UG118: *Bluetooth*®Profile Toolkit Developer's Guide

This version of UG118 has been deprecated.

For the latest version, see docs.silabs.com.

****

KEY POINTS

- Understanding Bluetooth GATT profiles, services, characteristics, attribute protocol
- Building the GATT database with the Profile Toolkit

Bluetooth GATT services and characteristics are the basis of the Bluetooth data exchange. They are used to describe the structure, access type, and security properties of the data exposed by a device, such as a heart-rate monitor. Bluetooth services and characteristics have a well-defined and structured format, and they can be easily described using XML mark-up language.

The Profile Toolkit is an XML-based mark-up language for describing the Bluetooth services and characteristics, also known as the GATT database, in both easy human-readable and machine-readable formats. This guide walks you through the XML syntax used in the Profile Toolkit and instructs you how to easily describe your own Bluetooth services and characteristics, configure the access and security properties, and how to include the GATT database as a part of the firmware.

This guide also contains practical examples showing the use of both standardized Bluetooth and vendor-specific proprietary services. These examples provide a good starting point for your own development work.

1. Understanding Profiles, Services, Characteristics, and the Attribute Protocol

This section provides a basic explanation of Bluetooth profiles, services, and characteristics, and also explains how the Attribute protocol is used in the data exchange between the GATT server and client. Further information on these topics can be found on the Bluetooth SIG website at: https://www.bluetooth.com/specifications/specs/.

1.1 GATT-Based Bluetooth Profiles and Services

A *Bluetooth* profile specifies the structure in which data is exchanged. The profile defines elements, such as services and characteristics used in a profile, but it may also contain definitions for security and connection-establishment parameters. Typically a profile consists of one or more services which are needed to accomplish a high-level use case, such as heart-rate or cadence monitoring. Standardized profiles allow device and software vendors to build inter-operable devices and applications.

1.2 Services

A service is a collection of data composed of one or more characteristics used to accomplish a specific function of a device, such as battery monitoring or temperature data, rather than a complete use case.

1.3 Characteristics

A characteristic is a value used in a service, either to expose and/or exchange data and/or to control information. Characteristics have a well-defined known format. They also contain information about how the value can be accessed, what security requirements must be fulfilled, and, optionally, how the characteristic value is displayed or interpreted. Characteristics may also contain descriptors that describe the value or permit configuration of characteristic data indications or notifications.

1.4 The Attribute Protocol

The Attribute protocol enables data exchange between the GATT server and the GATT client. The protocol also provides a set of operations, namely how to query, write, indicate, or notify the data and/or control information between the two GATT parties.

