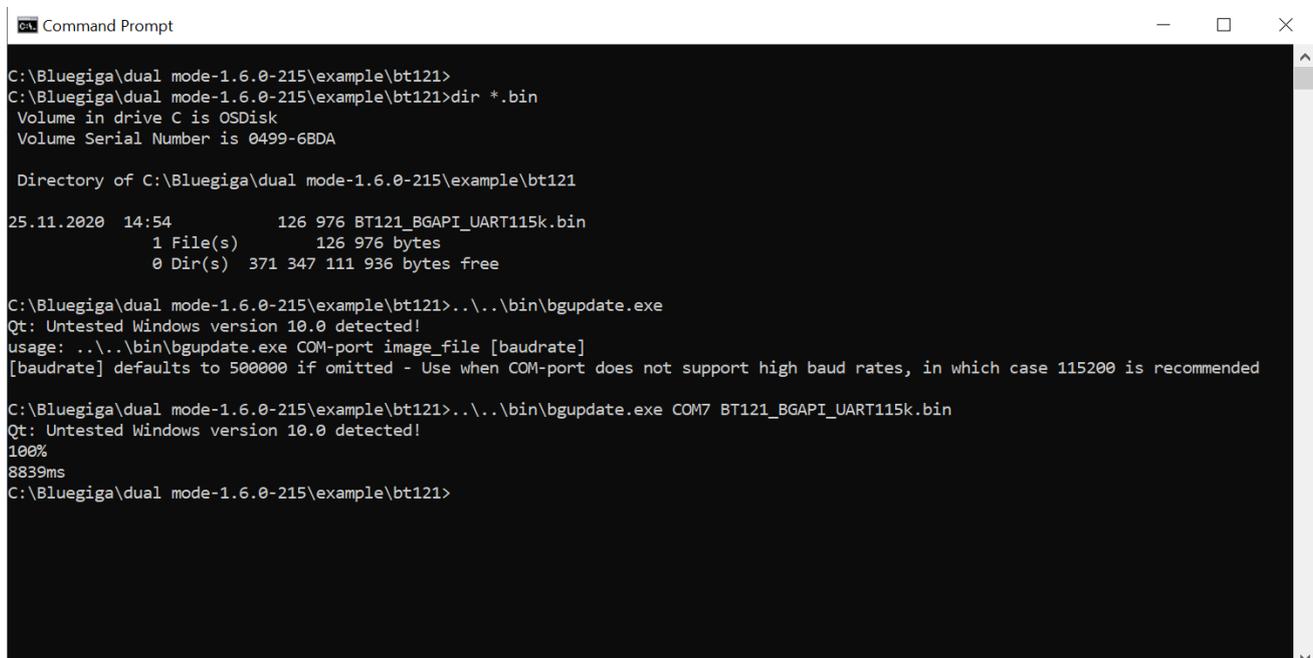
C:\Bluegiga\dual mode-1.6.0-215\example\bt121>
```

Figure 13: Compiling the demo application

4.5 Flashing the new firmware into the *Bluetooth* Dual Mode hardware

The **bgupdate.exe** utility from the **\bin** directory of the SDK is used to upload the new firmware to a module in a carrier board of the evaluation kit. With this particular example, the new firmware image file will be given the name of *BT121_BGAPI_UART115k.bin* in accordance to the project.xml file. At the Command Prompt window make sure you are still at, or change directory to the “**\example\bt121**” subfolder where the BT121 SDK is installed, then run “**..\..\bin\bgupdate.exe COMx BT121_BGAPI_UART115k.bin**” where x is the virtual COM port number assigned by the OS to the USB-serial converter of the main evaluation board. You might need to make sure that board’s power switch is set to USB and that also the “USB TO UART CONVERTER” switch is set to ON – When converter’s driver is installed correctly, the virtual COM’s port number is available in the Windows’ Device Manager. The screenshot below shows an example usage of the utility.

