CREATING CUSTOM BLE PROFILES

HANDS-ON LAB

Monday, 10 June 2013

Version 1.0

TABLE OF CONTENTS

1 Introduction

The purpose of this lab is to give you an understanding of what goes into a Bluetooth Low Energy profile, and how to use the Bluegiga SDK and Profile Toolkit to create a completely custom profile for your application. The Profile Toolkit and related documentation will allow you to create a profile that fits your data and transmission requirements perfectly.

Unlike classic Bluetooth profiles such as the Serial Port Profile (SPP) and Hands-Free Profile (HFP), many Bluetooth Low Energy projects use entirely customized profiles. There are many pre-defined BLE profiles and services that have been specified and adopted by the Bluetooth SIG, but there is no requirement to use these, and often it makes more sense to develop your own. You may also use a modified version of one of an official profiles. Since BLE data structures and transmission rules are all defined at the application level, this means you have complete control over your profile.

For this lab, we will create a simple “Hello IO” service which will allow the following functionality:

- Receive a name string from a remote device and store it locally
- Send a “Hello, [name]” string to a remote device via indications (pushed data w/acknowledgements)
- Send a 3-byte status report of all I/O pins (P0_0-7, P1_0-7, and P2_0-2) on demand

This will demonstrate simple two-way communication that is common to many applications: sending control data or commands from the master/client device, and receiving confirmations or status updates from the slave/peripheral device.

2 Understanding the Profile Toolkit

The development process for BLE profiles requires building multiple parts to work together, and to help with this process Bluegiga has created a set of reference materials and guides for each part of the process, in addition to the build tools themselves. The materials available are these:

- **BLE SDK archive** – Contains the BGBuild compiler and BLEGUI test application for building projects
- **BLE Update application** – Windows GUI tool for one-click compile + flash operations
- **Profile Toolkit Developer Guide** – Describes the BLE project file structure and XML syntax for all project definition files:
    - **project.xml** – Main project structure definition
    - **hardware.xml** – BLE hardware definition
    - **gatt.xml** – GATT database definition
    - **config.xml** – Application customizations for some extended behavior options
- **BGScript Developer Guide** – Describes the syntax and provides some examples of the BGScript language used for developing standalone on-module applications which do not require an external host processor for application logic
- **API Reference Guide** – Provides a comprehensive reference for all available API commands, responses, and events, including all parameters and return values

In addition to these core resources, we also provide a number of example projects to help demonstrate functionality. Some examples can be found in the **/example** folder of the BLE SDK, and others on the Example Projects subforum of the Bluetooth Smart forum on our support website (login required).

For the purposes of this lab, you should already have the BLE SDK archive downloaded and extracted, and the BLE Update application installed, as well as the CC debugger properly connected with any required drivers. The “**Getting Started with BLE Development Kit**” lab covers these topics if you need guidance.

3 Creating the Hello IO Project

In this section, you will create the entire BLE project from start to finish. This project will run as a standalone on-module BGScript application, requiring no external host for module control.

3.1 Main project definition (**project.xml**)

The our "Hello IO" project implementation is started by first creating a project file (**project.xml**), which defines the resources use by the project and the firmware output file. This project file includes a description of each of the main elements of the project. Using Notepad++ or another suitable text editor, create a file called "project.xml" and give it the following contents:

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

Figure 1: Project file for Hello IO (project.xml)

NOTE: You can also give this file a ".bgproj" extension, which will associate it with the BLE Update application if you have it installed. This will allow you to simply double-click on the project file from within Windows Explorer after you have finished, and it will automatically be opened in BLE Update, ready for flashing.

The main explanation of each of these tags is as follows:

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

Once you have added this content to **project.xml** (or **project.bgproj**) and saved the file, continue.