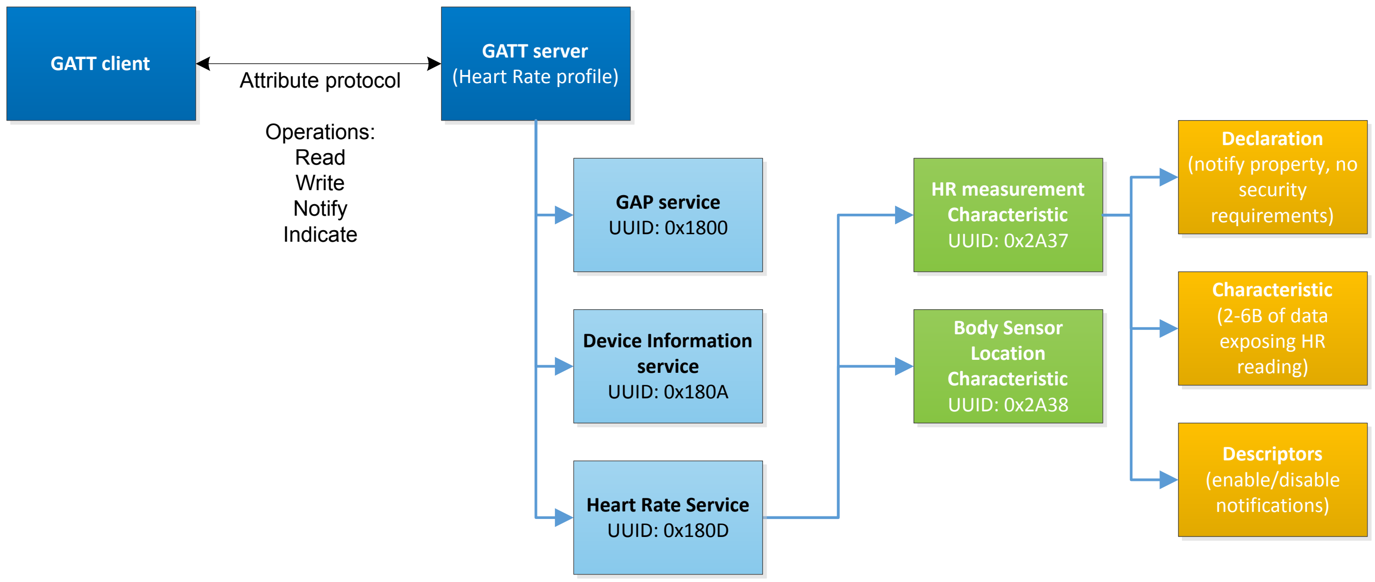


Figure 1.1. Profile, Service, and Characteristic Relationships

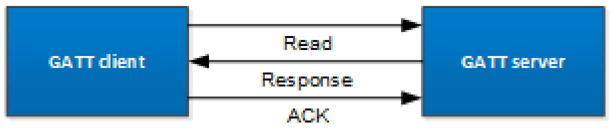


Figure 1.2. Attribute Read Operation

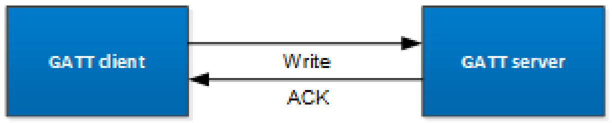


Figure 1.3. Attribute Write Operation

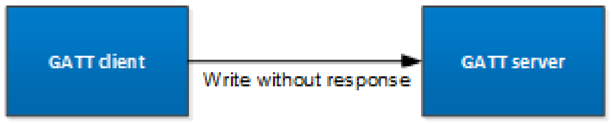


Figure 1.4. Attribute Write without Response Operation

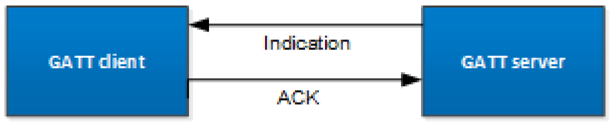


Figure 1.5. Attribute Indicate Operation

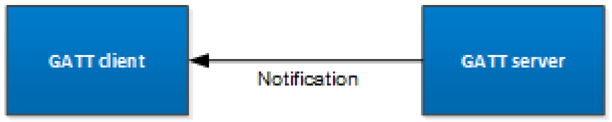


Figure 1.6. Attribute Notify Operation

2. Building the GATT Database with Profile Toolkit

This section of the document describes the XML syntax used in the Bluetooth Profile Toolkit and walks you through the different options you can use when building Bluetooth services and characteristics.

A few practical GATT database examples are also shown.

2.1 General Limitations

The table below shows the limitations of the GATT database supported by the EFR32BG devices.

Item	Limitation	Notes
Maximum number of characteristics	Not limited; practically restricted by the overall number of attributes in the database	All characteristics which do NOT have the property **const="true"** are included in this count.
Maximum length of a **type="user"** characteristic	255 bytes	These characteristics are handled by the application, which means that the amount of RAM available for the application will limit this. Note: GATT procedures **Write Long Characteristic Values**, **Reliable Writes** and **Read Multiple Characteristic Values** are not supported for these characteristics.
Maximum length of a **type="utf-8/hex"** characteristic	255 bytes	If **const="true"** then the amount of free flash on the device defines this limit. If **const="false"** then RAM will be allocated for the characteristic for storing its value. The amount of free flash available on the device used defines this.
Maximum number of attributes in a single GATT database	255	A single characteristic typically uses 3-5 attributes.
Maximum number of notifiable characteristics	64	
Maximum number of capabilities	16	The logic state of the capabilities will determine the visibility of each service/characteristic.

2.2 <gatt>

The GATT database along with the services and characteristics must be described inside the XML attribute **<gatt>**.