The syntax for the updating utility is shown here: **bgupdate.exe COM<port_number> <image_file>**



```
Command Prompt
C:\Bluegiga\dual mode-1.6.0-215\example\bt121>
C:\Bluegiga\dual mode-1.6.0-215\example\bt121>dir *.bin
Volume in drive C is OSDisk
Volume Serial Number is 0499-6BDA

Directory of C:\Bluegiga\dual mode-1.6.0-215\example\bt121

25.11.2020  14:54                126 976 BT121_BGAPI_UART115k.bin
             1 File(s)                126 976 bytes
             0 Dir(s)  371 347 111 936 bytes free

C:\Bluegiga\dual mode-1.6.0-215\example\bt121>..\..\bin\bgupdate.exe
Qt: Untested Windows version 10.0 detected!
usage: ..\..\bin\bgupdate.exe COM-port image_file [baudrate]
[baudrate] defaults to 500000 if omitted - Use when COM-port does not support high baud rates, in which case 115200 is recommended

C:\Bluegiga\dual mode-1.6.0-215\example\bt121>..\..\bin\bgupdate.exe COM7 BT121_BGAPI_UART115k.bin
Qt: Untested Windows version 10.0 detected!
100%
8839ms
C:\Bluegiga\dual mode-1.6.0-215\example\bt121>
```

Figure 14: flashing with the bgupdate.exe

4.6 Compiling the host application project with Visual Studio

The host project source files for this example can be found in the `host_example\spp_server\` subfolder from the SDK. Once you have [Microsoft Visual Studio](#) installed on your PC, you can simply open the Solution (`.sln`) file from this folder, and it will open directly in the application.

Here is what it looks like in Visual Studio 2013, after opening the project and then the “main.c” source file:

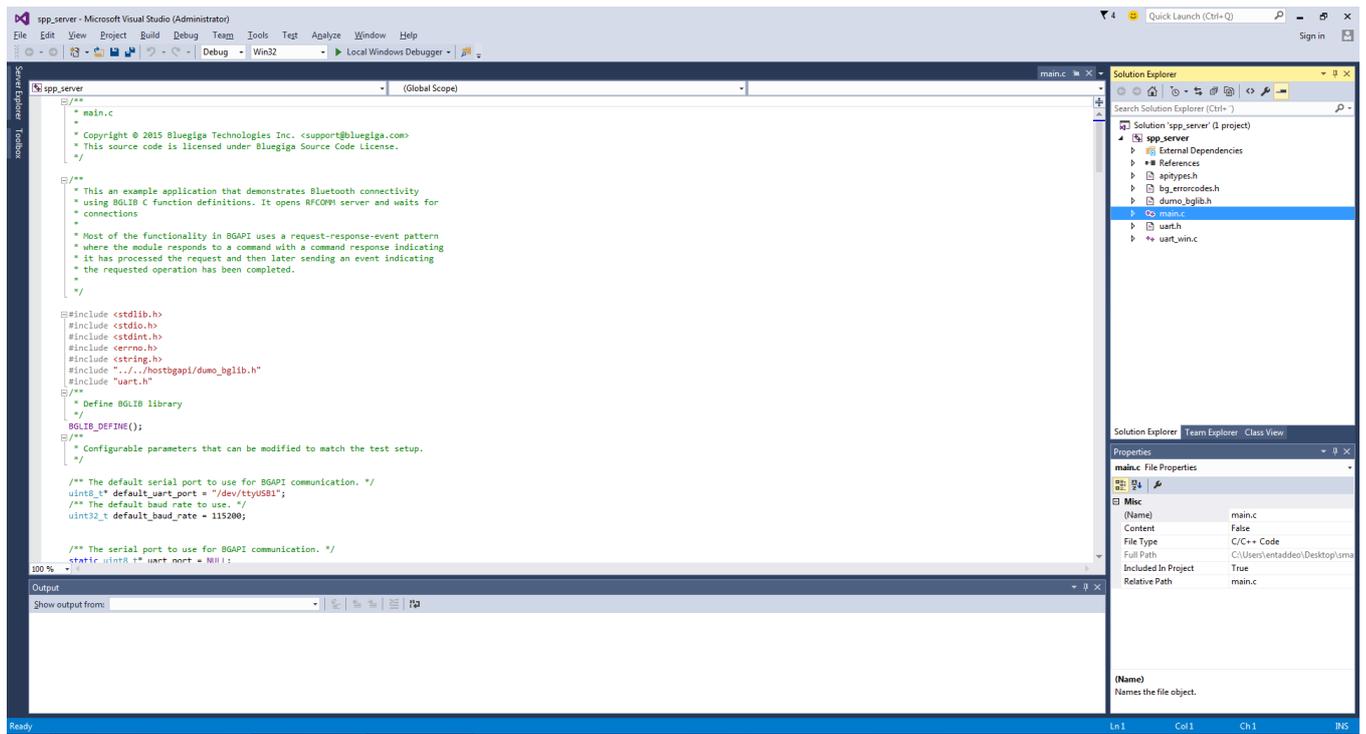


Figure 15: Visual Studio project view

You can compile the project using the BUILD menu and the **Build Solution** item, or by pressing **Ctrl+Shift+B** to perform the same action.