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

The **hardware.xml** file contains the hardware configuration for the BLE112 device. It describes which interfaces and functions are used and what their specific properties are. For our “Hello IO” device, we need only very basic functionality on a hardware level.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<hardware>
    <sleeposc enable="true" ppm="30" />
    <usb enable="false" />
    <txpower power="15" bias="5" />
    <script enable="true" />
    <pmux regulator_pin="7" />
    <port index="0" pull="down" />
    <port index="1" pull="down" />
    <port index="2" pull="down" />
</hardware>
```

Figure 2: Hardware configuration for Hello IO (hardware.xml)

- **<sleeposc>** The 32.768KHz sleep oscillator is enabled. Sleep oscillator allows the device to enter power mode 1 or 2 between *Bluetooth* operations, for example between connection intervals. This should **always** be used for BLE112 projects.

- **<usb>** USB interface is disabled to save power and allow the device to go to low-power modes. If USB is enabled, no low-power modes will be used. The USB port is enabled by default if this tag is omitted.

- **<txpower>** TX power is set to +3dBm value. Every step represents roughly a 2dBm change and the range of the parameter is 15 to 0, corresponding TX power values from +3dBm to -24dBm.

- **<script>** Scripting is enabled since the Hello IO example application is implemented with with the BGScript scripting language.

- **<pmux>** Enables automatic management of the DC/DC converter on the DKBLE112. This prevents momentarily large current draws from the CR2032 battery during transmissions, if battery power is used, and will extend the life of the battery.

- **<port>** Configures all Port 0/1/2 pins as inputs with internal pull-downs. This will put them into a known state for when we want to read the port status later.

  NOTE 1: the P1_0 and P1_1 pins do not have internal pull-downs because they are optionally configurable as high-current outputs (20ma). They will need to be pulled or driven externally for a reliable reading.

  NOTE 2: the P1_7 pin is configured to operate the DC/DC converter on the DKBLE112 board. This is operated by the stack automatically since the <pmux> tag is present, and will cause the pin to be driven high whenever the BLE radio is active. The status reported for this pin may fluctuate as a result.

Once you have added this content to **hardware.xml** and saved the file, continue.

3.3 GATT database definition (**gatt.xml**)

This section describes how to define our custom “Hello IO” profile’s services. This is where we define the data structure and which kinds of operations are allowed on each attribute in the database. It is important to understand the structure of a GATT database and what is meant by “profile,” “service,” and “characteristic” (also referred to as an “attribute”). Here is a quick summary:

- A **profile** is a collection of one or more **services**
- A **service** is a collection of one or more specifically defined **characteristics**
- A **characteristic** is a single value with specific access rules (read/write/notify/indicate/authentication)

Note that while a **profile** is a logical collection of services and does not itself have a unique identifier, each service and each characteristic requires a universally unique identifier (UUID). The Bluetooth SIG has designated different UUID classes for officially adopted services and characteristics (16 bits in length) vs. custom/proprietary services and characteristics (128 bits in length).

Our “Hello IO” profile will include two services, one of which is found in nearly every BLE peripheral’s GATT structure. The service/characteristic hierarchy looks like this:

a. **Generic Access Profile** service (UUID 0x1800)
    a. **Device Name** characteristic (UUID 0x2A00) – friendly name used of the device (read only)
    b. **Appearance** characteristic (UUID 0x2A01) – type of device (read only)
b. **Hello IO** service
    a. **User Name** characteristic – user’s name (read/write)
    b. **Greeting** characteristic – used to send a greeting message (indicate only)
    c. **Port Status** characteristic – used to poll current port status (read only)

The GAP service and characteristics have been adopted by the Bluetooth SIG, and so they have 16-bit UUIDs shown above. The Hello IO service and its characteristics are entirely custom, so we will need to generate our own 128-bit UUIDs for each of them. This is most easily done with a tool such as guidgenerator.com. We need four identifiers, one for the service and one for each of the three characteristics:

- `1006e5f7-9f72-4317-8a6c-b640b4a56659` – **Hello IO** service
- `e2df8530-49bc-4a9c-b7c2-79b6f9c1a9eb` – **User Name** characteristic
- `2fe67f97-2797-440c-9b8e-9d0ef2415df4` – **Greeting** characteristic
- `12b45989-8e38-42e4-afb6-f68cc2be254b` – **Port Status** characteristic

These will be used in just a moment. Represented visually, the profile looks like this:

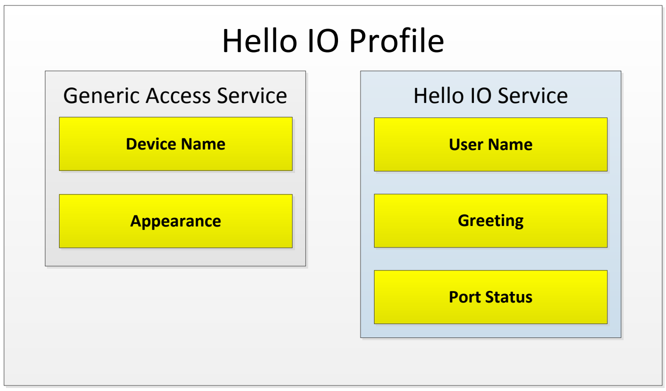


Figure 3: Visual GATT database structure

3.3.1 Generic Access Profile (GAP) service

Every *Bluetooth* low energy device needs to implement a GAP service. The GAP service here is very simple and consists of only two characteristics.

The service has two characteristics, which are explained in Table 1. In this example the characteristics are read-only, so they are also marked as **const.** Constant values are stored on the flash of BLE112 and the value is defined in the GATT database. Constant values cannot be changed. Feel free to give your device a unique name, but keep it relatively short since there is a limited amount of space available in advertising packets (typically 20 bytes is a good rule of thumb for the device name).

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

    <!-- 2A00: org.bluetooth.characteristic.gap.device_name -->
    <characteristic uuid="2A00" id="c_device_name">
        <description>Device Name</description>
        <properties read="true" const="true" />
        <value>Hello IO Demo</value>
    </characteristic>

    <!-- 2A01: org.bluetooth.characteristic.gap.appearance -->
    <characteristic uuid="2A01" id="c_appearance">
        <description>Appearance</description>
        <properties read="true" const="true" />
        <!-- Generic device, Generic category -->
        <value type="hex">0000</value>
     </characteristic>

</service>
```

Figure 4: GAP service

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

Table 1: GAP service description

Note in the XML definition above the use of the **id** attribute on <characteristic> tags. These named identifiers allow for programmatic access to characteristics from within your BGScript source file, rather than needing to use a numeric handle. The numeric handle is generated by the BGBuild compiler during the build process, and a simple text file called **attributes.txt** is created which contains a set of name/number pairs on each line. You can refer to this file if you need to know the numeric handle of any characteristic for which you have assigned a named identifier.