Parameter	Description
out	Filename for the GATT C source file **Value:** Any UTF-8 string. Must be valid as a filename and end with '.c' **Default:** gatt_db.c
header	Filename for the GATT C header file **Value:** Any UTF-8 string. Must be valid as a filename and end with '.h' **Default:** gatt_db.h
db_name	GATT database structure name and the prefix for data structures in the GATT C source file. **Value:** Any UTF-8 string; must be valid in C. **Default:** bg_gattdb_data
name	Free text, not used by the database compiler **Value:** Any UTF-8 string **Default:** Nothing
prefix	Prefix to add to each 'id' name for defining the handle macro that can be referenced from the C application. **Value:** Any UTF-8 string. Must be valid in C. **Default:** gattdb_ For example: If prefix="gattdb_" and id="temp_measurement" for a particular characteristic, then the following will be generated in the GATT C header file: #define gattdb_temp_measurement X (where X is the handle number for the temp_measurement characteristic)
generic_attribute_service	If it is set to true, Generic Attribute service and its serv-ice_changed characteristic will be added in the beginning of the database. The Bluetooth stack takes care of database structure change detection and will send service_changed notifications to clients when a change is detected. In addition, this will enable the GATT-caching feature introduced in Bluetooth 5.1. **Values:** **true:** Generic Attribute service is automatically added to the GATT database and GATT caching is enabled. **false:** Generic Attribute service is not automatically added to the GATT database and GATT caching is disabled. **Default:** false
gatt_caching	The GATT caching feature is enabled by default if generic_attrib-ute_service is set to true. However, it can be disabled by setting this attribute to false.

Example: A GATT database definition.

```
<?xml version="1.0" encoding="UTF-8" ?>
<gatt out="my_gatt_db.c" header="my_gatt_db.h" db_name="my_gatt_db_" prefix="my_gatt_"
   generic_attribute_service="true" name="My GATT database">
```

```
…
</gatt>
```

2.3 <capabilities_declare>

The GATT database services and characteristics can be made visible/invisible by using **capabilities**. A capability must be declared in a <capability> element and all capabilities in a GATT database must be first declared in a <capabilities_declare> element consisting of a sequence of <capability> elements. The maximum number of capabilities in a database is 16.

This new functionality does not affect legacy GATT XML databases (prior to Silicon Labs Bluetooth stack version 2.4.x). Because they don't have any capabilities explicitly declared, all services and characteristics will remain visible to a remote GATT client.

Example: Capabilities declaration

```
  <capabilities_declare>
    <capability enable="false">feature_1</capability>
    <capability enable="false">feature_2</capability>
  </capabilities_declare>
```

2.3.1 <capability>

Each capability must be declared individually within a <capabilities_declare> element using the <capability> element. The <capability> element has one attribute named "enable" that indicates the capability's default state at database initialization.

The text value of the <capability> element will be the identifier name for that capability in the generated database C header. Thus, it must be valid in C.

Inheritance of Capabilities

Services and characteristics can declare the capabilities that they want to use. If no capabilities are declared, then the following inheritance rules apply:

1. A service that does not declare any capabilities will have all the capabilities from <capabilities_declare> element.
2. A characteristic that does not declare any capabilities will have all the capabilities from the service that it belongs to. If the service declares a subset of the capabilities in <capabilities_declare>, then only that subset will be inherited by the characteristic.
3. All attributes of a characteristic inherit the characteristic's capabilities.

Visibility

Capabilities can be enabled/disabled to make services and characteristics visible/invisible to a GATT client according with the following logic:

1. A service and all its characteristics are **visible** when **at least one** of its capabilities is **enabled**.
2. A service and all its characteristics are **invisible** when **all** of its capabilities are **disabled**.
3. A characteristic and all its attributes are **visible** when **at least one** of its capabilities is **enabled**.
4. A characteristic and all its attributes are **invisible** when **all** of its capabilities are **disabled.**

Parameter	Description
enable	Sets the default state of a capability at database initialization. **Values:** **true:** Capability is enabled. **false:** Capability is disabled. **Default:** true

Example: Capabilities declaration

```
<capabilities_declare>
    <!-- This capability is enabled by default and the identifier is cap_light -->
    <capability enable="true">cap_light</capability>
    <!-- This capability is disabled by default and the identifier is cap_color -->
    <capability enable="false">cap_color</capability>
</capabilities_declare>
```

2.4 <service>

The GATT service definition is done with the XML attribute **<service>** and its parameters.

The following table below describes the parameters that can be used for defining the related values.

Parameter	Description
uuid	Universally Unique Identifier. The UUID uniquely identifies a service. 16-bit values are used for the services defined by the Bluetooth SIG and 128-bit UUIDs can be used for vendor specific implementations. **Range:** **0x0000 – 0xFFFF**: Reserved for Bluetooth SIG standardized services **0x00000000-0000-0000-0000-000000000000 - 0xFFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF**: Reserved for vendor specific services.
id	The ID is used to identify a service within the service database and can be used as a reference from other services (include statement). Typically, this does not need to be used. **Value:** Any UTF-8 string
type	The type field defines whether the service is a primary or a secondary service. Typically this does not need to be used. **Values:** **primary**: a primary service **secondary**: a secondary service **Default**: primary
advertise	This field defines if the service UUID is included in the advertisement data. The advertisement data can contain up to 13 16-bit UUIDs or one (1) 128-bit UUID. **Values:** **true**: UUID included in advertisement data **false**: UUID not included in advertisement data **Default**: false **Note:** You can override the advertisement data with the GAP API, in which case this is not valid.

