HID OVER *BLUETOOTH* BR/EDR

APPLICATION NOTE FOR BT121

Friday, 18 June 2021

Document Revision: 1.3

VERSION HISTORY

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

TABLE OF CONTENTS

1 Introduction

1.1 About the application note

This application note discusses how to configure and use a BT121 Dual Mode module in order to add the Human Interface Devices (HID) functionality to a device using the *Bluetooth* BR/EDR technology (often referred to as *Bluetooth* BR/EDR) and using the related HID Profile adopted by the *Bluetooth* Special Interest Group (SIG) as seen at https://www.bluetooth.com/specifications/adopted-specifications (under the “Traditional Profiles” section). Devices are typically keyboards or mouses, which are the focus of this application note, but can also be joysticks, digitizers and all others as defined by the USB Implementers Forum, Inc. at http://www.usb.org/developers/hidpage

The reference hardware platform used in the application note is the DKBT *Bluetooth* Dual Mode Development Kit (in short DKBT) and the BT121 *Bluetooth* Dual Mode module.

The example application discussed in this document is developed with the Bluegiga BGScript™ scripting language, which enables the application to run fully autonomously on the BT121 *Bluetooth* Dual Mode module without the need for an extra host MCU.

This application note contains references to the commands and events of the BGAPI serial host protocol and of the BGScript scripting language for in-module applications. The description of these commands and events is found in the document called “*Bluetooth* Dual Mode Software API Reference” and also in the HTML-based BGAPI reference called dumo.html which is available under the directory /hostbgapi/ of the Dual Mode SDK being used to generate the custom firmwares for the BT121.

Although this application note uses a BGScript script example to describe the application logic it is also possible to implement the same application on an external MCU (such as low-power MCU) using the BGAPI serial protocol and BGLIB host library.

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

1.2 About the *Bluetooth* HID BR/EDR Profile

The HID *Bluetooth* Profile defines the protocols, procedures and features to be used in the communication between devices such as keyboards, mouses, joysticks etc. and hosts such as PCs, smartphones, tablets etc.

In the HID *Bluetooth* Profile two roles are defined:

- HID-Device – The device providing the service of human data input and output to and from the host.
- HID-Host – The device using or requesting the services of a Device.

The HID *Bluetooth* Profile uses USB definition of a HID device in order to leverage the existing class drivers for USB HID devices. The HID *Bluetooth* Profile describes how to use the USB HID protocol to discover a HID class device's feature set and how a Bluetooth enabled device can support HID services on top of the *Bluetooth's* L2CAP layer. The HID *Bluetooth* Profile is designed to enable initialization and control self-describing devices as well as provide a low latency link with low power requirements.

The HID *Bluetooth* Profile is built upon the Generic Access Profile (GAP). In order to provide the simplest possible implementation, the HID protocol runs natively over L2CAP and does not reuse *Bluetooth* protocols other than the Service Discovery Protocol.

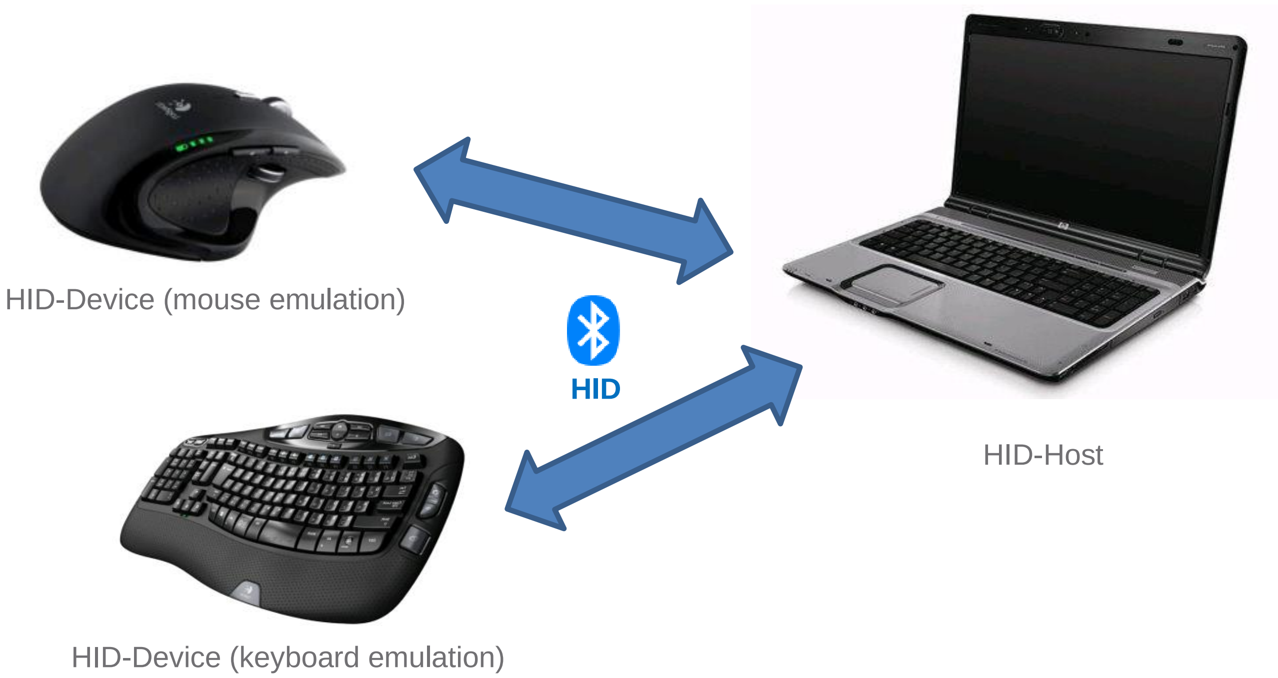


Figure 1: Typical HID use case

2 HID features in the *Bluetooth* Dual Mode stack

2.1 General main features of the BT121 stack

The table below summarizes the main features of the Dual Mode stack for the BT121 modules:

Feature	Value
Simultaneous BR/EDR connections	6
Simultaneous LE connections	7
Combined BR/EDR and LE connections	1 x BR/EDR + 6 x LE
MAX data rate	1000 kbps SPP (transparent mode using BGScripting) 700 kbps SPP (BGAPI mode) 200 kbps iAP2 (both transparent and BGAPI mode)
MAX UART baud rate	4 000 000 bps
Data transmission delay over BR	5-10 ms
Supported encryptions	E0 for *Bluetooth* BR/EDR (128-bit) AES-128 for *Bluetooth* LE
MAX simultaneous pairings	32
MAX Friendly name length	Configurable up to 30 characters
L2CAP packet size	255 bytes
Supported *Bluetooth* BR/EDR profiles	GAP, DI, SPP, HID-Device and GATT-over-BR/EDR
Supported *Bluetooth* LE profiles	Any with Profile Toolkit, SPP-over-BLE proprietary solution also available
Apple iAP support	iAP2 (Devices with Lightning connector)
Supported power saving modes	Sniff mode, Peripheral latency and two MCU sleep modes
Secure Simple Pairing mechanisms	Just works mode Man-in-the-middle (MITM) protected modes

Table 1: *Bluetooth* Dual Mode stack features

2.2 HID features

The Dual Mode stack running in the BT121 allows HID-Devices to be easily implemented. Notice that the stack does not offer support for implementing HID-Hosts.

Using the Dual Mode SDK it is possible to create a firmware embedding the desired HID SDP record carrying among others the desired HID descriptor. In other words, HID-based SDP records and descriptors are fully configurable by the user and are stored in the Flash memory of the module at the time the module is programmed. Programming of a module happens with a custom firmware obtained from the SDK using the appropriate project files that are described later in this document. SDP records can be loaded and unloaded at any time during runtime, according to the user application requirements, using dedicated BGAPI commands.

Multiple BGAPI commands (and corresponding responses/events) exist for the management of HID connections and for issuing (and parsing) input and output HID reports. Most of the available BGAPI commands and events related to the HID functionality will be mentioned in this document as well.

Virtual cable unplugging is also supported, allowing the local or the remote party do disassociate so that both parties will disconnect first, and then remove each other's pairing/bonding information from their databases.

3 Prerequisites for developing and testing

Before you can test the example script-based application discussed later, or before starting to develop your own solution, the following prerequisites should be met:

1a. You have downloaded and installed the latest *Bluetooth* Dual Mode SDK for Windows and you have gone through the Getting Started guide. Notice that the support for the HID BR/EDR Profile has been introduced in version 1.1.0

1b. After SDK installation you have localized the two HID examples, normally under the directory \example\ of the SDK, and you have verified that the BGTool Windows demo program can be launched. Quick access to the directory containing the firmware project files examples and to the BGTool is in the dedicated Windows Start menu entry created by the SDK installer

2a. You have a DKBT *Bluetooth* Dual Mode Development Kit

2b. Alternatively you have a BT121 *Bluetooth* Dual Mode module already soldered in your PCB according to the guidelines in the module's data sheet

3a. You have a PC equipped with *Bluetooth* hardware and running an OS (or third party software) which provides the HID-Host support

3b. Alternatively, or in addition, you have a modern smartphone or a tablet. Make sure that *Bluetooth* functionality is indeed integrated in the device, while notice as well that iOS does not support HID-Device for the mouse case, as of the date of release of this document.

4. You have already practiced the steps of editing some of the existing project files for creating your own custom firmware, and you are familiar with the firmware image creation using the bgbuild.exe and with the firmware re-flashing of the module, for example using the bgupdate.exe. Both .exe executables are found under the directory \bin\ of the SDK. Remember as well that building firmware form given project files plus upgrading modules can be performed also from within the BGTool program, under the "Upload tool" tab.

4 Configuring and using the HID Profile

In order to turn a device equipped with the BT121 into a *Bluetooth* HID-Device, first of all a new firmware must be uploaded to the module, where such firmware is obtained from project files which are appropriately created and/or edited from existing files, as discussed below.

Once the appropriate firmware is running in the module, some special initialization commands are given to it, either by the host system when module is used in Network Co-processor (NCP) mode, or by a BGScript running in the module itself.

When finally all is readily configured, the HID-Host will be the side to make the first connection, because together with pairing and bonding the HID-Host will also need to read through the BT121's loaded SDP record using the Service Discovery protocol. This SDP procedure is needed for the HID-Host to learn what kind of HID device the BT121 is in fact, and to also learn (by parsing the HID Descriptor included in the record) how to interpret the data coming from the module (HID input reports) and how to format the data to send to module (HID output reports). Once devices are bonded, and HID-Host knows about the HID-Device, then any future connection can be started by any of the sides, and the same is valid for simply closing a current connection or for the virtual cable unplug operation.

For a clearer understanding of this chapter it is recommended to refer to the existing example project files for the NCP mode found under the directory \example\bt121_br_edr_hid\ of the SDK.