The **id** attribute on <service> tags is used only for including service definitions within other services, which we will not do in this exercise. Services cannot be directly referenced or modified from within BGScript, since all data operations are technically done on characteristics only, rather than services.

In both cases, the **id** value is completely arbitrary, but it is a good idea to follow a helpful naming convention. I tend to split words with underscores, and to begin service IDs with “s_” and characteristic IDs with “c_”. The only requirement is that they are alphanumeric (underscores allowed) and that they do not overlap with BGScript keywords.

3.3.2 Hello IO Service

Our custom profile will include the custom Hello IO service, which has no official definition as far as the Bluetooth SIG is concerned. However, we can still describe the structure here in the same manner as above, for the sake of clarity. The **Hello IO** service is defined as below for this lab:

Characteristic	UUID	Length	Type	Support	Security	Properties
User Name	`e2df8530-49bc-4a9c-b7c2-79b6f9c1a9eb`	Variable (max 12B)	UTF8	Mandatory	None	Read, Write
Greeting	`2fe67f97-2797-440c-9b8e-9d0ef2415df4`	Variable (max 20B)	UTF8	Mandatory	None	Indicate
Port Status	`12b45989-8e38-42e4-afb6-f68cc2be254b`	3 bytes	Hex	Mandatory	None	Read (user)

Table 2: Hello IO Service description

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

```
<!-- custom Hello IO service -->
<service uuid="1006e5f7-9f72-4317-8a6c-b640b4a56659" advertise="true">
    <description>Hello IO Service</description>
    <characteristic uuid="e2df8530-49bc-4a9c-b7c2-79b6f9c1a9eb" id="c_user_name">
        <description>Hello User Name</description>
        <properties read="true" write="true" />
        <value length="12" variable_length="true" />
    </characteristic>
    <characteristic uuid="2fe67f97-2797-440c-9b8e-9d0ef2415df4" id="c_greeting">
        <description>Hello Greeting</description>
        <properties indicate="true" />
        <value length="20" variable_length="true" />
    </characteristic>
    <characteristic uuid="12b45989-8e38-42e4-afb6-f68cc2be254b" id="c_port_status">
        <description>Hello Port Status</description>
        <properties read="true" />
        <value length="3" type="user" />
    </characteristic>
</service>
```

Figure 5: Hello IO Service

The **Hello IO** service is explained below:

- The **advertise="true"** option is needed for the for the **<service>** tag. When the sensor advertises, the UUID of the service is included in the data field of the advertisement packets, so the device can be easily identified by remote client devices. Many devices (including iPhones/iPads) often filter scans based on UUID, so if this is not included in the advertisement packet, your device will not be seen by the client device.
- The **Hello User Name** characteristic is only allowed to be up to 12 bytes long. This is because we will be sending back the name as part of a larger string ("Hello, [name]!") in the **Hello Greeting** attribute, and only 20 bytes may be sent at a time in a single BLE packet. It is possible to work with longer characteristics with special API functions, but we will not attempt this for the basic lab exercise.
- **Read**, **write**, and **indicate** properties are enabled on the various attributes as described in the service specification above. Using **indicate** means that the client will be able to "subscribe" (enable indications) to this attribute, and then any value updates done by the peripheral will be automatically pushed out to the remote client.

- The **Hello Port Status** characteristic has its “type” attribute set to “user” instead of something else. This means that our BGScript application will be responsible for handling any remotely requested operations. Typically, the stack takes care of this for you and keeps an internal RAM-stored buffer for each characteristic’s value. However, in this case, this will not serve our purpose very well, since we always want to send back the current port status. Specifying the “user” type will allow this functionality. The actual control logic will be handled later on in the exercise.

3.3.3 Summary

The full GATT database implementation is shown below. This should be the content placed in the **gatt.xml** file for your project and saved before continuing:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <!-- 1800: org.bluetooth.service.generic_access -->
    <service uuid="1800" id="s_generic_access">
        <description>Generic Access</description>

        <!-- 2A00: org.bluetooth.characteristic.gap.device_name -->
        <characteristic uuid="2A00" id="c_device_name">
            <description>Device Name</description>
            <properties read="true" const="true" />
            <value>Hello IO Demo</value>
        </characteristic>

        <!-- 2A01: org.bluetooth.characteristic.gap.appearance -->
        <characteristic uuid="2A01" id="c_appearance">
            <description>Appearance</description>
            <properties read="true" const="true" />
            <!-- Generic device, Generic category -->
            <value type="hex">0000</value>
         </characteristic>
    </service>