Example: A Generic Access Profile (GAP) service definition.

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

Example: A vendor-specific service definition.

```
<!-- A vendor specific service -->
<service uuid="25be6a60-2040-11e5-bd86-0002a5d5c51b">
      …
</service>
```

Example: A Heart Rate service definition with UUID included in the advertisement data and ID “hrs”.

```
<!-- Heart Rate Service -->
<service uuid="180D" id="hrs" advertise=”true”>
      …
</service>
```

Note: You can generate your own 128-bit UUIDs at: http://www.itu.int/en/ITU-T/asn1/Pages/UUID/uuids.aspx

2.4.1 <capabilities>

A service can declare the capabilities it has with a <capabilities> element. The element consists of a sequence of <capability> elements whose identifiers **must also be part** of the <capabilities_declare> element. The attribute "enable" has no effect in the capabilities declared within this context so it can be excluded.

If a service does not declare any capabilities, it will have **all the capabilities** from **<capabilities_declare>** per the **inheritance rules**.

A service and all its characteristics will be **visible** when **at least one** of its capabilities is **enabled** and **invisible** when **all its capabilities** are **disabled**.

Example: Capabilities declaration

```
<capabilities>
    <capability>cap_light</capability>
    <capability>cap_color</capability>
</capabilities>
```

2.4.2 <informativeText>

The XML element <informativeText> can be used for informative purposes (commenting) and is not exposed in the actual GATT database.

2.4.3 <include>

A service can be included within another service by using the XML attribute **<include>**.

Parameter	Description
id	ID of the included service **Value:** ID of another service

Example: Including Heart Rate service within the GAP service.

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

      <!-- Include HR Service -->
      <include id="hrs” />
      …
</service>
```

2.5 <characteristic>

All the characteristics exposed by a service are defined with the XML attribute **<characteristic>** and its parameters, which must be used inside the **<service>** XML attribute tags.

The table below describes the parameters that can be used for defining the related values.

Parameter	Description
uuid	Universally Unique Identifier. The UUID uniquely identifies a characteristic. 16-bit values are used for the services defined by the Bluetooth SIG and 128-bit UUIDs can be used for vendor specific implementations. **Range:** **0x0000 – 0xFFFF**: Reserved for Bluetooth SIG standardized characteristics. **0x00000000-0000-0000-0000-000000000000 to 0xFFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF** : Reserved for vendor specific characteristics.
id	The ID is used to identify a characteristic. The ID is used within a C application to read and write characteristic values or to detect if notifications or indications are enabled or disabled for a specific characteristic. When the project is built, the generated GATT C header file contains a macro with the characteristic 'id' and corresponding handle value. **Value:**Any UTF-8 string
const	Defines if the value stored in the characteristic is a constant. **Default:** false
name	Free text, not used by the database compiler. **Value:**Any UTF-8 string **Default:** Nothing

Example: Adding Device name characteristic into GAP service.

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

      <!-- Device name -->
      <characteristic uuid="2a00">
            …
      </characteristic>
      …
</service>
```

Example: Adding a vendor-specific characteristic into a vendor-specific service with ID.

```
<!-- A vendor specific service -->
<service uuid="25be6a60-2040-11e5-bd86-0002a5d5c51b">

      <!-- My proprietary data -->
      <characteristic uuid="59cd69c0-2043-11e5-a717-0002a5d5c51b" id="mydata”>
            …
      </characteristic>
      …
</service>
```

2.5.1 <capabilities>

A characteristic can declare the capabilities it has with a <capabilities> element. The element consists of a sequence of <capability> elements whose identifiers **must also be declared (or fully inherited)** by the parent service. The attribute "enable" has no effect in the capabilities declared within this context so it can be excluded.

If a characteristic does not declare any capabilities it will have **all the capabilities** from the **service** that it belongs to per the **inheritance rules**. All attributes of a characteristic inherit the characteristic's capabilities.

A characteristic and all its attributes will be **visible** when **at least one** of its capabilities is **enabled** and **invisible** when **all its capabilities** are **disabled**.

Example: Capabilities declaration

```
<capabilities>
    <capability>cap_light</capability>
    <capability>cap_color</capability>
</capabilities>
```

2.5.2 <properties>

The characteristic's access properties and its permission level are defined by the children attributes of the XML attribute **<properties>**, which must be used inside **<characteristic>** XML attribute tags. A characteristic can have multiple access properties at the same time —for example, it can be readable, writable, or both. Each access property can have a different permission level (for example, encrypted or authenticated).