4.1 Configuration of firmware project files

A typical firmware for the BT121 in NCP mode is created from a minimum of four .xml files, plus as many .xml files as you want SDP records to be available for loading/unloading during runtime. The four files are:

- the *project.xml* which collects all the information for the bgbuild.exe to create the desired firmware image file, where the filesystem paths to the other project files are provided among others

- the *hardware.xml* which contains the hardware configuration, for example the UART settings and if the BGAPI protocol is enabled over it or not (use the option bgapi="true" for the NCP mode)

- the *gatt.xml* which contains the definition of the GATT Database for *Bluetooth* Low Energy operations (not under discussion in this application note)

- the *did.xml* which defines the mandatory record for the *Bluetooth* BR/EDR's Device Information profile

Among the above mentioned files, the first one is relevant for making sure that the obtained firmware has HID functionality available in it. In particular, the *project.xml* should contain the following additional entries when compared for example with the project file for the factory default firmware of the bare modules (actually the one under the directory \example\bt121\ of the SDK):

- instruct firmware compiler to include the base firmware containing the HID functionality with:

```
<software>
  <library in="bt121_hid"/>
</software>
```

- instruct firmware compiler to include the desired HID SDP record(s) into the firmware image with:

```
<sdp>
  <entry file="did.xml" autoload="true"/>
  <entry file="hid_mouse.xml" id="4"/>
  <entry file="hid_keyboard.xml" id="5"/>
  <entry file="my_hid_device.xml" id="6"/>
</sdp>
```

where the *file*= parameter points to the .xml file containing the SDP record definitions and where the *id*= indicates a number which will be used to identify the desired record when it will come the time to load (or unload) it during runtime. Notice here that *autoload*= parameter cannot be used with HID-based .xml files, meaning that HID SDP records must be loaded during runtime. Notice also that only one of the HID SDP records can be loaded at any time, because the *Bluetooth* HID specification mandates that one and only one HID service may exist on a device. If for example a combination mouse-keyboard device is required, both the mouse and the keyboard definitions must be combined into a single SDP record (and into a single HID Descriptor)

In addition to the above four project files, two more files are to be used for each HID service that we want our BT121 to support. The first of the two is the .xml file mentioned in the project.xml and it has to be formatted according to the two tables below.

.xml File Contents	Description
<ServiceClassIDList> <ServiceClass uuid128="1124"/> </ServiceClassIDList>	This defines the UUID of the *Bluetooth* profile. For the HID profile the UUID must be 1124 and **should not be changed**.
<BrowseGroupList> <UUID16 value="1002"/> </BrowseGroupList>	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.
<ProtocolDescriptorList> <Protocol> <UUID16 value="0100"/> <UINT16 value="0011"/> </Protocol> <Protocol> <UUID16 value="0011"/> </Protocol> </ProtocolDescriptorList>	value="0100" means this profile is based on top of L2CAP the first value="0011" refers to the PSM for HID Control the second value="0011" refers to the Protocol Identifier's UUID **You should not change this section**.
<ServiceName text="BT121 Mouse" language_id="0100"/>	This entry defines the service name for the SDP record. If you want to rename the service you can modify the value of the text= attribute
<LanguageBaseAttributeIDList> <UINT16 value="656e"/> <UINT16 value="006a"/> <UINT16 value="0100"/> </LanguageBaseAttributeIDList>	value="656e" is for "en" - English value="006a" is for UTF-8 encoding value="0100" is to define PrimaryLanguageBaseId = 0
<BluetoothProfileDescriptorList> <Profile> <UUID16 value="0011"/> <UINT16 value="0101"/> </Profile> </BluetoothProfileDescriptorList>	value="0011" refers to the Protocol Identifier's UUID for the HID profile value="0101" is to define the version to 1.1 **You should not change this section**.
<AdditionalProtocolDescriptorLists> <AdditionalProtocolDescriptorList> <Protocol> <UUID16 value="0100"/> <UINT16 value="0013"/> </Protocol> <Protocol> <UUID16 value="0011"/> </Protocol> </AdditionalProtocolDescriptorList> </AdditionalProtocolDescriptorLists>	value="0100" means this profile is based on top of L2CAP value="0013" refers to the PSM for HID Interrupt value="0011" refers to the Protocol Identifier's UUID for the HID profile **You should not change this section**.

Table 2: Main SDP record entries for HID

.xml File Contents	Description
<HIDParserVersion value="0111"/>	HIDParserVersion It has fixed value of 0x0111 in the current BT HID specification
<HIDDeviceSubclass value="80"/>	HIDDeviceSubclass This must match bits #2 to #7 of the Class of Device, bits #0 and #1 are set to zero Common values are 0x40 for keyboard, 0x80 for pointing device For a comprehensive list see Tables 9-10 (Minor Device Class field - Peripheral Major Class) at https://www.bluetooth.com/specifications/assigned-numbers/baseband
<HIDCountryCode value="0"/> *<HIDCountryCode value="21"/>* (0 for not localized, typical for mouse; 21 for US-style keyboard)	HIDCountryCode 0x00 for non-localized devices For localized devices such as keyboards, see the USB country code list at http://www.usb.org/developers/hidpage/HID1_11.pdf, section 6.2.1
<HIDVirtualCable value="1"/>	HIDVirtualCable Indicates whether the Device should be associated with only one Host at a time, like a wired keyboard can be connected to only one computer Enabling this means your device should never store the pairing information of more than one Host at a time If enabled, your device MUST also support either HIDReconnectInitiate or HIDNormallyConnectable
<HIDReconnectInitiate value="1"/>	HIDReconnectInitiate Indicates whether the Device can reconnect to the Host
<HIDDescriptorList> *<HIDClassDescriptor>* *<UINT8 value="22"/>* *<HIDUSBDescriptor file="hid_mouse.txt"/>* *</HIDClassDescriptor>* *</HIDDescriptorList>* (value="22" is for type of Report Descriptor; hid_mouse.txt contains the actual descriptor as text string, not DataElementSequence)	HIDDescriptorList Defines the HID Descriptor itself The easiest way to create and validate one is to use the USB HID Descriptor Tool (http://www.usb.org/developers/hidpage#HID%20Descriptor%20Tool) and export the descriptor in .txt format The descriptors are pre-defined in the USB HID specification; they are not listed in the HIDDescriptorList
<HIDLANGIDBaseList> *<HIDLANGIDBase>* *<UINT16 value="0409"/>* *<UINT16 value="0100"/>* *</HIDLANGIDBase>* *</HIDLANGIDBaseList>* (value value="0409 is for en-US; value="0100" is the Bluetooth String Offset)	HIDLANGIDBaseList
<HIDBatteryPower value="1"/>	HIDBatteryPower Indicates whether the Device is battery powered
<HIDRemoteWake value="1"/>	HIDRemoteWake Indicates capability to wake up the Host from Suspend, if supported by the Host, which in practice requires two things: The Device can send an Exit Suspend command upon user input, if the Host doesn't disconnect the *Bluetooth* link while in Suspend mode Reconnect upon user input, if the Host disconnects the *Bluetooth* link when entering Suspend mode

<HIDNormallyConnectable value="0"/>	HIDNormallyConnectable Indicates whether the device normally accepts incoming connections from the host Generally for battery-powered devices this should be false, because scanning for paging consumes battery power
<HIDBootDevice value="1"/>	HIDBootDevice Indicates whether the Device implements either the Boot Keyboard or Boot Mouse, or both Mandatory to support for keyboards and mice devices

Table 3: Additional mandatory SDP record entries for HID

As seen from the second of the two tables above, the service record .xml definition contains an entry to indicate an additional text file. This is the mentioned second file of the two for each service record, which contains the actual HID Descriptor formatted according to the USB HID specification. A couple of examples of such files are found in the SDK (hid_keyboard.txt and hid_mouse.txt) The user can freely edit the HID Descriptor in the text file for its own custom application, however it is recommended to conform to the standard, and make use for example of the USB HID tools found at http://www.usb.org/developers/hidpage/ or http://hidedit.org/

4.2 *Bluetooth* and HID service initialization after boot

When the basic NCP firmware is running, the module is idle until its host MCU issues a set of initialization BGAPI commands over UART. For the HID case these initialization commands are listed below (for an additional description of the commands you might want to refer to the BGAPI reference):

- ***dumo_cmd_sm_configure(0,3)*** to configure the local Secure Simple Pairing (SSP) capabilities (in this case, configuration is meant to use the “Just Works” pairing mechanism)

- ***dumo_cmd_sm_set_bondable_mode(1)*** to allow the local module to store the link keys generated during pairing (the same will happen at the remote side and the devices will be considered bonded - this is mandatory with *Bluetooth* BR/EDR)

- ***dumo_cmd_system_set_class_of_device(0x580)*** to configure the Class-of-Device (CoD) reported by the module in *Bluetooth* BR/EDR’s Inquiry Responses. Based on the CoD the PC or the smartphone will get an initial understanding of the kind of HID-Device is found when searching for *Bluetooth* devices in range, and for example will display the correct icon next to it. Typical CoD is 0x580 for mouse or 0x540 for keyboard. A CoD can be realized by looking at the *Bluetooth* Specification, in particular at https://www.bluetooth.com/specifications/assigned-numbers/baseband

- ***dumo_cmd_bt_gap_set_mode(1,1,0)*** to make the module visible and connectable

- ***dumo_cmd_bt_hid_start_server(4,0)*** to load the SDP record with the given ID from the Flash memory so to make it available for remote SDP queries, and to get the HID service started (in this example the record to load has ID of 4 and corresponds to the information in the .xml file that was assigned ID of 4 in the project.xml). Let’s not forget that only one HID SDP record can be loaded at any time, so if you need to switch between services, like alternating mouse and keyboard, you will have to stop the currently loaded service with the command ***bt_hid_stop_server(4)*** before loading the new one. The command to start the service also contains an additional parameter (mentioned as destination endpoint in the API reference) which is reserved for future use and should always be set to 0.

4.3 Starting outgoing connections and capturing incoming connections

Once the module has been initialized, it will be ready to receive the HID connection from a HID-Host in case there is none already bonded, or it will be ready to start and accept connections to/from bonded HID-Hosts at any time.

In the former case the expected sequence of BGAPI events sent from module to its host MCU over the UART interface is shown below:

- ***dumo_evt_bt_connection_opened(remote_address, central_no, bt_connection_id, bonding_none)***
- ***dumo_evt_sm_bonded(bt_connection_id, bonding_id)***
- ***dumo_evt_endpoint_status(hid_endpoint_id, hid_type, destination_endpoint_id, flags)***
- ***dumo_evt_bt_hid_opened(hid_endpoint_id, remote_address)***

Depending on the HID-Host and the HID Descriptor being presented, the HID-Host may send an Output Report, indicated by the event *dumo_evt_bt_hid_output_report*, or may request the current status of the fields of the Input Reports, indicated by the event *dumo_evt_bt_hid_get_report*. With the latter, the module must respond with the command *dumo_cmd_bt_hid_get_report_response* in a timely manner.

When devices are already bonded, any subsequent incoming connection will be indicated by the following events:

- ***dumo_evt_bt_connection_opened(remote_address, central_no, bt_connection_id, bonding_id)***
- ***dumo_evt_endpoint_status(hid_endpoint_id, hid_type, destination_endpoint_id, flags)***
- ***dumo_evt_bt_hid_opened(hid_endpoint_id, remote_address)***

When devices are already bonded, the host MCU can also instruct the BT121 to place calls to the remote HID-Host using the following command:

- ***dumo_cmd_bt_hid_open(remote_address, destination_endpoint_id)***

The module in turn will send to its host the BGAPI response to the command with any error code (0x0 for success) and with the anticipated ID for the soon-to-be-created HID endpoint. As soon as the connection is fully established the usual events *dumo_evt_bt_connection_opened* + *dumo_evt_endpoint_status* + *dumo_evt_bt_hid_opened* are to be expected.