    <!-- custom Hello IO service -->
    <service uuid="1006e5f7-9f72-4317-8a6c-b640b4a56659" advertise="true">
        <description>Hello IO Service</description>
        <characteristic uuid="e2df8530-49bc-4a9c-b7c2-79b6f9c1a9eb" id="c_user_name">
            <description>Hello User Name</description>
            <properties read="true" write="true" />
            <value length="12" variable_length="true" />
        </characteristic>
        <characteristic uuid="2fe67f97-2797-440c-9b8e-9d0ef2415df4" id="c_greeting">
            <description>Hello Greeting</description>
            <properties indicate="true" />
            <value length="20" variable_length="true" />
        </characteristic>
        <characteristic uuid="12b45989-8e38-42e4-afb6-f68cc2be254b" id="c_port_status">
            <description>Hello Port Status</description>
            <properties read="true" />
            <value length="3" type="user" />
        </characteristic>
    </service>
</configuration>
```

Figure 6: Hello IO Profile Full GATT database

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

The **config.xml** file contains the extended application configuration for BLE112 device. This file is not necessary for this project since all of the default values are acceptable. Options which can be controlled in this file include UART optimization, script timeout (useful if you use “while” loops), and multiple connection support (only useful on the master end of things, which this is not).

3.5 BGScript for Hello IO Project (**hello_io.bgs**)

Our Hello IO project implements a standalone BLE peripheral device where no external host processor is needed. The application logic is created as a BGScript source file and the code is explained in this section.

BGScript uses an event-based programming approach. The script is executed when an event takes place, and the programmer may register listeners for various events. The Hello IO project uses a few different event listeners, and will require a dozen or so user-declared variables. Let's review how the whole project will be implemented at a high level:

3.5.1 Functional overview

The Hello IO project will work in the following way:

- Advertising will not start automatically on boot, but instead will be toggled by the P0_0 button on the DKBLE112 board. This means that our application will setup and listen for a GPIO interrupt on that pin.
- Sending out the greeting will be accomplished by the P0_1 button on the DKBLE112 board. This also will be detected with a GPIO interrupt.
- Whenever the remote client writes a new "user name" attribute value, we should update the buffer storing the greeting string to include the new name so that it is ready to send when necessary.
- Whenever the remote client reads the "port status" attribute value, we should spontaneously read the actual GPIO status and send that back. This value will not need to be stored, since it may change frequently and will always be re-read anyway.

Now that we have an idea of the functionality, we can move on to the code.

3.5.2 Declaring variables

All user variables must be declared globally (typically at the top of the script). Variables in BGScript may be either **32-bit signed integers** or **byte arrays**. There are no other available datatypes, including floats or pointers. Additionally, due to the limited amount of RAM available on the internal CC2540 processor inside the BLE112, only 255 bytes worth of user-declared variables may be added.

As it turns out, the Hello IO project needs a few different variables for all of the possible functionality. Add the following content to the top of your BGScript source file:

```
dim io_buf(3)           # buffer for IO status reads (19 bits of data = 3 bytes)
dim greeting_buf(20)    # buffer for greeting message (20 bytes max)
dim user_name_buf(12)   # buffer for user name attribute value
dim user_name_len       # length of name written to c_user_name attribute
dim advertising         # advertisement status
dim connected           # connection status
dim encrypted           # encryption status
dim gpio_delta          # de-bounce comparison variable
dim ipr_result          # container for io_port_read() call's returned "result" value
dim ipr_port            # container for io_port_read() call's returned "port" value
dim ipr_data            # container for io_port_read() call's returned "data" value
```

Most of these are explained well enough in the comments. The ipr_* variables are used as return values later on in the code that reads the GPIO levels before sending the data back to the client. It is important to note that all return values must be stored in user-defined variables such as these, and that they are not allocated and named at runtime for you. You will see where they are used inside the "**attributes_user_read_request**" event handler.

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

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

In this lab, the following tasks take place in the **system_boot** event handler:

- Initialize status tracking variables and default user name and greeting data
- Set up GPIO interrupts for catching button presses

Add this event handler code to your script:

```
event system_boot(major, minor, patch, build, ll_version, protocol_version, hw)
    # initialize status vars
    advertising = 0
    connected = 0
    encrypted = 0

    # initialize simple GPIO debouncing test var to 0 (first press will be accepted)
    gpio_delta = 0

    # initialize use name
    user_name_buf(0:1) = $42 # 'B'
    user_name_buf(1:1) = $6c # 'l'
    user_name_buf(2:1) = $75 # 'u'
    user_name_buf(3:1) = $65 # 'e'
    user_name_buf(4:1) = $67 # 'g'
    user_name_buf(5:1) = $69 # 'i'
    user_name_buf(6:1) = $67 # 'g'
    user_name_buf(7:1) = $61 # 'a'
    user_name_len = 8

    # initialize greeting
    greeting_buf(0:1) = $48 # 'H'
    greeting_buf(1:1) = $65 # 'e'
    greeting_buf(2:1) = $6c # 'l'
    greeting_buf(3:1) = $6c # 'l'
    greeting_buf(4:1) = $6f # 'o'
    greeting_buf(5:1) = $2c # ','
    greeting_buf(6:1) = $20 # ' '
    memcpy(greeting_buf(7), user_name_buf(0), user_name_len)
    greeting_buf(7 + user_name_len:1) = $21 # '!'
    # (when we write the greeting to the c_greeting attribute later,
    # it will always be user_name_len+8 bytes long)