The following table lists the possible access properties. Each row in the table defines a new attribute under the **<properties>** attribute.

Attribute	Description
read	Characteristic can be read by a remote device. **Values:** **true**: Characteristic can be read **false**: Characteristic cannot be read **Default**: false
write	Characteristic can be written by a remote device **Values:** **true**: Characteristic can be written **false**: Characteristic cannot be written **Default**: false
write_no_response	Characteristic can be written by a remote device. Write without response is not acknowledged over the Attribute Protocol. **Values:** **true**: Characteristic can be written **false**: Characteristic cannot be written **Default**: false
notify	Characteristic has the notify property and characteristic value changes are notified over the Attribute Protocol. Notifications are not acknowledged over the Attribute Protocol. **Values:** **true**: Characteristic has notify property. **false**: Characteristic does not have notify property. **Default**: false **Note**: The *notify* attribute is stored in the SIG defined Client Characteristic Configuration Descriptor (a descriptor with the UUID 0x2902, which will be autogenerated when notifications are enabled). If you manually add a CCCD to the characteristic, the descriptor's value will overwrite this setting.
indicate	Characteristic has the indicate property and characteristic value changes are indicated over the Attribute Protocol. Indications are acknowledged over the Attribute Protocol. **Values:** **true**: Characteristic has indicate property. **false**: Characteristic does not have indicate property. **Default**: false **Note**: The *indicate* attribute is stored in the SIG defined Client Characteristic Configuration Descriptor (a descriptor with the UUID 0x2902, which will be autogenerated when indications are enabled). If you manually add a CCCD to the characteristic, the descriptor's value will overwrite this setting.

Attribute	Description
reliable_write	Allows using a reliable write procedure to modify an attribute; this is just a hint to a GATT client. The Bluetooth stack always allows the use of reliable writes to modify attributes. **Values:** **true**: Reliable write enabled. **false**: Reliable write disenabled. **Default**: false

The following table lists the permission levels or security requirements. Any security requirement can be assigned to any access property.

Attribute	Description
authenticated	Accessing the characteristic value requires an authentication. To access the characteristic with this property the remote device has to be bonded using MITM protection and the connection must be also encrypted. **Values:** **true**: Authentication is required **false**: Authentication is not required **Default**: false
encrypted	Accessing the characteristic value requires an encrypted link. Devices with iOS 9.1 or higher must also be bonded at least with Just Works pairing. **Values:** **true**: Encryption is required **false**: Encryption is not required **Default**: false
bonded	Accessing the characteristic value requires an encrypted link. Devices must also be bonded at least with Just Works pairing. **Values:** **true**: Bonding and encryption are required **false**: Bonding is not required **Default**: false

Example: Device name characteristic with *const* and *read* properties.

```
<!-Device Name-->
<characteristic const = true uuid="2a00">
<properties>
    <read authenticated="false" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

Example: Device name characteristic with *read* and *write* properties to allow the value to be modified by the remote device.

```
<!-Device Name-->
<characteristic uuid="2a00">
<properties>
    <read authenticated="false" bonded="false" encrypted="false"/>
        <write authenticated="false" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

Example: Heart Rate Measurement characteristic with *notify* property.

```
<!-Heart Rate Measurement -->
<characteristic uuid="180D">
<properties>
    <notify authenticated="false" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

Example: Characteristic with *encrypted read* property.

```
<!-Device Name-->
<characteristic uuid="1234">
<properties>
        <read authenticated="false" bonded="false" encrypted="true"/>
</properties>
</characteristic>
```

Example: Characteristic with *authenticated write* property.

```
<!-Device Name-->
<characteristic uuid="1234">
<properties>
    <write authenticated="true" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