In all the cases above, and similarly to the command *dumo_cmd_bt_hid_start_server* seen in the previous chapter, the parameter called *destination_endpoint_id* is reserved for future use and is or should always be set to 0.

4.4 Sending data via HID reports

As soon as the module's host is notified of the existing HID connection, by mean of the events listed above, the data exchange in the form of HID reports can be started.

The most typical BGAPI command used is the ***dumo_cmd_bt_hid_send_input_report*** which is meant for the module implementing the HID-Device to send the so-called HID input reports to the HID-Host. These HID input reports carry for example the information about which keys are pressed in the case of the *Bluetooth* HID keyboard or the pointer movements and button clicks by a *Bluetooth* HID mouse.

The *dumo_cmd_bt_hid_send_input_report* takes three parameters, the first being the ID of the current HID endpoint (referred previously as *hid_endpoint_id* and not to be confused with the ID of the *Bluetooth* connection, referred to as *bt_connection_id*), and the second being the Report ID which is defined in the HID Descriptor and is recognized easily in the text project file like in the SDK examples hid_keyboard.txt and hid_mouse.txt

As for the third parameter of the above mentioned command, this is the actual HID input report and it is to be formatted according to the guidelines below for the mouse and keyboard cases when their example HID Descriptors from the SDK are used.

HID Input Report for keyboard is made of eight bytes:

- first byte is the modifier key

- second byte must always be 0

- third to eighth bytes are the codes of keyboard keys.

Examples:

- an input report with value of 0000230000000000 would indicate that the key corresponding to number 6 of the US keyboard is pressed

- an input report with value of 0200230000000000 would indicate that the key corresponding to number 6 of the US keyboard is pressed while SHIFT button is also pressed (in the US keyboard this would result in the character ^ to be entered, while in the Finnish keyboard the entered character would be &)

- an input report with value of 000000000000 is needed to release all keys

- multiple key presses can be indicated at the same time, for example with a report like 0000232400000000 indicating that both US keyboard's keys 6 and 7 are being pressed.

HID Input Report for mouse is made of three bytes:

- first byte is a bitmask for button presses, where for example main button (normally left button) is pressed by sending the byte with the first bit set; for releasing the button another report should be sent with the byte having the first bit unset

- Second byte is for pointer movement along the X axis

- third byte is for pointer movement along the Y axis.

Examples:

- an input report with value of 003212 would mean mouse pointer moving 0x32=50 pixels to the right and 0x12=18 pixels down

- an input report with value of 0100FB would mean mouse pointer moving 0xFB=-5 pixels up and keeping the main button pressed

- an input report with value of 00000 after the above would be meant to release the main button

Note about the *Bluetooth* HID keyboard:

You can find country codes in the document named "Device Class Definition HID" at page 23, available at USB Implementers Forum website (http://www.usb.org/developers/hidpage/). This HID record's country code is a localization information only (required by the *Bluetooth* specification) and does not have any influence on the HID reports that are sent. In addition, the stack itself does not implement keyboard layout re-mapping across different languages, so for your reports you will have to select the key codes (from the document named "HID Usage Tables" at pages 53-59, also available at USB Implementers Forum website) so to match the keyboard layouts which are documented e.g. at http://www-01.ibm.com/software/globalization/topics/keyboards/registry_index.html

4.5 Disconnections

An ongoing HID connection can be closed by either side of the *Bluetooth* link. For the module to close the connection the following command is issued:

- ***dumo_cmd_endpoint_close(hid_endpoint_id)***

In addition to the response to the above command, simply the event ***dumo_evt_bt_connection_closed(reason_code, bt_connection_id)*** will be sent by the model to its host after a short delay.

If it is the HID-Host to close the connection instead, not just one but multiple events will be sent, as follows:

- ***dumo_evt_endpoint_closing(reason_code, hid_endpoint_id*)**
- ***dumo_evt_endpoint_status(hid_endpoint_id, hid_type, destination_endpoint_id, flags*)**
- ***dumo_evt_bt_connection_closed(reason_code, bt_connection_id*)**

Because of the event *endpoint_closing* and especially because of the event *endpoint_status* with the flags indicating that the HID endpoint is awaiting closure, you will have to explicitly close the endpoint for releasing the resources tied to the endpoint. This must be done by issuing the following command:

- ***dumo_cmd_endpoint_close(hid_endpoint_id)***

Closing the connection can happen also via the Virtual Cable Unplug procedure, in which case the request to delete any bonding information between the two connected devices is sent concurrently. The stack in the BT121 is capable of recognizing the incoming (and also its own) request and automatically deletes the bonding information accordingly. For the module to carry on the procedure the following command is used:

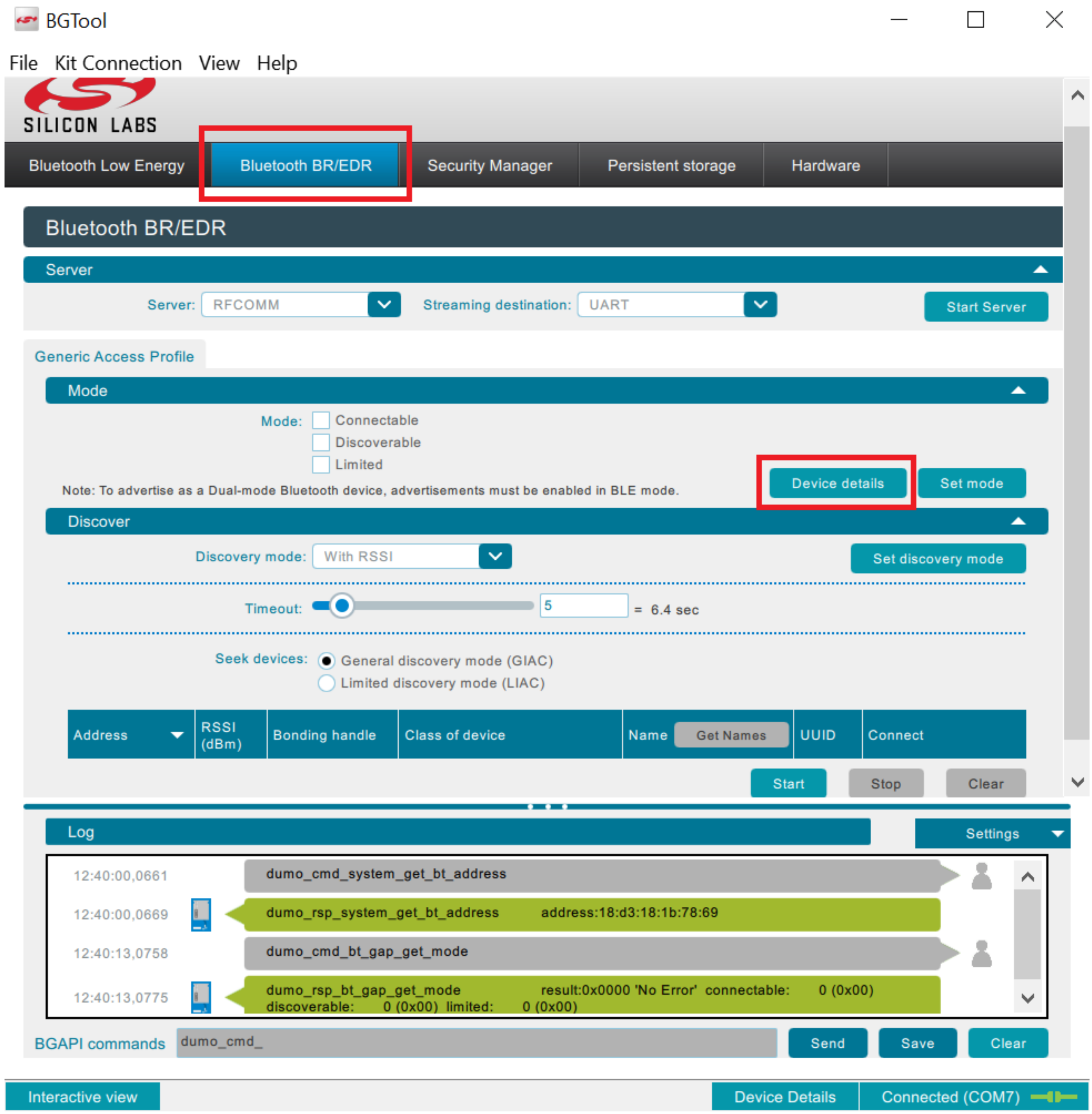
- ***dumo_cmd_bt_hid_virtual_cable_unplug(hid_endpoint_id)***

In addition to the response to the above command, the same three events as in the case above are received, and again the *dumo_cmd_endpoint_close(hid_endpoint_id)* will have to be issued. When it is the remote HID-Host to start the procedure the event ***dumo_evt_bt_hid_state_changed(hid_endpoint_id, state)*** might additionally be sent by the module to its host depending on the *Bluetooth* HID implementation at the remote side.

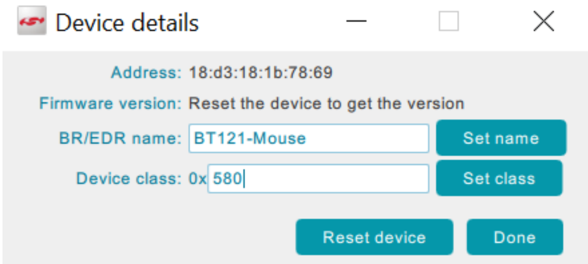
4.6 Testing with the BGTool

BGTool provides an effective way to test the HID implementation in the stack of the BT121. The screenshots in the following pages will show how a test can be carried on when the ready NCP example firmware image from the project directory \example\bt121_ br_edr_hid\ of the SDK is programmed to the module.