    # set user name attribute value
    call attributes_write(c_user_name, 0, 8, user_name_buf(0:8))

    # enable interrupt on P0_0 and P0_1 rising edge
    # (parameters are port=0, bitmask=0b00000011, edge=rising)
    call hardware_io_port_config_irq(0, 3, 0)
end
```

Notable functions and commands in use here:

- **memcpy** (built-in BGScript function)
- **attributes_write** (BGAPI command)
- **hardware_io_port_config** (BGAPI command)

The “memcpy” function is documented in the BGScript User Guide, and the two BGAPI commands can be found in the API Reference Guide.

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

When the *Bluetooth* connection is established a connection event occurs. For this purpose, an event listener is added to the BGScript code which tracks the connection status. This project does not do anything specific with the connection info, but it is often helpful to have it.

Note that if **bonding** (e.g. pairing) is used, then a second **connection_status** event will be raised when the connection becomes encrypted after bonding. This simple project does not make use of bonding or encryption, but the stub code is included for reference.

Add this event handler code to your script:

```
event connection_status(connection, flags, address, address_type, conn_interval, timeout, latency, bonding)
    # check for "new connection established" event
    if (flags & $05) = $05 then
        # update status vars
        advertising = 0
        connected = 1
    end if

    # check for "encrypted" status (e.g. connected + bonded)
    if (flags & $02) = $02 then
        # update status fars
        encrypted = 1
    end if
end
```

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

If the *Bluetooth* connection is a lost, a disconnection event occurs. For this purpose, an event listener is added to the BGScript code for reference. As mentioned above, the connection status is not used for the basic functionality of this lab.

Note that a Bluetooth *Smart* device will not automatically resume advertising when a connection is lost. It will be put into an **idle** state. If you want the device to resume advertising upon disconnection, you must explicitly tell it to do so. In this project, we are intentionally toggling the advertising state using a GPIO interrupt, so our event handler will not do this. The code to do so is present but commented out here.

Add this event handler code to your script:

```
event connection_disconnected(handle, result)
    # if disconnected, return to advertisement mode (disabled for lab environment)
    #call gap_set_mode(gap_general_discoverable, gap_undirected_connectable)

    # update status vars
    connected = 0
    encrypted = 0
end
```

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

For controlling the behavior of the application, this project makes use of two buttons on the DKBLE112 board:

- **P0_0** is used to toggle advertising mode. It is off (not advertising) by default, but pressing the button will enable it. Or if it is already on, pressing the button will disable it.
- **P0_1** is used to push the greeting to the remote client

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

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

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

0 = Port 0

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

0 = Rising edge

Remember that in **hardware.xml**, we also configured Port 0 to be pulled down, so that the rising edge will be more reliably detected. The I/O signals are very simply de-bounced with a 100ms event interval requirement.

Add this event handler code to your script:

```
event hardware_io_port_status(timestamp, port, irq, state)
    # only accept this press if it's more then 100ms after the last one (32768 = 1 sec)
    if timestamp - gpio_delta > 3277 then
        if port = 0 then
            if (irq & 1) = 1 && connected = 0 then
                # P0_0 is HIGH and the source of this interrupt
                # toggle advertising mode if we aren't connected

                if advertising = 0 then
                    # update status vars
                    advertising = 1

                    # start advertising in general discoverable / undirected connectable mode (normal)
                    call gap_set_mode(gap_general_discoverable, gap_undirected_connectable)
                else
                    # update status vars
                    advertising = 0

                    # stop advertising
                    call gap_set_mode(0, 0)
                end if
            end if

            if (irq & 2) = 2 && connected = 1 then
                # P0_1 is HIGH and the source of this interrupt
                # write greeting value if connected

                # write value to c_greeting attribute, which will push the data
                # to the remote client if they have enabled indications on it
                call attributes_write(c_greeting, 0, user_name_len + 8, greeting_buf(0:user_name_len + 8))
            end if
        end if

        # update last delta to current time
        gpio_delta = timestamp
    end if
end
```

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

The ATTribute protocol is used to transmit data over a *Bluetooth* connection. A remote device can use an ATT write operation to write up to 20 bytes of data at a time. In the Hello IO profile, only one attribute is writable: **Hello User Name**.

This BGScript code checks to make sure the attribute handle matches the **c_user_name** ID value first (not technically necessary in this case since only one attribute is writable, but very good practice), and then creates a new greeting value to use for the **Hello Greeting** characteristic. It will then be available to push out to the remote client via indications upon pressing the P0_1 button on the DKBLE112 board. Note that this greeting value is only stored in a variable and not actually written to the characteristic, because the act of writing it is what causes the stack to perform the indication. This is done by a GPIO interrupt in the previous event handler.

Add this event handler code to your script:

```
event attributes_value(connection, reason, handle, offset, value_len, value_data)
    # check to make sure it's the one we're interested in
    if handle = c_user_name then
        # copying new data info user name buffer and save length
        memcpy(user_name_buf(0), value_data(0), value_len)
        user_name_len = value_len

        # update greeting string to use the new name
        memcpy(greeting_buf(7), user_name_buf(0), user_name_len)
        greeting_buf(7 + user_name_len:1) = $21 # '!'
    end if
end
```

3.5.8 Data: Modifying data requested by the client (**attributes_user_read_request**)

One final bit of wizardry is required to make sure we always send the current GPIO port status whenever it is requested. It would be possible to do this with two operations, first detecting a “tell me the status” command written to an attribute and then push the result using an indication, but this takes more time and is less elegant. This method allows us to intercept the request and send back whatever we want.

Add this event handler code to your script:

```
event attributes_user_read_request(connection, handle, offset, maxsize)
    # check to make sure it's the one we're interested in
    if handle = c_port_status then
        # IO port status, so now were going to read the status, store it, and send it back

        # read and store Port0
        call hardware_io_port_read(0, $ff)(ipr_result, ipr_port, ipr_data)
        io_buf(0:1) = ipr_data

        # read and store Port1
        call hardware_io_port_read(1, $ff)(ipr_result, ipr_port, ipr_data)
        io_buf(1:1) = ipr_data

        # read and store Port2 (only 3 pins available, mask is 0x07)
        call hardware_io_port_read(2, $07)(ipr_result, ipr_port, ipr_data)
        io_buf(2:1) = ipr_data

        # send back buffer value (use "0" for the status to indicate no error)
        call attributes_user_read_response(connection, 0, 3, io_buf(0:3))
    end if
end
```

You should now have a complete project with all necessary source code. The full set of project files with all of the above code should additionally be available along with this document for reference.

4 Compiling and flashing the Hello IO project

You should now have a total of four files in your project folder:

a. project.xml (or project.bgproj)

b. hardware.xml

c. gatt.xml

d. hello_io.bgs

It is now ready to compile and flash onto the development kit.

4.1.1 Connect the CC Debugger

In order to flash a new firmware image onto the device, the CC debugger must be connected. This is a pretty straightforward process:

a. Connect a mini USB cable between the CC debugger and your PC

b. Connect the small 0.1” to 0.05” adapter board to the CC debugger

c. Connect the small 10-pin 0.05” ribbon cable between the CC debugger and the DKBLE112, making sure that both rows of pins are aligned properly in the holes and that the red wire (pin 1) is on the far right edge of the header on the dev kit, not on the side closest to the USB connector.

d. With the DKBLE112 powered and the CC debugger connected, press the small black button on the CC debugger and verify that the debugger’s LED is **green**, not red. This indiates a proper connection. Note that you may need to make extra sure that the 0.1” to 0.05” adapter board is seated properly, as it has been known to make intermittent connections.

e. The CC debugger may not enumerate properly without the correct drivers. If necessary, you can download the drivers directly from TI, or download and install the TI SmartRF Flash Programmer utility which includes drivers for the CC debugger.

4.1.2 Compile and flash the project with BLE Update

The following steps will build and flash the modified firmware onto the module, assuming the CC debugger is properly connected as described above:

a. Run the BLE Update utility

b. Make sure the CC debugger device is selected from the Port dropdown

c. Click the ”Info” button in BLE Update to verify that the debugger can communicate with the module

d. Click the ”Browse” button and locate the ”project.xml” file (or ”project.bgproj”) file your project folder

e. Click the ”BGBuild” menu and verify that an SDK is selected

- If not, click the ”Select manually...” item and browse to the **/bin/bgbuild.exe** file from the extracted SDK archive

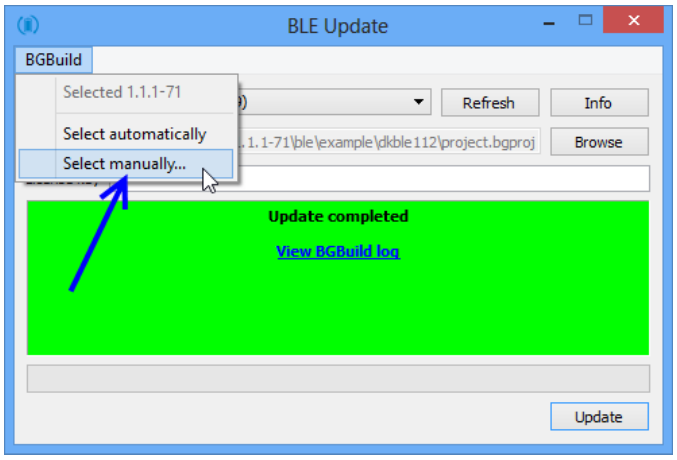


Figure 7: Manually selecting a BGBuild executable

f. Click the ”Update” button to compile and flash the project onto the module

You should now have your custom Hello IO project running on the development kit, and can proceed on to testing with the BLEGUI application and a BLED112 dongle.

5 Testing Hello IO with BLEGUI

This section assumes you have already downloaded and extracted the BLE SDK archive, which contains the BLEGUI application as **/bin/blegui2.exe**. If you do not have it already, please download and extract it.

5.1 Preparing the BLED112 USB dongle

Plug the BLED112 into an available USB port on your PC. If you have already installed the USB CDC drivers on your computer, then it should enumerate as a new COM port (make a note of which port this is) and begin working immediately. If you have not installed the USB CDC driver and you receive an "Unknown device" error, then you will need to go to Device Manager via the Control Panel and install the driver from the **/windrv** folder found inside the SDK archive.