Likewise, you can compile and run the project at the same time automatically simply by using the DEBUG menu and the **"Start Debugging"** menu item, or by pressing the **F5** key.

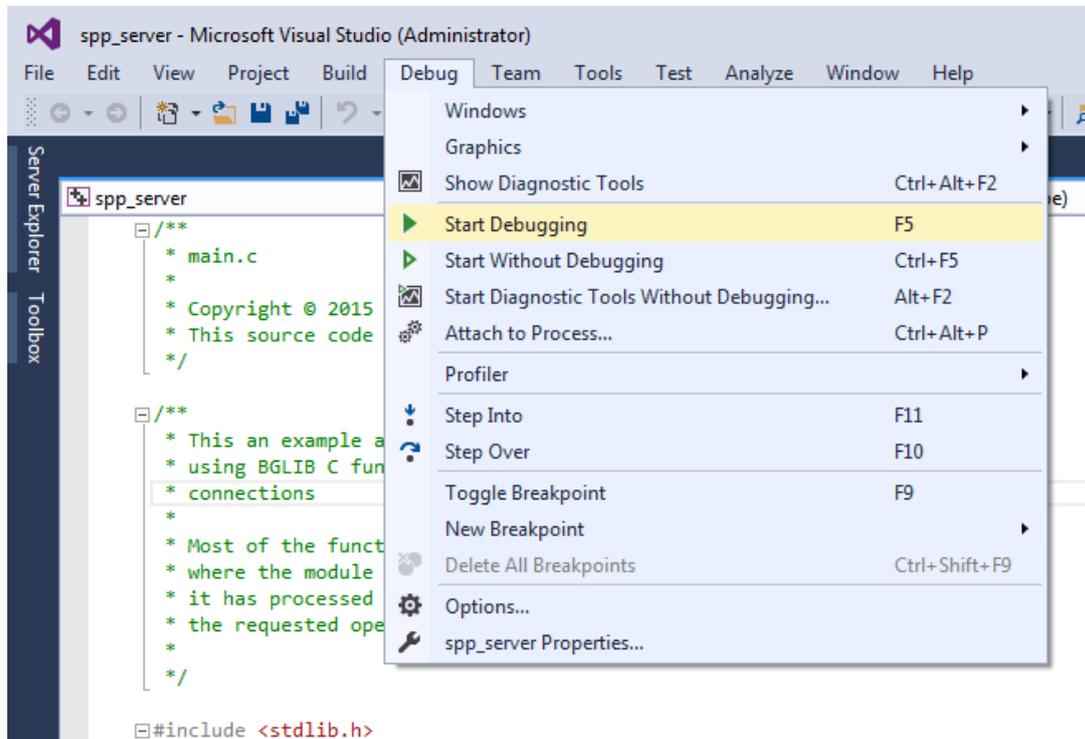


Figure 16: Compiling and running in Visual Studio



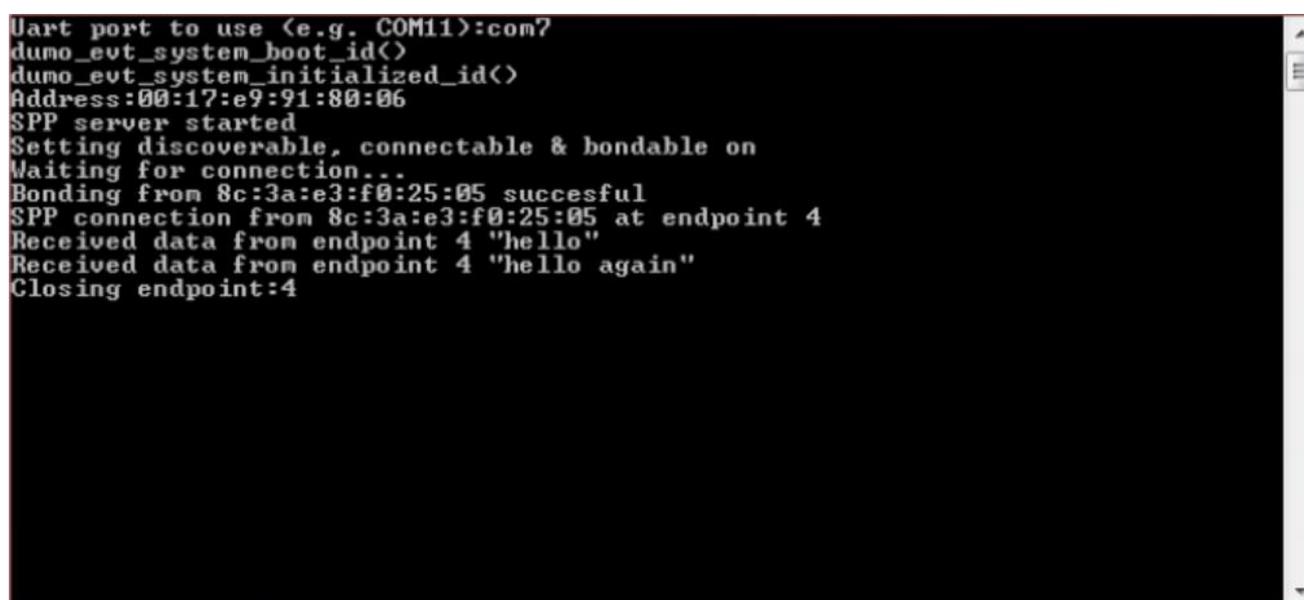
The source code takes advantage of the **_DEBUG** compiler macro exposed by the Visual Studio compiler depending on whether you have selected the **Debug** or **Release** project configuration. While both will work, the **Debug** configuration will generate much more console output during testing. The steps below show the output when using the **Debug** configuration.

4.7 Running the host application on Windows

Once you have compiled the host application in Visual Studio, there will be a new executable file called **spp_server.exe** found in the **/host_example/spp_server/Release** folder (or **/Debug** if you chose the Debug configuration instead). To run the host application on a PC, do the following:

- Ensure the module in the carrier board has been flashed with the corresponding module firmware project
- Connect the DKBT main board to the PC via a micro USB cable
- Make sure the power switch on the main board is set to the **USB** position
- Run the “spp_server.exe” file that is in the relevant **/Release** or **/Debug** subfolder
- Enter the correct COM port

At this point simply review the on-screen messages as the module initializes, then wait for more messages as remote devices pair, connect, and send data, as seen in the following screenshot.

A screenshot of a console application window with a black background and white text. The text shows the following sequence of events: a prompt for a UART port (COM7), system boot and initialization events, the SPP server starting, setting discoverability, waiting for a connection, a successful bonding event from a specific MAC address, an SPP connection established at endpoint 4, and two data packets received from that endpoint: "hello" and "hello again". The application then closes endpoint 4.