Example: Characteristic with *authenticated indicate* properties.

```
<!-Descriptor value changed -->
<characteristic uuid="2A7D">
<properties>
    <indicate authenticated="true" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

2.5.3 <value>

The data type and length for a characteristic is defined with the XML attribute **<value>** and its parameters, which must be used inside the **<characteristic>** XML attribute tags.

The table below describes the parameters that can be used for defining the related values.

Parameter	Description
length	Defines a fixed length for the characteristic or the maximum length if ***variable_length*** is true. If length is not defined and there is a value (e.g. data exists inside <value></value>), then the value length is used to define the length. If both ***length*** and value are defined, then the following rules apply: 1. If ***variable_length*** is false and ***length*** is bigger than the value's length, then the value will be padded with 0's at the end to match the attribute's ***length***. 2. If ***length*** is smaller than the value's length, then the value will be clipped to match ***length***, regardless of whether ***variable_length*** is true or false. **Range:** **0 – 255**: Length in bytes if ***type*** is 'hex', 'utf-8', or 'user' **0 – 512:** Length in bytes if **type** is 'user' **Default**: 0
variable_length	Defines that the value is of variable length. The maximum length must also be defined with the ***length*** attribute or by defining a value. If both ***length*** and value are defined, then the rules described in ***length*** apply. **Values:** **true**: Value is of variable length **false**: Value has a fixed length **Default**: false
type	Defines the data type. **Values:** **hex**: Value type is hex **utf-8**: Value is a string **user:** When the characteristic type is marked as type="user", the application is responsible for initializing the characteristic value and also providing it, for example, when read operation occurs. The Bluetooth stack does not initialize the value or automatically provide the value when it is being read. When this is set, the Bluetooth stack generates ***gatt_server_user_read_request*** or ***gatt_server_user_write_request***, which must be handled by the application. **Default**: utf-8

Example: Heart Rate Measurement characteristic with *notify* property and fixed length of two (2) bytes.

```
<!-Heart Rate Measurement -->
<characteristic uuid="180D">
<value length="2" type="hex" variable_length = "false"/>
<properties>
<notify authenticated="false" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

Example: A variable length vendor-specific characteristic with maximum length of 20 bytes.

```
<!-My proprietary data -->
<characteristic uuid="59cd69c0-2043-11e5-a717-0002a5d5c51b" id="mydata">
<value variable_length="true" length="20" type="hex" />
```

```
<properties>
    <notify authenticated="false" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

Example: The value and length of a characteristic can also be defined by typing the actual value inside the <value> tags.

```
<characteristic          const="true"          id="device_name"          name="Device          Name"
sourceId="org.bluetooth.characteristic.gap.device_name" uuid="2A00">
<value length="17" type="utf-8" variable_length="false">EFR32 BGM111</value>
<properties>
        <read authenticated="false" bonded="false" encrypted="false"/>
</properties>
</characteristic>
```

In the above example, the value is “**EFR32 BGM111”** and the length is 17 bytes.

Example: Defining both length and value with length **bigger** than the value's length.

```
<!-- Device name -->
<characteristic uuid="2a00">
     <properties read="true" />
     <value type="hex" length="4" variable_length="false">0102</value>
</characteristic>
```

In the example above, the value will be “**01020000”** because the ***length*** is bigger than the value's length and the value gets padded with 0's.

Example: Defining both length and value with length **smaller** than the value's length.

```
<!-- Device name -->
<characteristic uuid="2a00">
     <properties read="true" />
     <value type="hex" length="2" variable_length="false">01020304</value>
</characteristic>
```

In the example above, the value will be “**0102”** because the ***length*** is smaller than the value's length, so the value gets clipped to match the ***length***.

2.5.4 <descriptor>

The XML element <descriptor> can be used to define a generic characteristic descriptor.

Descriptor properties are defined by the <properties> element and only read and/or write access is allowed. Value is defined by <value> element the same way as for characteristics values.

Note: If you manually add a Client Characteristic Configuration Descriptor (UUID: 0x2902), its value will overwrite the *notify*/*indicate* properties of its characteristic. If no CCCD added manually, it will be generated automatically if the characteristic has enabled notification or indication.

Example: Adding a characteristic descriptor with type UUID 2908.

```
<characteristic uuid="2a4d" id="hid_input">
<properties notify="true" read="true" />
<value length="3" />
      <descriptor const="false" discoverable="true" id="" name="Custom Descriptor" sourceId="" uuid="2908">
        <properties>
          <read authenticated="false" bonded="false" encrypted="false"/>
        </properties>
        <value length="0" type="hex" variable_length="false">00</value>
      </descriptor>
</characteristic>
```

2.5.5 <description>

Characteristic user description values are defined with the XML attribute <description>, which must be used inside the <characteristic> XML attribute tags.

Characteristic user description is an optional value. It is exposed to the remote device and can be used, for example, to provide a user-friendly description of the characteristic shown in the application's user interface.

Example: Constant string "Heart Rate Measurement"

```
<characteristic uuid="2a37">
<properties>
    <notify authenticated="false" bonded="false" encrypted="false"/>
</properties>
<description> Heart Rate Measurement </description>
</characteristic>
```

Properties element can be used to allow remote modification of an attribute.

Example: Allow remote reading but require bonding for writing