5.2 Find and connect to the Hello IO peripheral device

Follow these steps to connect to the dev kit running Hello IO

a. Start BLEGUI (find and run the **/bin/blegui2.exe** application from the extracted SDK)
b. Select the "Bluegiga Bluetooth Low Energy (COM[x])" device in the dropdown near the top
c. Click the "Attach" button, at which point you should see a green Connected indicator
d. Make sure the box on the left side marked "Active Scanning" is checked
e. Make sure the "Generic" Scan type option is selected
f. Click the "Start" button to begin scanning for BLE devices
g. Press the P0_0 button on the DKBLE112 board to begin advertising
h. You should see your device appear in BLEGUI in the area on the right
i. Observe the continuously changing RSSI value which indicates signal strength, and note how it changes if you move the DKBLE112 around or put some obstruction between the BLED112 and DKBLE112
j. Click the "Connect" button next to your advertising peripheral device in the list

If everything went smoothly, you are now connected to your Hello IO BLE device!

5.3 Explore the GATT structure to find relevant attributes

Before you can work with the GATT database, you will need to know how to find what you are looking for. Remember that our custom Hello IO service contains three attributes, or "characteristics," each of which has its own 128-bit UUID. The service itself also has a 128-bit UUID. For reference, here are the values we used:

- `1006e5f7-9f72-4317-8a6c-b640b4a56659` – **Hello IO** service
- `e2df8530-49bc-4a9c-b7c2-79b6f9c1a9eb` – **User Name** characteristic
- `2fe67f97-2797-440c-9b8e-9d0ef2415df4` – **Greeting** characteristic
- `12b45989-8e38-42e4-afb6-f68cc2be254b` – **Port Status** characteristic

Assuming you are already connected, perform the following steps to identify the right services and attributes:

a. Click the ”GATT” button to the right of your device in the list in BLEGUI

b. Click the ”Service Discover” button

c. Select the service with UUID=**`1006e5`...** (this is our Hello IO service)

d. Click the ”Descriptors Discover” button

e. Find these three descriptors:

- UUID=**`e2df85`**... (this is the **User Name** attribute)
- UUID=**`2fe67f`**... (this is the **Greeting** attribute)
- UUID=**`12b459`**... (this is the **Port Status** attribute)

f. Make note of the numeric handle value for each, for easy reference

You have now discovered and identified all of the important pieces of the GATT database for our testing purposes.

5.4 Test the greeting and set a new name

Now that you know which attributes are which, you can test the functionality of the BGScript application logic. To do this, we will need to enable **indications** on the **Greeting** characteristic, because that’s how the greeting string is pushed out to us. Here are the steps to accomplish this and test it:

a. Locate the **Greeting** attribute (UUID=**`2fe67f`**...)

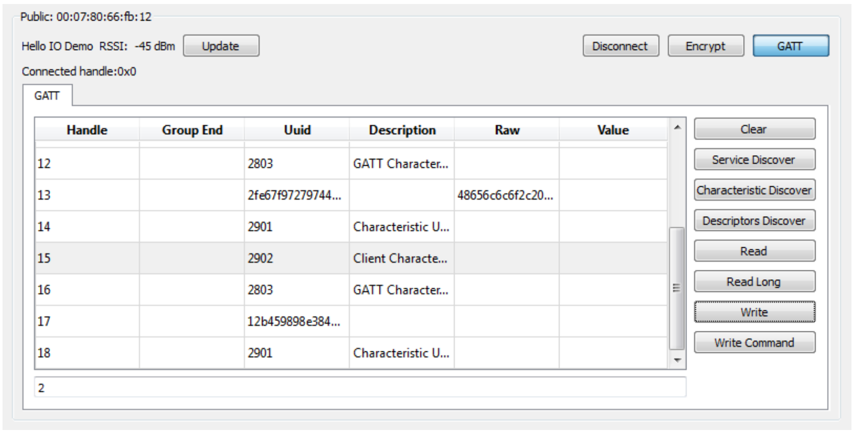
b. Select the Client Characteristic Configuration attribute with UUID=2902) immediately below it

c. Enter the value ”2” in the long text field immediately below the GATT listing

d. Click the ”Write” button to write ”2” to the Client Characteristic Configuration attribute

e. Make sure you can see the **Greeting** attribute in the list, then click the **P0_1** button on the DKBLE112

f. This wil push a new value to the Greeting attribute in BLEGUI, displayed in hex: **`48656c6c6f2c20426c75656769676121`**



Handle	Group End	Uuid	Description	Raw	Value
12		2803	GATT Character...		
13		2fe67f97279744...		48656c6c6f2c20...	
14		2901	Characteristic U...		
15		2902	Client Characte...		
16		2803	GATT Character...		
17		12b459898e384...			
18		2901	Characteristic U...		

2

Figure 8: Receiving indicated value from the Greeting attribute

Of course, this is not very easy to read. BLEGUI displays all data in hexadecimal notation, so if you are checking on an ASCII string, you have to convert it using a tool like asciitohex.com. If you convert the above value to ASCII, you get the following string:

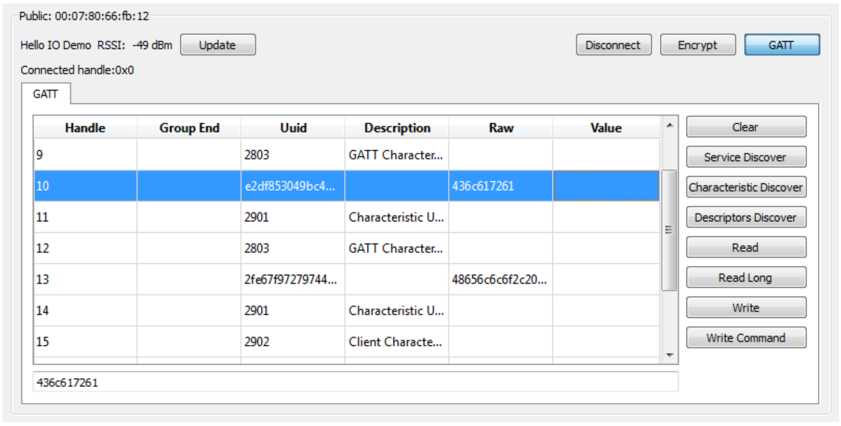
`Hello, Bluegiga!`

Now that’s more like it. But what if we want a different greeting?