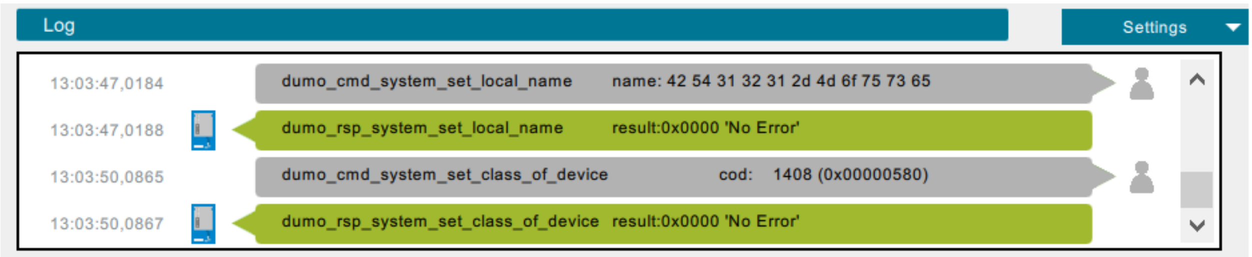
After powering on the module, then connecting the BGTool to the module's COM port, and finally selecting the "BR/EDR" tab, the program interface will look like in the screenshot below:



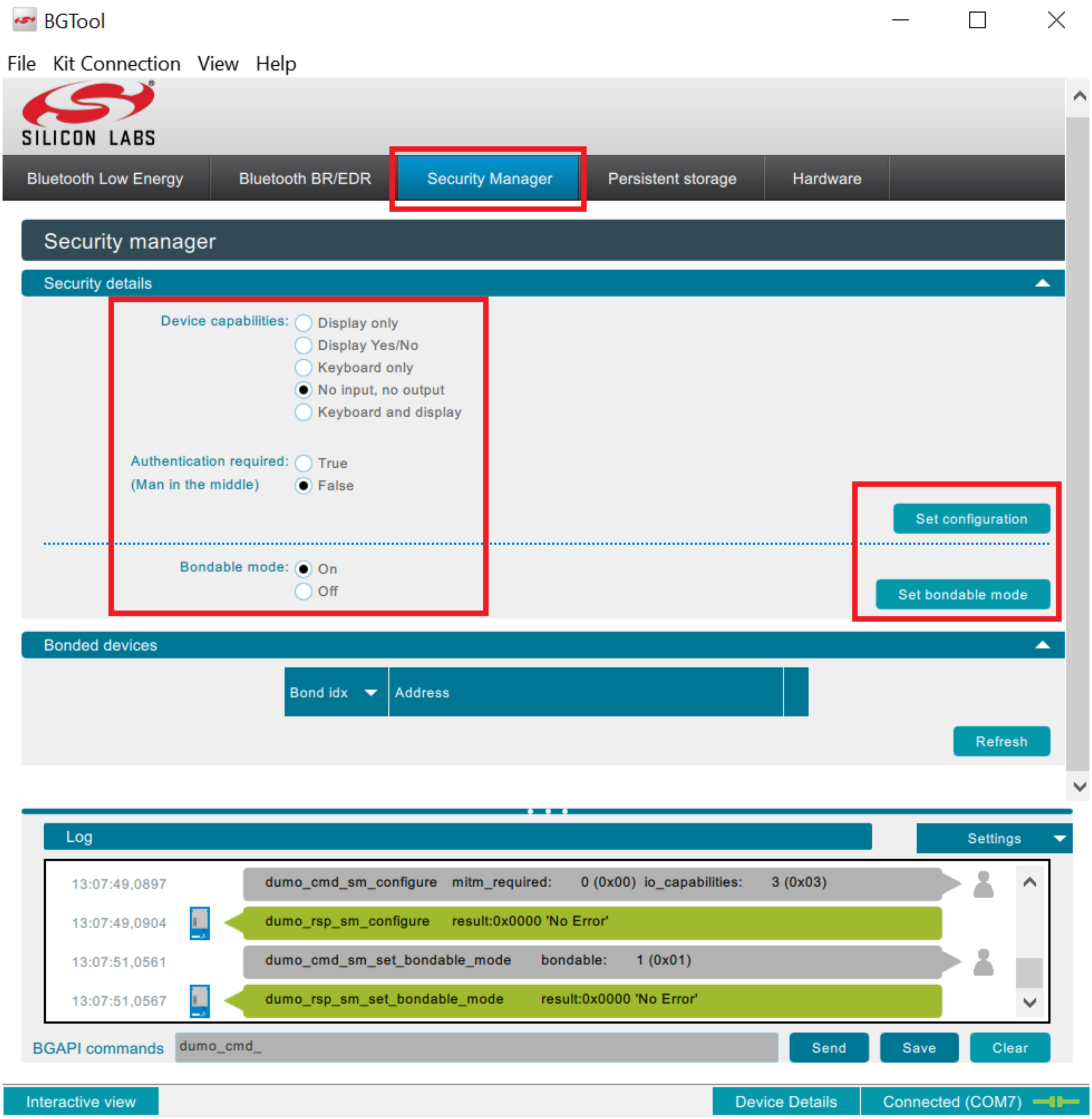
You will then click the “Device details” button to verify that the appropriate firmware build is running in the module, and to change the CoD (and the *Bluetooth* BR/EDR Friendly Name if desired):

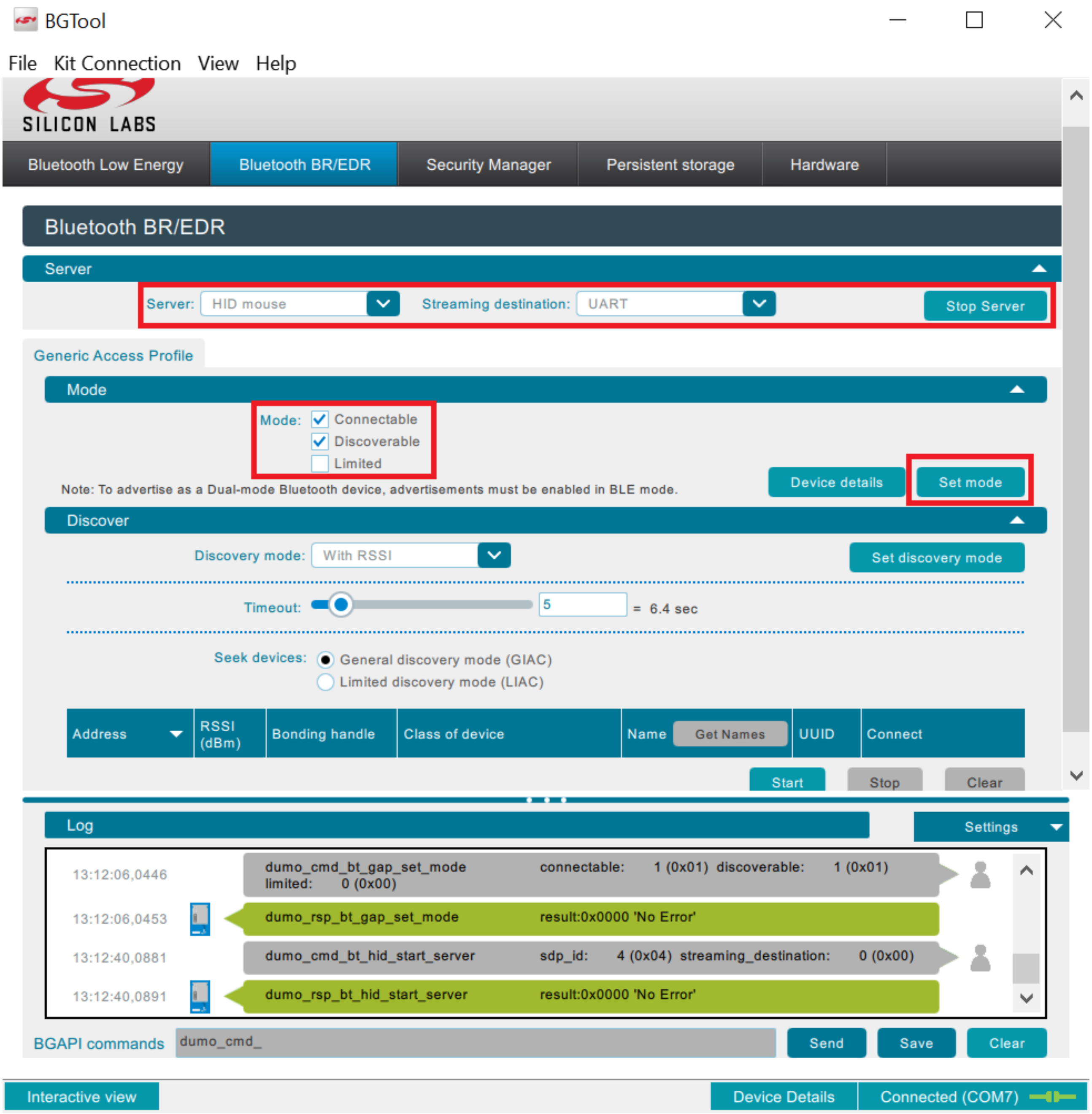


The commands actually sent by the BGTool to module, and the responses from the module are seen in the log window, as depicted below:

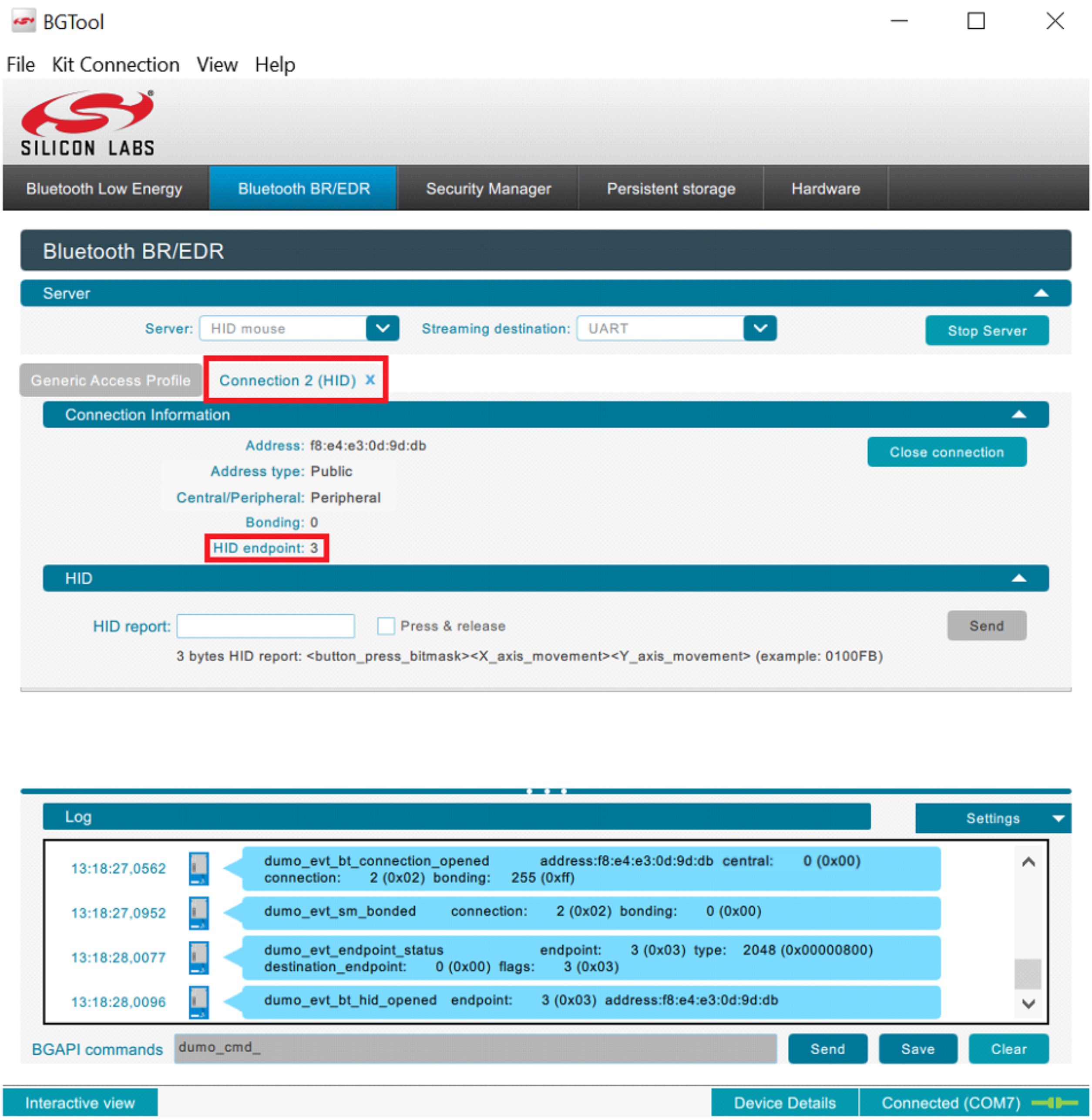


Next you will configure security, and finally you will make the module visible and connectable while starting the HID service, as seen in the next two screenshots:

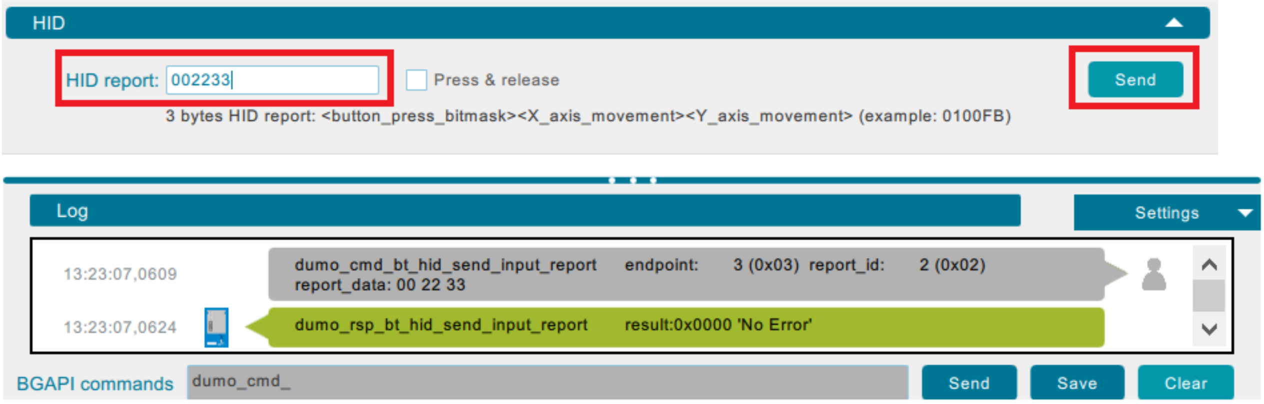



BGTool
File Kit Connection View Help
SILICON LABS
Bluetooth Low Energy
Bluetooth BR/EDR
Security Manager
Persistent storage
Hardware
Bluetooth BR/EDR
Server
Server: HID mouse
Streaming destination: UART
Stop Server
Generic Access Profile
Mode
Mode: Connectable
Discoverable
Limited
Note: To advertise as a Dual-mode Bluetooth device, advertisements must be enabled in BLE mode.
Device details
Set mode
Discover
Discovery mode: With RSSI
Set discovery mode
Timeout: 5 = 6.4 sec
Seek devices: General discovery mode (GIAC)
Limited discovery mode (LIAC)
Address
RSSI (dBm)
Bonding handle
Class of device
Name
Get Names
UUID
Connect
Start
Stop
Clear
Log
Settings
13:12:06,0446 dumo_cmd_bt_gap_set_mode connectable: 1 (0x01) discoverable: 1 (0x01) limited: 0 (0x00)
13:12:06,0453 dumo_rsp_bt_gap_set_mode result:0x0000 'No Error'
13:12:40,0881 dumo_cmd_bt_hid_start_server sdp_id: 4 (0x04) streaming_destination: 0 (0x00)
13:12:40,0891 dumo_rsp_bt_hid_start_server result:0x0000 'No Error'
BGAPI commands
dumo_cmd_
Send
Save
Clear
Interactive view
Device Details
Connected (COM7)

At this point the module is ready to receive the connection from the HID-Host:



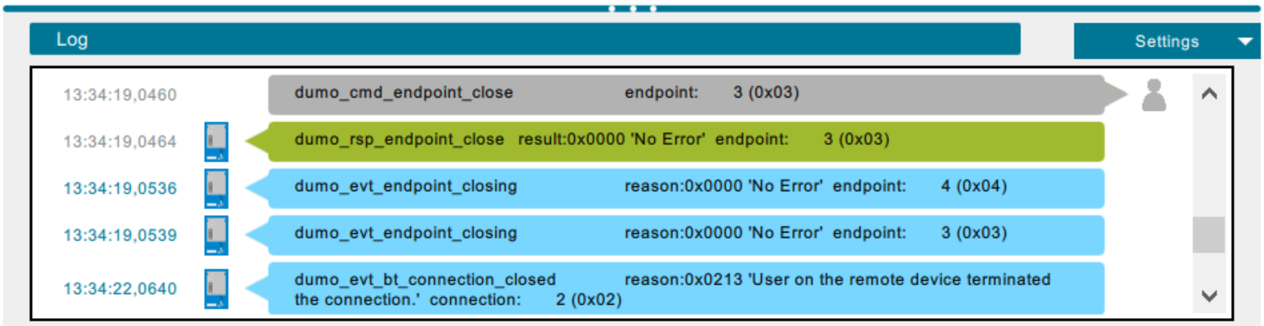
Once connected, the new tab dedicated to the HID connection will pop-up, providing all the information about the connection just created, and allowing you to easily send HID Input Reports to the remote HID-Host, as seen in screenshot on the next page:



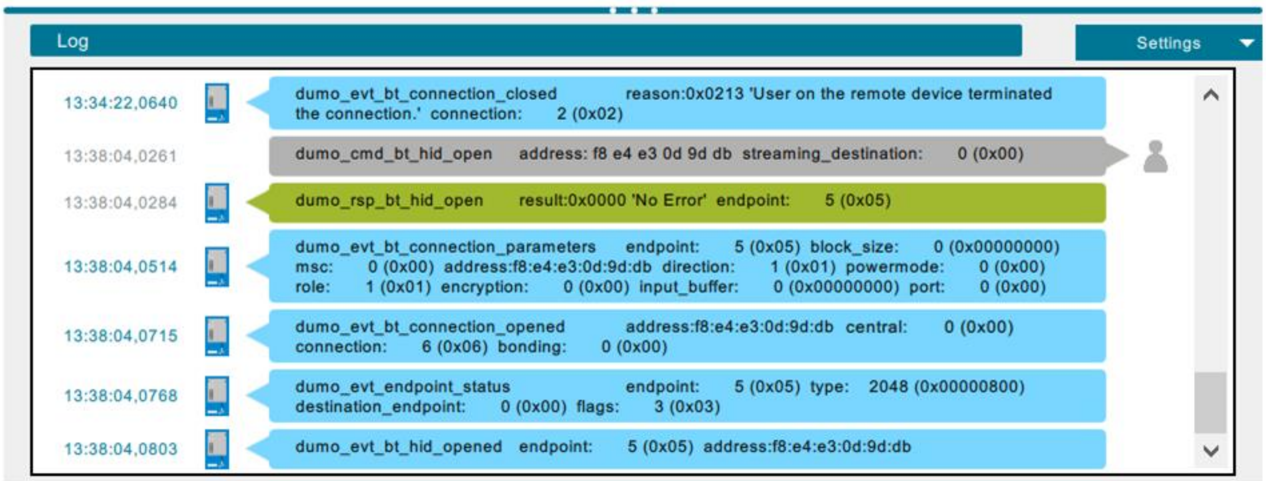
The HID Report above causes the mouse pointer at the PC or at the Android smartphone/tablet to move 0x22=34 pixels to the right and 0x33=51 pixels down.

You can also check the box “Press & release” for the BGTool to automatically send a report with all zeroes after each report being sent, which is handy to automatically release the mouse button presses, or the keyboard keys presses.

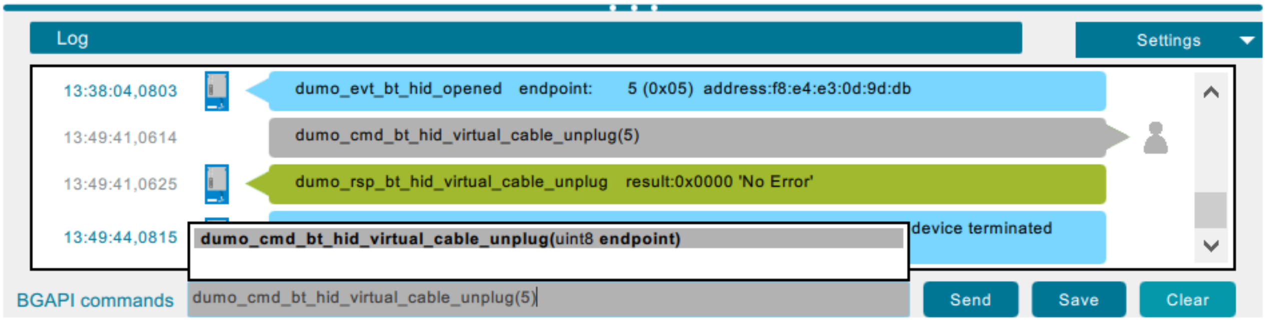
The HID tab in BGTool also offer the option to close the HID connection, in which case the resulting events discussed earlier are also seen in the log window as below:



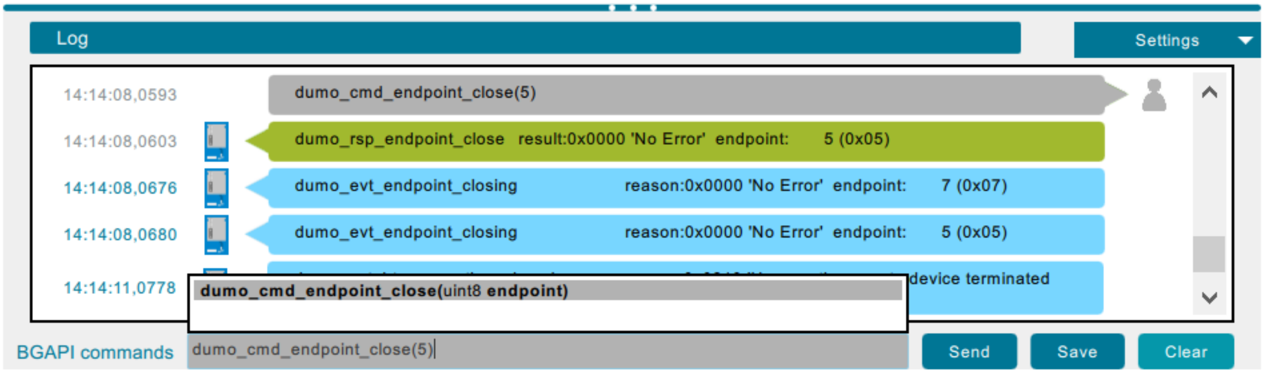
The button used to close the connection will turn into a button to re-open the connection, and when clicked the command *bt_hid_open* will be launched as seen in the log below:



If you wish instead to try the Virtual Cable Unplug feature, you will have to explicitly enter the appropriate command in the bar at the bottom of the BGTool, as depicted in the last screenshot below:



Because of the event *endpoint_closing* and especially because of the event *endpoint_status* with the flags indicating that the HID endpoint is awaiting closure, you will have to explicitly close the endpoint for releasing the resources tied to the endpoind. This is done by issuing the *endpoint_close* command at the command bar, similarly to the previous case with the *bt_hid_virtual_cable_unplug*, as seen below:

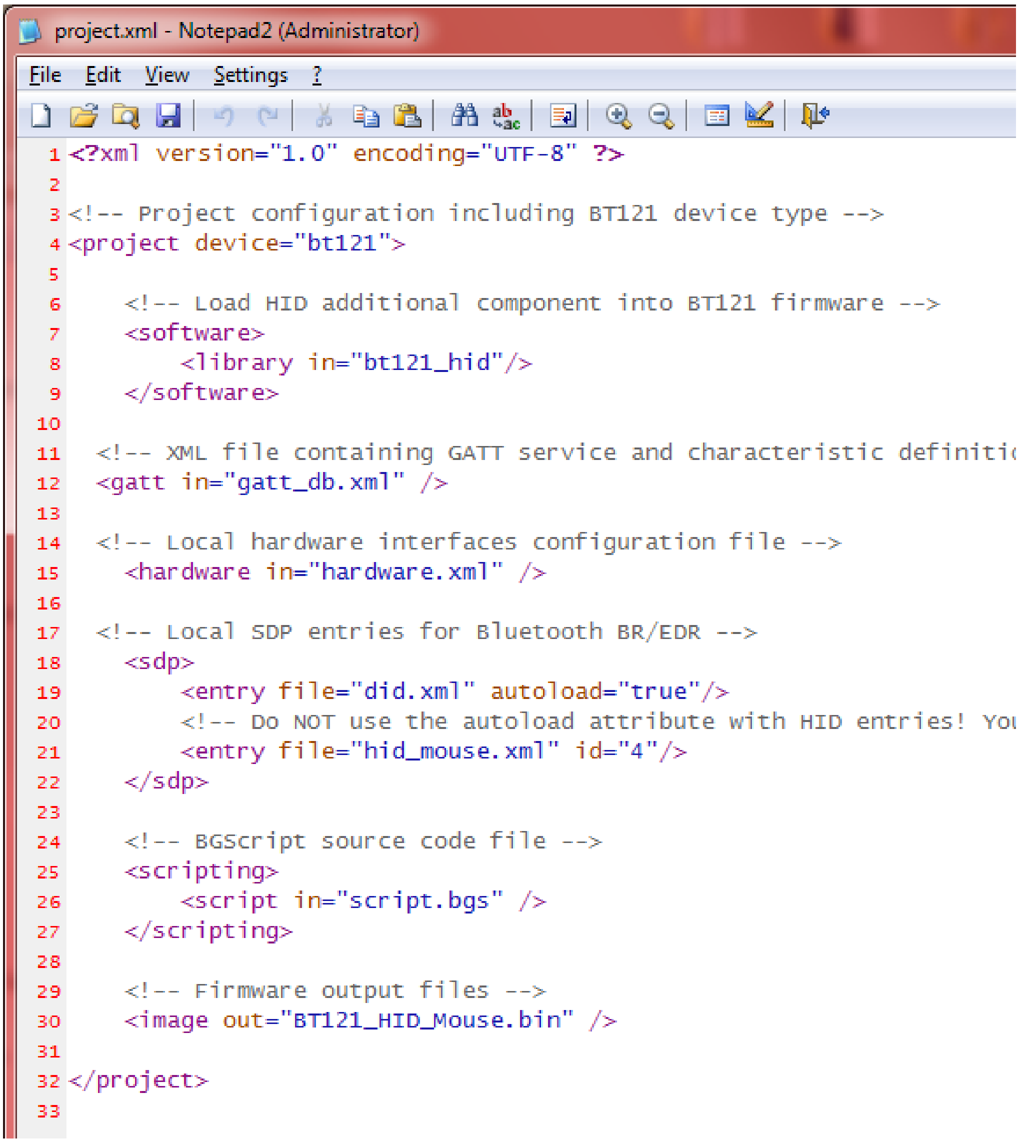


5 Walkthrough of the HID script-based example application

While the previous section had a strong focus on how to implement HID functionality with a module in hosted mode (also known as NCP = Network Co-processor mode), the aim of this section is to walk the user through the demo HID application which is enabled by the firmware found in the SDK under the directory *\example\br_edr_hid_mouse_demo* where such firmware is obtained from the project files discussed in this section and found under the same directory. This firmware allows the module to operate in standalone mode thanks to the provided example BGScript.

5.1 Project configuration file

Building a *Bluetooth* Dual Mode project starts always by making a project file, which is a simple XML file defining the resources used in the project. The example project file under discussion in this section has its content as shown below in Figure 2:



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

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

    <!-- Load HID additional component into BT121 firmware -->
    <software>
        <library in="bt121_hid"/>
    </software>

  <!-- XML file containing GATT service and characteristic definiti
  <gatt in="gatt_db.xml" />

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

  <!-- Local SDP entries for Bluetooth BR/EDR -->
    <sdp>
        <entry file="did.xml" autoload="true"/>
        <!-- Do NOT use the autoload attribute with HID entries! You
        <entry file="hid_mouse.xml" id="4"/>
    </sdp>

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

    <!-- Firmware output files -->
    <image out="BT121_HID_Mouse.bin" />

</project>

```

Figure 2: Project file

An explanation of the project file content is shown in the Table 4 on next page.

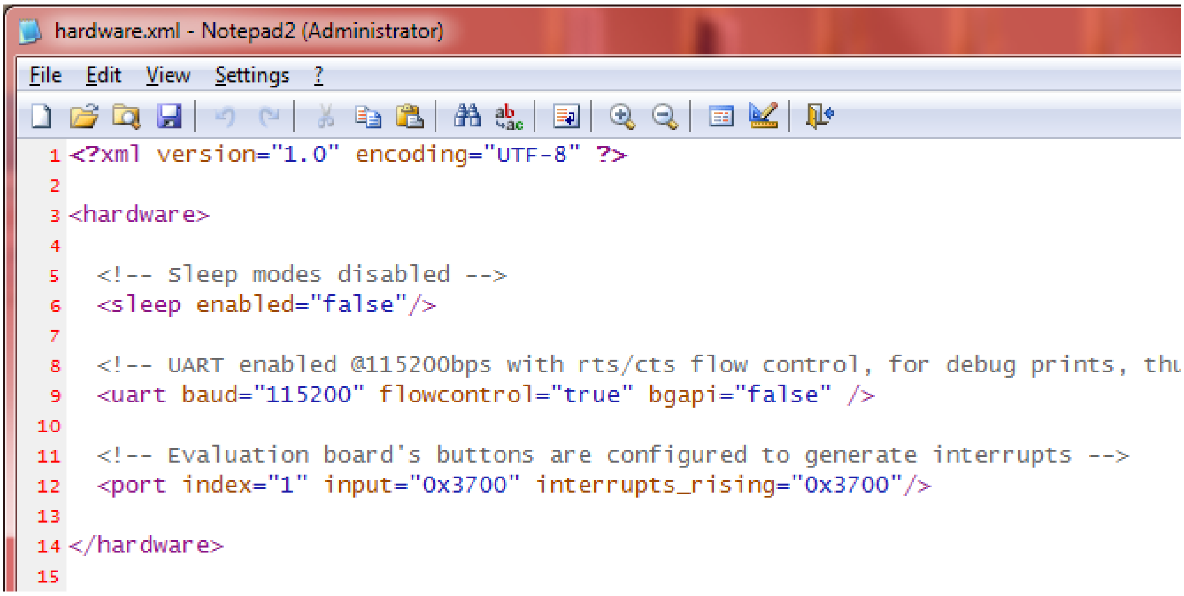
<project device="bt121">	This tag starts the project definition and the project file must end in </project> tag. The device is used to define for which *Bluetooth* module the project is used for.
<software> <library in="bt121_hid"/> </software>	The <software> tag is used to define what exact base firmware should the final firmware image file be based upon. The <library> tag allows the selection of the special base firmware containing the HID functionality.
<gatt in="gatt_db.xml"/>	The <gatt> tag is used to define the XML file containing the GATT service and characteristic database used for BLE and GATT over BR profile. **NOTE:** This example uses a minimal GATT database which is not in use given that the application does not include any BLE operation. For more information on how to use GATT database and its functions see Software Getting Started Guide.
<hardware in="hardware.xml" />	The <hardware> tag define which file contains the hardware configuration for interfaces like UART, SPI or I^2C.
<sdp> <entry file="DID.xml" autoload="true"/> <entry file="hid_mouse.xml" id="4"/> </sdp>	The <sdp> tag is used to define the *Bluetooth* BR/EDR Service Discover Protocol (SDP) records that we want to embed in the firmware image. The <entry> tags are used to point to the actual XML files for each SDP entry, and are also meant to provide a unique ID for each SDP entry which can later on be used in the BGScript code to load the desired SDP entry. In this example the HID SDP record is given an ID of 4. The xml file structure for the HID record is described in chapter 4.1
<scripting> <script in=" script.bgs" /> </scripting>	The BGScript file containing the code for the module's standalone functionality is defined inside the <scripting> tags, and further on inside the <script> tag.
<image out=" BT121_HID_Mouse.bin"	The <image> tag defines the desired name of the firmware image file generated by the bgbuild.exe free compiler. The generated .bin file contains the *Bluetooth* Dual Mode stack, the GATT database, the SDP entries, the hardware configuration and the BGScript code. The compiler will also generate a corresponding file with extension .bootdfu and this is meant to replace the bootloader should the BGAPI-based DFU upgrading method be used to re-flash a module running an older firmware build (refer to separate instructions)

Table 4: explanation of the project configuration file entry by entry

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 file

The next logical step is to define the hardware configuration of the *Bluetooth* module, and to define in particular which hardware interfaces are enabled and what are their default settings and/or configurations. The hardware configuration file content for the example under discussion is very simple and is shown below in figure 3:




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

<hardware>

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

  <!-- UART enabled @115200bps with rts/cts flow control, for debug prints, th
  <uart baud="115200" flowcontrol="true" bgapi="false" />

  <!-- Evaluation board's buttons are configured to generate interrupts -->
  <port index="1" input="0x3700" interrupts_rising="0x3700"/>

</hardware>
```

Figure 3: Hardware configuration file

An explanation of the hardware file content is shown in Table 5 below:

<sleep enabled="false" />	This tag defines that the MCU sleep modes (low power modes) are disabled.
<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 this example project the UART is used only for debug and/or status messages. BGAPI serial protocol is disabled as this example uses a BGScript application to implement the application logic.
<port index="1" input="0x3700" interrupts_rising="0x3700"/>	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 example, pins 8, 9, 10, 12 and 13 of port B are configured as inputs and for rising interrupts. PB8, PB9, PB10, PB12 and PB13 are connected to the 5 buttons of the main board of the evaluation kit and are used to simulate the mouse movements and the main button's presses.