```
Uart port to use (e.g. COM11):com7
dumo_evt_system_boot_id()
dumo_evt_system_initialized_id()
Address:00:17:e9:91:80:06
SPP server started
Setting discoverable, connectable & bondable on
Waiting for connection...
Bonding from 8c:3a:e3:f0:25:05 succesful
SPP connection from 8c:3a:e3:f0:25:05 at endpoint 4
Received data from endpoint 4 "hello"
Received data from endpoint 4 "hello again"
Closing endpoint:4
```

Figure 17: SPP Server demo console application

Any steps which requires some interaction from a remote device is briefly described in the console output, such as the “**Waiting for connection...**” note shown in the screenshot above. You can follow these prompts and the functional application overview from chapter 4.3.4 above to observe the interaction that this demo provides, and refer to the detailed instructions for client-side testing from a mobile device in the next section.

4.8 Testing client connectivity with an Android smartphone

The module firmware and host application project are principally designed to allow a remote device to connect to the Dual Mode module using the BR/EDR Serial Port Profile for the exchange of data.

This section describes how to do this using an Android smartphone and the app called *S2 Bluetooth Terminal*.



It is possible to perform the steps outlined below using any standards-compliant *Bluetooth* client device; this choice of hardware and software is chosen since it is fairly common and works reliably. An alternative includes using a PC equipped with *Bluetooth* hardware and a stack supporting SPP, such as any Windows-based PC since XP-SP2