To change the greeting to something more personal, just write a new value to the User Name attribute:

a. Locate and select the **User Name** attribute (UUID=**`e2df85`**...)

b. Enter a new value **in hexadecimal** in the long text field (e.g. ”Clara” is ”436c617261”)



Handle	Group End	Uuid	Description	Raw	Value
9		2803	GATT Character...		
10		e2df853049bc4...		436c617261	
11		2901	Characteristic U...		
12		2803	GATT Character...		
13		2fe67f97279744...		48656c6c6f2c20...	
14		2901	Characteristic U...		
15		2902	Client Characte...		

436c617261

Figure 9: Writing a new name to the User Name attribute

c. Click the ”Write” button to write the new name to the **User Name** attribute

d. Make sure you can see the **Greeting** attribute in the list, then click the **P0_1** button on the DKBLE112

e. This wil push a new value to the **Greeting** attribute in BLEGUI, displayed in hex: **`48656c6c6f2c20436c61726121`**

Once again, if we convert this back to ASCII, we get the following:

`Hello, Clara!`

You can use any name string in step (b) above as long as it is 12 bytes or less. Longer values will be truncated.

5.5 Request the IO status

One test remains, and that is to read the status of the GPIO pins. This is extremely simple to do:

a. Locate and select the **Port Status** attribute (`12b459`...)

b. Click the "Read" button to read the status

c. This will return a 3-byte hex value showing P0, P1, and P2 in that order:
   `008200`

   ...or, if the RS232 switch is on:
   `248200`

You may get other values depending on the position of the Accelerometer switch. So, why are all of those pins reading as logic high when **hardware.xml** has pulled them all low, and (presumably) you haven't connected them to anything else? To answer that, first we must break down the actual bit values.

Each byte represents all of the pins on that respective port. The bits in each byte are "big-endian," which means the most significant bit (the one representing 128 instead of 1) comes first. The bits are indexed in the same order as the pins, so the 7th bit of the first byte is P0_7, the 6th bit is P0_6, etc. down to the so-called "zeroth" bit being P0_0. In the case of the "248200" value with the RS232 switch on, this means the pin logic values are these:

PORT 0								**PORT 1**								**PORT 2**		
_7	_6	_5	_4	_3	_2	_1	_0	_7	_6	_5	_4	_3	_2	_1	_0	_2	_1	_0
LO	LO	HI	LO	HI	LO	LO	LO	HI	LO	LO	LO	LO	LO	HI	LO	LO	LO	LO

So all of the pins are low (as expected) except for P0_3, P0_5, P1_1, and P1_7 (not expected). Each has a good reason given the state of the board:

a. P0_3 (0x04) is wired to the RS232 level shifter's RTS pin. Nothing is connected to the DB9 port on the board, so this is de-asserted (high).

b. P0_5 (0x20) is wired to the RS232 level shifter's RX pin. This also is de-asserted (idle) .

c. P1_1 (0x02) has no internal pull capability (it is a high-drive output) and no external pull, so it is simply floating. If you touch it with your fingertip and read the I/O status again, it should drop to 0.

d. P1_7 (0x80) controls the DC/DC converter automatically through the <pmux> configuration option, and we read the value right in the middle of a radio operation, so this is high (active).

You can also turn on the "POTENTIOMETER" switch and play with the potentiometer on the board, which is connected to P0_6. About half-way through the rotation, you can see the 0x40 value for the PORT 0 byte change if you keep reading the status.

Congratulations!

You have now worked through all of the steps necessary to understand the basics of implementing your own custom BLE profile.

6 Contact Information

Sales:	sales@bluegiga.com
Technical support:	support@bluegiga.com http://techforum.bluegiga.com
Orders:	orders@bluegiga.com
WWW:	www.bluegiga.com www.bluegiga.hk

Head Office / Finland:

Phone: +358-9-4355 060
Fax: +358-9-4355 0660
Sinikalliontie 5A
02630 ESPOO
FINLAND

Postal address / Finland:

P.O. BOX 120
02631 ESPOO
FINLAND

Sales Office / USA:

Phone: +1 770 291 2181
Fax: +1 770 291 2183
Bluegiga Technologies, Inc.
3235 Satellite Boulevard, Building 400, Suite 300
Duluth, GA, 30096, USA

Sales Office / Hong-Kong:

Phone: +852 3182 7321
Fax: +852 3972 5777
Bluegiga Technologies, Inc.
19/F Silver Fortune Plaza, 1 Wellington Street,
Central Hong Kong