```
<characteristic uuid="2a37">
    <properties>
        <read authenticated="false" bonded="false" encrypted="true"/>
        <write authenticated="false" bonded="true" encrypted="false"/>
    </properties>
</characteristic>
```

Note: If a description is writable, then the GATT Parser automatically adds the extended properties attribute with *writable_auxiliaries* bit set to be Bluetooth-compliant.

2.5.6 <aggregate>

The XML element <aggregate> enables the creation of an aggregated characteristic format descriptor by automatically converting IDs to attribute handles.

Attribute IDs should refer to characteristic presentation format descriptors.

Example: Adding a characteristic aggregate

```
<characteristic uuid="da8a80c0-829d-498f-b70b-e85c95e0f839">
  <properties notify="true" read="true"/>
  <value length="10" />
  <aggregate>
    <attribute id="format1" />
    <attribute id="format2" />
  </aggregate>
</characteristic>
```

2.6 GATT Examples

Example: A full GAP service with device name and appearance characteristics as constant values with *read* property.

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

<!--Generic Access-->
<service        advertise="false"        name="Generic        Access"        requirement="mandatory"
sourceId="org.bluetooth.service.generic_access" type="primary" uuid="1800">
     <informativeText>Abstract: The generic_access service contains generic information about the device. All
available
       Characteristics are readonly. </informativeText>
    <!--Device Name-->
                   <characteristic      const="true"      id="device_name"      name="Device      Name"
sourceId="org.bluetooth.characteristic.gap.device_name"
     uuid="2A00">
      <informativeText/>
      <value length="17" type="utf-8" variable_length="false">EFR32 BGM111</value>
      <properties>
        <read authenticated="false" bonded="false" encrypted="false"/>
      </properties>
    </characteristic>

    <!--Appearance-->
      <characteristic  const="true"  name="Appearance"  sourceId="org.bluetooth.characteristic.gap.appearance"
uuid="2A01">
      <informativeText>Abstract: The external appearance of this device. The values are composed of a category
(10-bits) and sub-categories (6-bits). </informativeText>
      <value length="2" type="hex" variable_length="false">0000</value>
      <properties>
        <read authenticated="false" bonded="false" encrypted="false"/>
      </properties>
    </characteristic>
  </service>
</gatt>
```

Example: Link Loss and Immediate Alert services.

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

<gatt>
<!--Link Loss-->
        <service    advertise="false"    id="link_loss"    name="Link    Loss"    requirement="mandatory"
sourceId="org.bluetooth.service.link_loss" type="primary" uuid="1803">
    <!--Alert Level-->
                      <characteristic     const="false"     id="alert_level"     name="Alert     Level"
sourceId="org.bluetooth.characteristic.alert_level" uuid="2A06">
      <value length="1" type="hex" variable_length="false"/>
      <properties>
        <read authenticated="false" bonded="false" encrypted="false"/>
        <write authenticated="false" bonded="false" encrypted="false"/>
      </properties>
    </characteristic>
  </service>

  <!--Immediate Alert-->
     <service   advertise="false"   id="immediate_alert"   name="Immediate   Alert"   requirement="mandatory"
sourceId="org.bluetooth.service.immediate_alert" type="primary" uuid="1802">
    <!--Alert Level-->
                      <characteristic     const="false"     id="alert_level"     name="Alert     Level"
sourceId="org.bluetooth.characteristic.alert_level" uuid="2A06">
      <value length="1" type="hex" variable_length="false"/>
      <properties>
        <write_no_response authenticated="false" bonded="false" encrypted="false"/>
      </properties>
    </characteristic>
  </service>

</gatt>
```

Example: GATT database with capabilities

```
<gatt db_name="light_gattdb" out="gatt_db.c" header="gatt_db.h" generic_attribute_service="true">
  <capabilities_declare>
    <capability enable="true">cap_light</capability>
    <capability enable="false">cap_color</capability>
  </capabilities_declare>

  <!--Light Service-->
   <service advertise="false" id="light_service" name="Light Service" requirement="mandatory" sourceId="" 
type="primary" uuid="257f993d-756e-baa6-e69c-8b101e4e6b3f">
    <informativeText>Info about custom service</informativeText>
    <capabilities>
      <capability>cap_light</capability>
      <capability>cap_color</capability>
    </capabilities>