4.8.1 Scanning and pairing to the Dual Mode module

To begin, first ensure that all firmware compilation/flashing is completed as described in previous sections, and that the host application is running on the PC that the DKBT board is connected to. The console output should indicate that the device is waiting for connections from a remote device. Once this is prepared:

1. Go to the *Bluetooth* settings in the configuration section of the Android smartphone
2. A device under the name of **BT121** should appear in the list of discovered devices
3. Click it to pair the smartphone with the device
4. Exit the configuration page and start the app called *S2 Bluetooth Terminal*
5. From within the app connect to the paired BT121
6. Send some data from the app and observe the characters sent in return by the module
7. Eventually instruct the app to disconnect

Screenshots of the smartphone at the different stages listed above are shown in the next two pages.

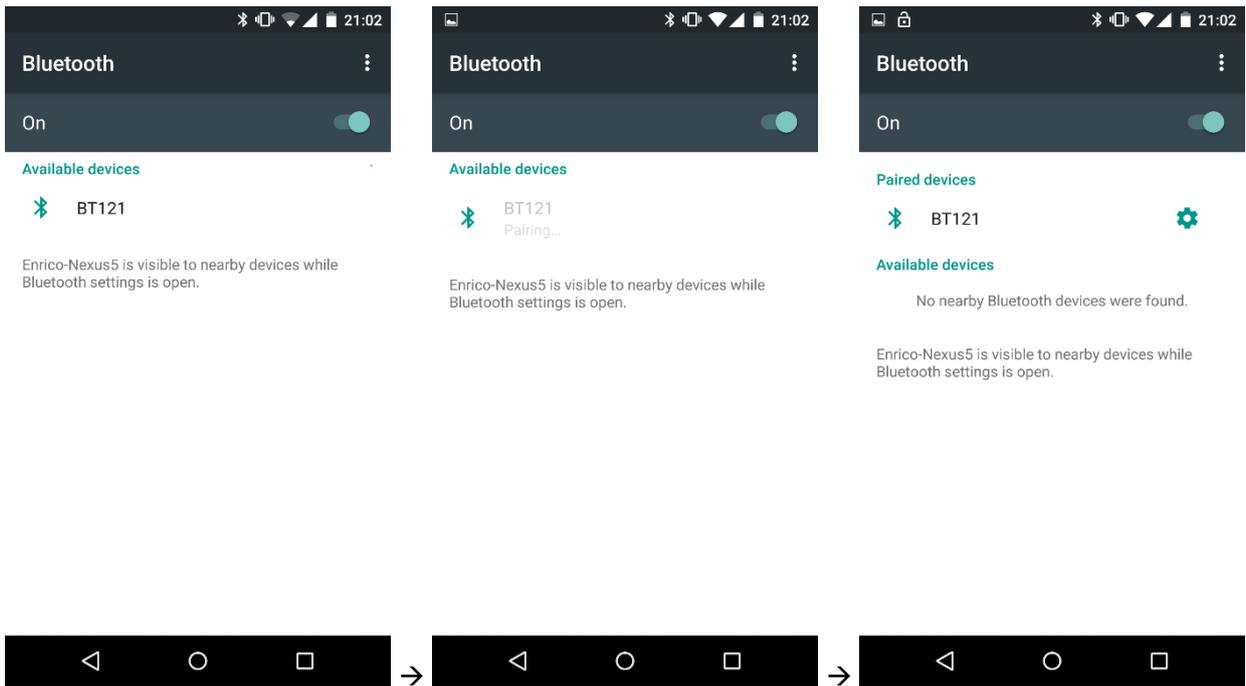


Figure 18: Android smartphone scanning and pairing with the BT121

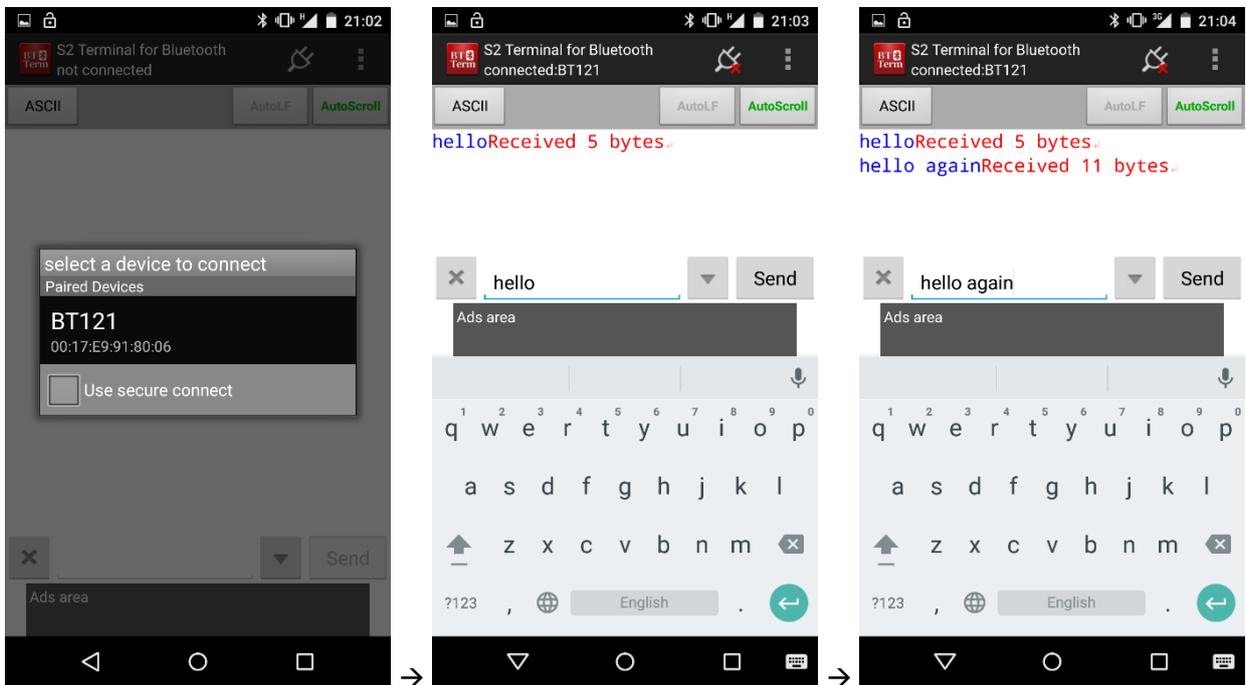


Figure 19: S2 Terminal app used to establish SPP connection and send “hello” messages

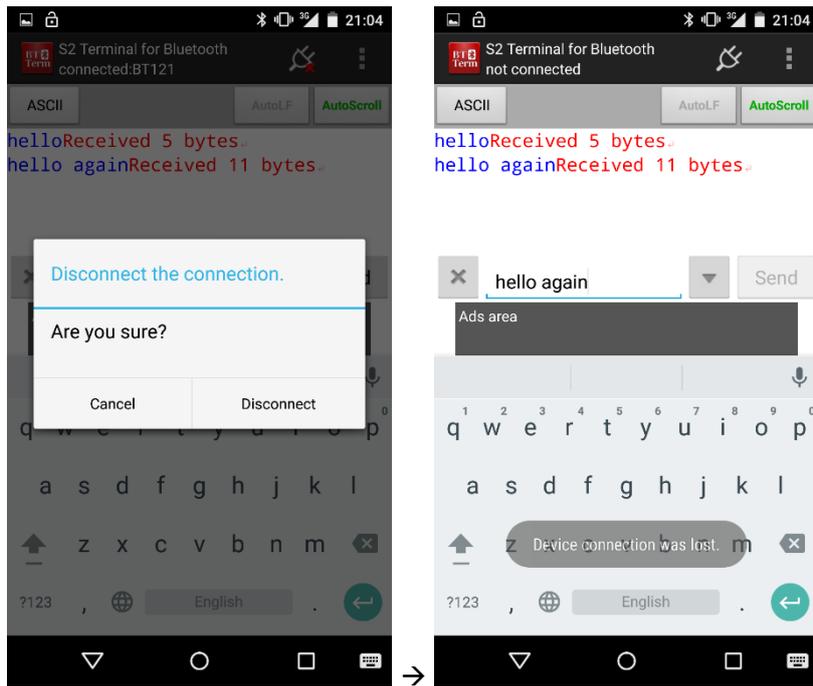


Figure 20: disconnecting the SPP connection from the app

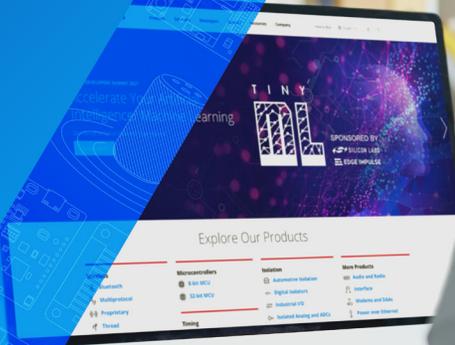
5 Summary and BGAPI quick-reference guide

This document shows the steps required to implement a BGLib/BGAPI-based design using a Windows PC host for the host application. If you are implementing a project on a different platform, keep the following checklist handy for reference, while remembering that module firmware and host application are separate projects that interact with each other over UART:

- Module firmware project
 - Firmware requires UART enabled with BGAPI control configured.
 - Flow control between host as modules is recommended if at all possible.
 - BGScript is not normally used for *Bluetooth* Dual Mode projects which are built on BGAPI/BGLib.
 - Named characteristic IDs are generated as **#define**'d constants in a "gatt_db.h" file generated as a result of the compile process, which you can copy or include in your host application.
 - The module will boot into an idle state and requires commands sent from the host before it will exhibit any other behavior. Usually this means starting advertisements or something similar.

- Host application project
 - Include all of the required BGLib source and support files in your main project.
 - Define platform-specific UART RX/TX routines with length/data arguments.
 - Use **BGLIB_DEFINE** and **BGLIB_INITIALIZE** macros to set up the environment.
 - All packets have one 4-byte header, followed by zero or more payload bytes.
 - Create an event-catching loop and switch/case statement (or "if" block) for all events needed.
 - Use packet pointers to keep track of events and responses.
 - Use the **data** structure pointers and members to access event and response parameters:
`obj -> packet_name.member_name`
 - The SDK's provided BGLib implementation may be substituted with a custom implementation as long as it can correctly parse and create command, response, and event packets.

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