Table 5: Hardware configuration file explained

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

5.3 *Bluetooth* services configuration

The example project is meant for a firmware containing only one SDP records exposing one single usable *Bluetooth* profile, that of the Bluetooth HID, and in particular a mouse device. This SDP record is defined in the file called *hid_mouse.xml* the content of which is described in detail in chapter 4.1

The *hid_mouse.xml* has itself an entry to point to another file. The entry is ***<HIDUSBDescriptor file="hid_mouse.txt"/>*** and the pointed file contains the actual HID descriptor, which was also discussed at the end of chapter 4.1

Among the project files there exists one additional entry called *did.xml* which is as well included into the firmware image thanks to the directive *<entry file="did.xml" autoload="true"/>* inside the project configuration file. This is meant for the informational yet mandatory Device Information Profile. An example of how this would look is shown in Figure 4 below:

```
<!-- 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 4: Record definition for the *Bluetooth* Device Information Profile

With the DI profile more configuration options exist. Notice that the order of the configurations as seen in the example .xml file must not be changed. The Device Information Profile tags are explained in more detail below in table 6:

<UINT16 value="0200"/> <UINT16 value="0103"/>	**<u>This MUST not be changed.</u>**
<UINT16 value="0201"/> <UINT16 value="0047"/>	0201 refers to vendor ID parameter and **<u>you must change the 0047 to your own vendor ID</u>** received either from USB Implementers Forum or Bluetooth SIG.
<UINT16 value="0202"/> <UINT16 value="1234"/>	0202 refers to product ID parameter and you can replace the value 1234 with your own ID.
<UINT16 value="0203"/> <UINT16 value="0000"/>	0203 refers to product version and you can replace the value 0000 with your own version number.
<UINT16 value="0204"/> <UINT16 value="1"/>	**<u>This MUST not be changed.</u>**
<UINT16 value="0205"/> <UINT16 value="0001"/>	0205 refers to the source of the vendor ID and it must tell if your own vendor ID is from Bluetooth SIG or USB Implementers Forum 0000: Source of vendor ID is USB Implementers Forum 0001: Source of vendor ID is Bluetooth SIG

Table 6: Record definition of the *Bluetooth* Device Information Profile explained

5.4 BGScript™ code

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

The *system_boot* event is generated when power is applied to the *Bluetooth* module and this is the starting point for the code execution. Immediately after, the event *system_initialized* is also generated, denoting that the Bluetooth chipset in the BT121 has become ready to receive commands.

The Figure 5 in the next page shows the initial part of the script which includes the variables definition and all the commands that are launched at the time the first two single events are captured. Functionality based on the commands that are called within these two events is also described after the figure, and follows closely the comments in the code itself.

Notice that the script is simply a text file that can be edited for example with Notepad++ for which a syntax highlighter can be downloaded from the Bluegiga web site.

```
#variable definitions
const uart_endpoint = 0
dim ascii_string_len
dim ascii_string_data(3)
dim hid_endpoint
dim hid_connected
dim hid_host_address(6)

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

    # Write welcome messages out of the UART interface at boot
    call endpoint_send(uart_endpoint, , "\r\n\r\n======System Started======\r\n")
    call endpoint_send(uart_endpoint, , "\r\n--> Firmware Build: ")
    call util_itoa(build)(ascii_string_len, ascii_string_data(0:ascii_string_len))
    call endpoint_send(uart_endpoint, ascii_string_len, ascii_string_data(0:ascii_string_len))
    call endpoint_send(uart_endpoint, ,"\r\n--> This is the BT121 HID-Mouse demo\r\n\r\n")

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

    # Let's initialize a few variables
    hid_endpoint = -1
    hid_connected = 0

end

# 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(15,"BT121-HID-Mouse")

    # Load HID-Mouse SDP record having ID of 4 according to the project.xml
    call bt_hid_start_server(4, 0)

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

    # 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)

    # Configure Class-of-Device so that the remote side can realize in its inquiry the kind of dev
    call system_set_class_of_device($580)

end
```

Figure 5: Start of the BGScript example

According to the commands launched by the script when the *system_boot* and the *system_initialized* events are captured, the module will first send out of its UART interface a welcome message that includes the firmware build; then the script will initialize a few variables while arranging for any data sent to module over UART to be discarded; finally, the script completes the initialization of the module by configuring the following parts:

- Set the Bluetooth BR/EDR's Friendly Name to "BT121-HID-Mouse": this will be the name seen for example at the smartphone/tablet or at the PC when searching for devices in range

- Set the Class-of-Device to 0x580: this was discussed in detail in chapter 4.2 and will allow the smartphone or the tablet or the PC to realize the kind of device was found and to present an appropriate icon next to it

- Set the Secure Simple Pairing (SSP) capabilities of the module to no-input no-output, and disable MITM protection, so that the “Just-Works” pairing mechanism will be in use: with this configuration the module and the smartphone or PC will pair without the user having to enter a passkey

- Set the module to store the pairing information for future use by making it bondable: when devices are bonded, no SSP procedures have to happen again for the exchange of link keys. SSP-based pairing and bonding is mandatory between Bluetooth BR/EDR devices since Core Specification 2.1

- Start the HID server, so that incoming HID connections can be recognized and accepted, and so that the HID SDP record (that we defined to have ID of 4) is loaded from memory and made available for SDP queries by remote devices

- Make the module visible and connectable.

Another important part of the script concerns the connection and disconnection handlers: these are meant to trigger pre-defined actions in cases when a connection is established or a disconnection is detected. Figure 6 in the next page shows this part of the script, and a description of the functionality enabled by the launched commands is given after the figure.

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

script.bgs

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

    # When the endpoint having type of HID has become active, meaning that HID connection is established
    if (type = endpoint_hid) && (flags & ENDPOINT_FLAG_ACTIVE) then

        # Store endpoint ID and configure the intended variable to indicate that connection is up
        hid_endpoint = endpoint
        hid_connected = 1

        # Write a message to UART interface to indicate connection
        call endpoint_send(uart_endpoint, ,"HID connection established\r\n")

    end if

end

# This event is generated when an endpoint is closing, for example when the HID connection is closed by the remote side
event endpoint_closing(reason,endpoint)

    # Explicitly close the endpoint to free memory (this could be skipped here as it is done automatically by the firmw
    call endpoint_close(endpoint)

    # Write a message to UART
    call endpoint_send(uart_endpoint, ,"HID connection closed\r\n")

    # Release endpoint ID variable and configure the intended variable to indicate that no connection is ongoing
    hid_endpoint = -1
    hid_connected = 0

end

# This event indicates the HID Host has notified the Device about a state change
event bt_hid_state_changed(endpoint, state)

    # Virtual cable unplugging happens for example when HID Host forgets/unpair the BT121
    # BT121 will also automatically remove the bonding entry
    if endpoint = hid_endpoint & state = bt_hid_state_virtual_cable_unplug then

        # Write a message to UART
        call endpoint_send(uart_endpoint, ,"HID connection closed and devices unpaired\r\n")

        # Release endpoint ID variable and configure the intended variable to indicate that no connection is ongoing
        hid_endpoint = -1
        hid_connected = 0

    end if

end
```

Figure 6: connection and disconnection event handlers

The first event handler shown above is used to launch a few commands upon detecting an established HID connection. Commands will:

- Send a status message out of the UART telling that an HID connection is established

- Store the endpoint ID to a global variable that can be used within other events

- Set the *hid_connected* variable so to define the exact function associated to the two GPIOs with double functionality for the case when the connection exists.

In order to catch the basic disconnection event the second event listener is needed, which will launch other few commands:

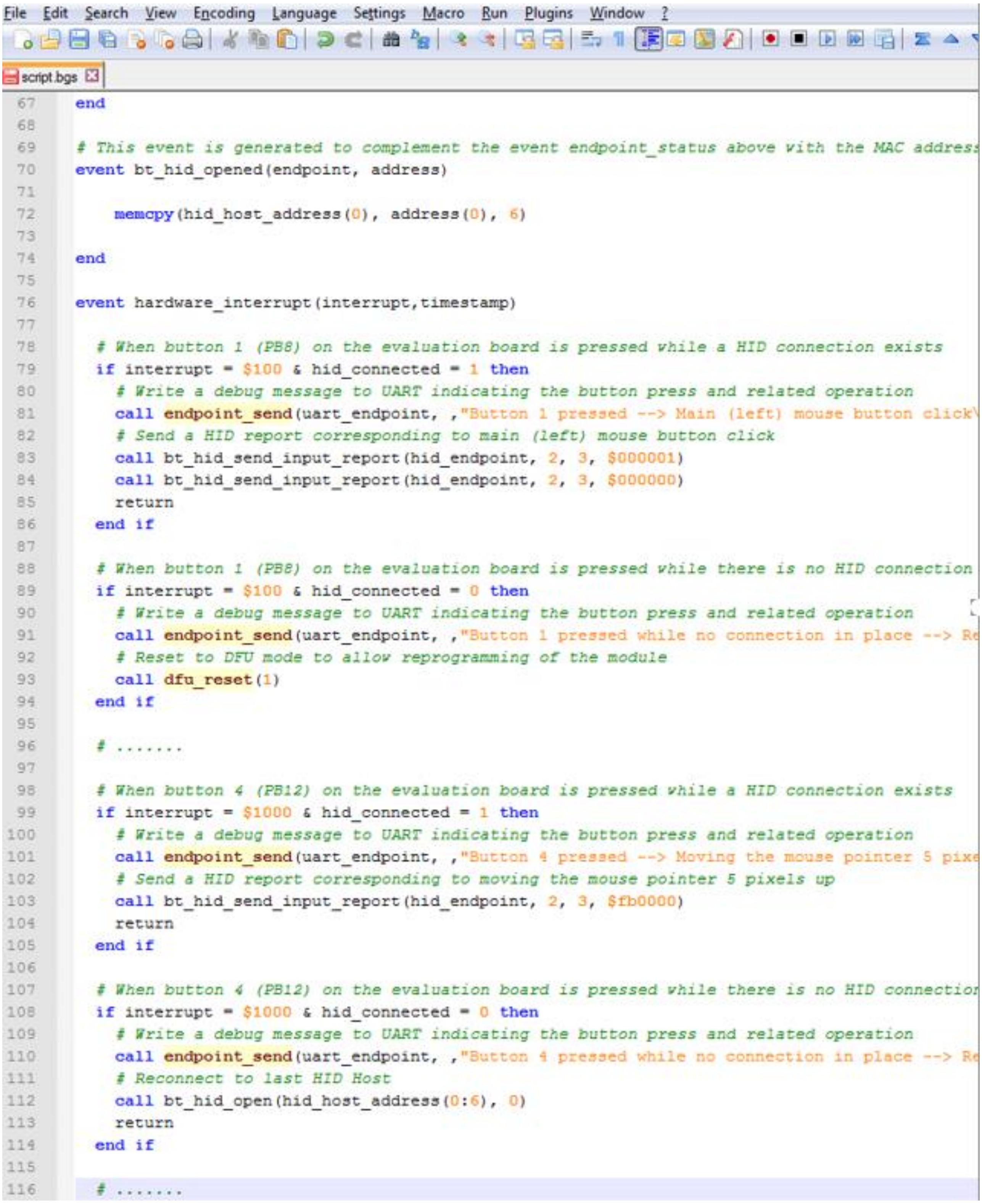
- Send a status message out of the UART telling that the HID connection closed

- Explicitly close the endpoint to free resources tied to the endpoint

- Unset the *hid_connected* variable and resets to initial state the variable used to temporarily hold the current HID endpoint when one is assigned.