    <!--Ligth Control-->
              <characteristic const="false" id="light_control" name="Ligth Control" sourceId="" 
uuid="85e82a1c-8423-610b-9fea-5ad999445231">
      <description>User description</description>
      <informativeText/>
      <capabilities>
        <capability>cap_light</capability>
      </capabilities>
      <value length="0" type="user" variable_length="false"/>
      <properties>
        <read authenticated="false" bonded="false" encrypted="false"/>
        <write authenticated="false" bonded="false" encrypted="false"/>
        <write_no_response authenticated="false" bonded="false" encrypted="false"/>
        <reliable_write authenticated="false" bonded="false" encrypted="false"/>
        <indicate authenticated="false" bonded="false" encrypted="false"/>
        <notify authenticated="false" bonded="false" encrypted="false"/>
      </properties>
    </characteristic>

    <!--Color control-->
      <characteristic const="false" id="color_control" name="Color control" sourceId="" uuid="16b90591-c54a-
e7c9-413e-a82748a1e783">
      <informativeText/>
      <capabilities>
        <capability>cap_color</capability>
      </capabilities>
      <value length="0" type="user" variable_length="false"/>
      <properties>
        <read authenticated="false" bonded="false" encrypted="false"/>
        <write authenticated="false" bonded="false" encrypted="false"/>
      </properties>
    </characteristic>
  </service>
</gatt>
```

- If the capabilities cap_light and cap_color are enabled, the entire light_service will be visible.
- If the capability cap_light is disabled, the characteristic light_control will be invisible.
- If the capability cap_color is disabled, the characteristic color_control will be invisible.

3. Generating the Code Files

This section of the document describes how to use Simplicity Studio or the `bgbuild` Python script and your IDE to generate the gatt_db.c/h code files from the gatt_configuration.btconf XML file.

3.1 Using Simplicity Studio

When using the GATT Configurator in Simplicity Studio, the code files (gatt_db.c/gatt_db.h) are generated automatically each time the configuration is saved.

3.2 Using bgbuild

The code files can be generated independently from the IDE, using the `bgbuild` Python script provided in the SDK:

`$GSDK_PATH\protocol\bluetooth\bin\gatt\bgbuild.py`

`$GSDK_PATH` is the installation directory of the selected GSDK, for example ~/SimplicityStudio/SDKs/gecko_sdk/.

The script requires installing Python 3 and the Jinja2 package by calling `pip install jinja2`. The script can parse only GATT configurations created with Simplicity Studio 5 / SDK v4.x, or written manually according to this user guide. The GATT Configurator can import older gatt.xml formats.

Mandatory argument:

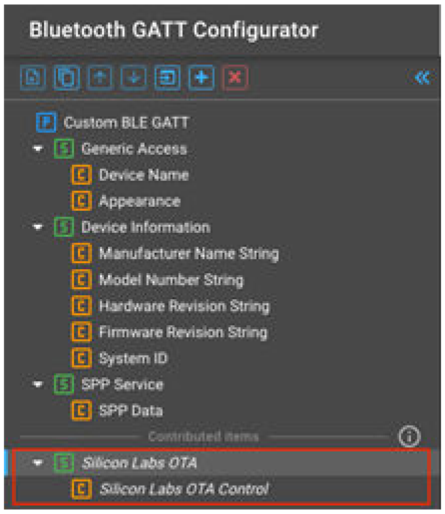
- Path to the GATT XML files, or directories to find the XML files. Separate input with “;”

Optional arguments:

- -h, --help: display help message
- -o OUTDIR, --outdir OUTDIR: the output directory, where the files will be generated (Simplicity Studio 5 generates the sources into the autogen folder. Generating them somewhere else in the project structure could cause potential collisions.)

```
python $GSDK_PATH/protocol/bluetooth/bin/gatt/bgbuild.py
~/SimplicityStudio/v5_workspace/btmesh_soc_empty/config/btconf/gatt_configuration.btconf
-o ~/SimplicityStudio/v5_workspace/btmesh_soc_empty/autogen/
```

In the Bluetooth GATT Configurator, the GATT can contain a **Contributed items** section. These are additional XML files that are non-modifiable from the user interface, but the source generation in Simplicity Studio will include them.



To produce the same result with `bgbuild`, you have to include the whole `config/btconf` folder.

```
python $GSDK_PATH/protocol/bluetooth/bin/gatt/bgbuild.py
~/SimplicityStudio/v5_workspace/btmesh_soc_empty/config/btconf/
-o ~/SimplicityStudio/v5_workspace/btmesh_soc_empty/autogen/
```

4. Revision History

Revision 2.8

October 2023

- In section 3.2 Using bgbuild of Japanese translation only, corrected “pip install jinja22” to “pip install jinja2”.
- In section 3.2 Using bgbuild, updated the process to generate code files manually.

IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

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

www.silabs.com