The last event listener in figure 6 is meant to recognize if the remote side is sending the virtual cable unplug instruction, in which case a message is sent out of the UART to notify about this, while the two variables are rewritten just as above. Remember that the firmware will automatically remove the bonding entry related to the remote HID-Host that sent the instruction.

The last and fundamental part of the script is where the actual mouse functionality is defined. Figure 7 in the next page shows an extract of this part and an explanation will follow.



```
end

# This event is generated to complement the event endpoint_status above with the MAC address
event bt_hid_opened(endpoint, address)

    memcpy(hid_host_address(0), address(0), 6)

end

event hardware_interrupt(interrupt,timestamp)

  # When button 1 (PB8) on the evaluation board is pressed while a HID connection exists
  if interrupt = $100 & hid_connected = 1 then
    # Write a debug message to UART indicating the button press and related operation
    call endpoint_send(uart_endpoint, ,"Button 1 pressed --> Main (left) mouse button click\
    # Send a HID report corresponding to main (left) mouse button click
    call bt_hid_send_input_report(hid_endpoint, 2, 3, $000001)
    call bt_hid_send_input_report(hid_endpoint, 2, 3, $000000)
    return
  end if

  # When button 1 (PB8) on the evaluation board is pressed while there is no HID connection
  if interrupt = $100 & hid_connected = 0 then
    # Write a debug message to UART indicating the button press and related operation
    call endpoint_send(uart_endpoint, ,"Button 1 pressed while no connection in place --> Re
    # Reset to DFU mode to allow reprogramming of the module
    call dfu_reset(1)
  end if

  # .......

  # When button 4 (PB12) on the evaluation board is pressed while a HID connection exists
  if interrupt = $1000 & hid_connected = 1 then
    # Write a debug message to UART indicating the button press and related operation
    call endpoint_send(uart_endpoint, ,"Button 4 pressed --> Moving the mouse pointer 5 pixe
    # Send a HID report corresponding to moving the mouse pointer 5 pixels up
    call bt_hid_send_input_report(hid_endpoint, 2, 3, $fb0000)
    return
  end if

  # When button 4 (PB12) on the evaluation board is pressed while there is no HID connectio
  if interrupt = $1000 & hid_connected = 0 then
    # Write a debug message to UART indicating the button press and related operation
    call endpoint_send(uart_endpoint, ,"Button 4 pressed while no connection in place --> Re
    # Reconnect to last HID Host
    call bt_hid_open(hid_host_address(0:6), 0)
    return
  end if

  # .......
```

Figure 7: interrupt handler for mouse pointer movements, main button press, reconnection and DFU reset

The event *hardware_interrupt* is generated whenever a button of the evaluation main board is pressed, given the configuration of the pins PB8, PB9, PB10, PB12 and PB13 as interrupts in the hardware configuration file (as seen in chapter 5.2), and given that these module's pins are mapped respectively to the buttons 1 to 5 in the main board of the evaluation kit.

When this event is generated, "if" statements are used to recognize the actual button being pressed and if a connection is in place or not. Whenever the conditions are met, a message is sent out of the UART to indicate the button press and the action that the button press will cause.

In particular, when the HID connection exists a button press will trigger the movement of the mouse pointer in the direction assigned to the button, namely 5 pixels upwards for button 4, as in figure 7 above, 5 pixels downwards for button 2, 5 pixels to the right for button 3, and 5 pixels to the left for button 5. The mouse pointer movements are launched with the call *bt_hid_send_input_report* in accordance to the description of this command given in chapter 4.4

In addition to the 4 buttons, 2 to 4, used for the pointer movements, button 1 is also assigned a double function: while a HID connection exists, pressing this button will translate into a main button mouse click where the click happens via the two consecutive hid reports seen in the first "if" group in figure 7 (the second of the reports is needed for the button release). If no connection is in place, the press of button 1 will reset the module into DFU mode to allow re-flashing of the module using the BGAPI-based DFU over UART.

There is actually a second button which is assigned double functionality: this is button 4 which will command the module to attempt to re-connect to the last connected HID-Host when there is no ongoing connection already. Re-connection is called via the command *bt_hid_open* and since this command requires the MAC address of the HID-Host to connect to, the event *bt_hid_opened* is also used by the script so to capture the MAC address into a global variable upon connection establishment.

The last two figures in this chapter depict the UART messages from the module seen at the terminal, and how the HID-enabled module is seen by an Android phone after bonding. (Unfortunately the Nexus 5X under test removes the mouse pointer just before taking a screenshot, that is why a photo is used instead.)

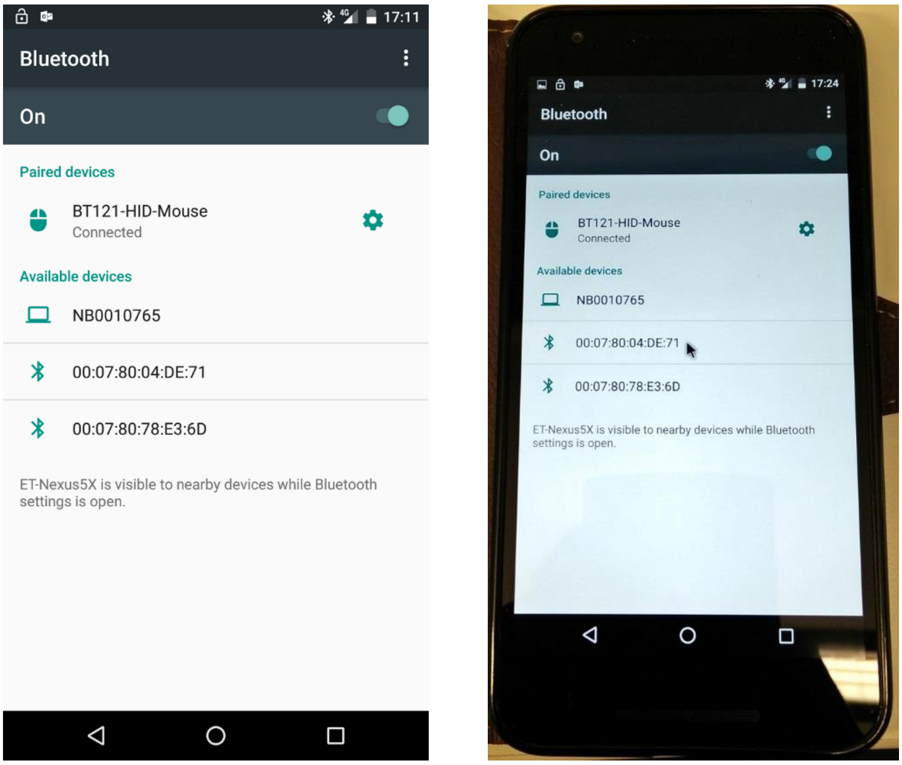


Figure 8: Bluetooth HID connection and mouse pointer in Android

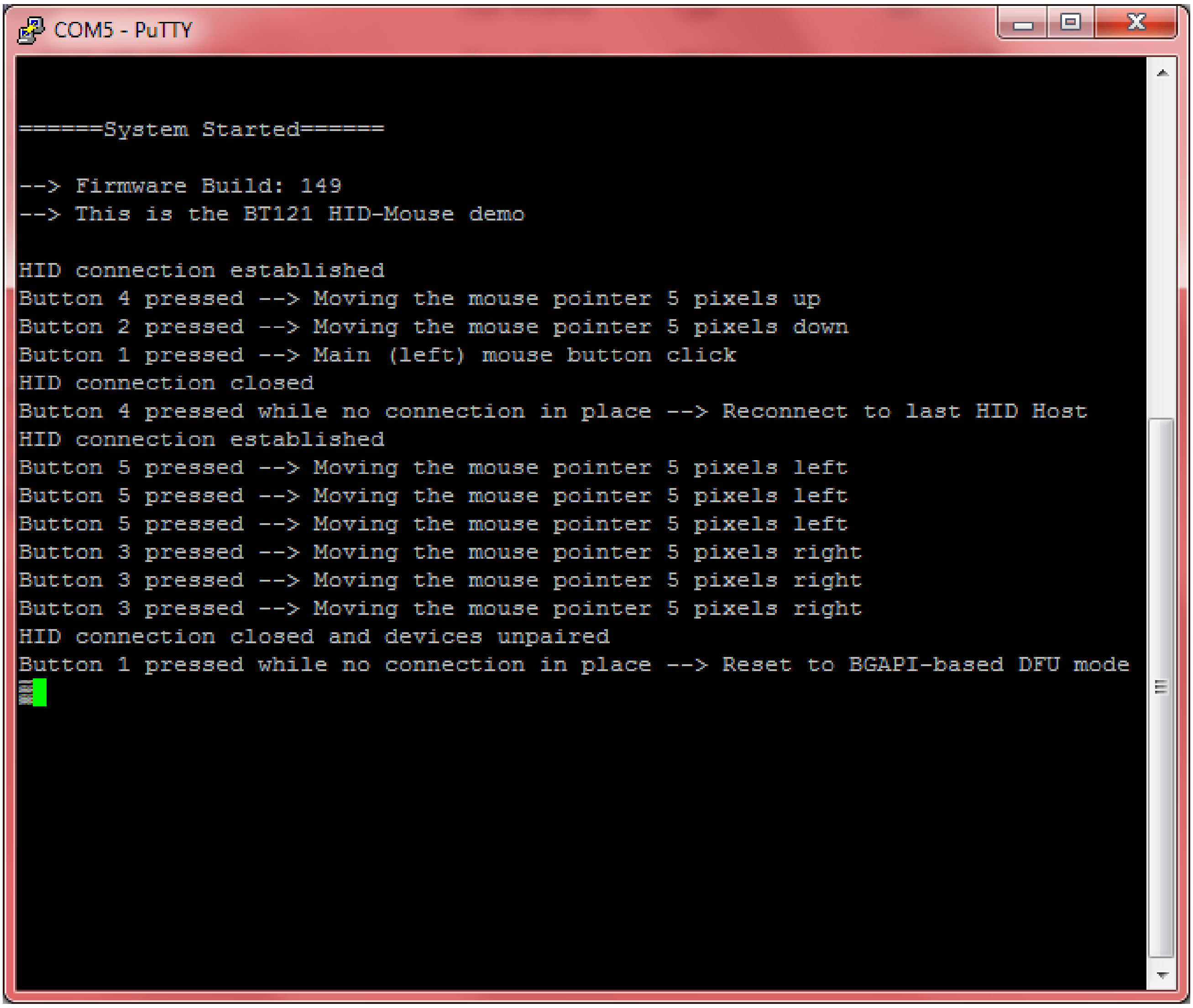


Figure 9: Debug and status messages out of module's UART

6 Bluetooth Qualification

The HID implementation in the new firmware of the BT121 has been tested against the Bluetooth PTS software by the SIG and verified to be interoperable and fully working according to the specification.

The updated qualification ID for the new firmware is found here: https://www.bluetooth.org/tpg/listings.cfm

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 LABS

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

www.silabs.com