ChapterÂ 4.Â GATT (Services and Characteristics)

The Generic Attribute Profile (GATT) establishes in detail how to exchange all profile and user data over a BLE connection. In contrast with GAP (ChapterÂ 3), which defines the low-level interactions with devices, GATT deals only with actual data transfer procedures and formats.

GATT also provides the reference framework for all *GATT-based profiles* (discussed in SIG-defined GATT-based profiles), which cover precise use cases and ensure interoperability between devices from different vendors. All standard BLE profiles are therefore based on GATT and must comply with it to operate correctly. This makes GATT a key section of the BLE specification, because every single item of data relevant to applications and users must be formatted, packed, and sent according to its rules.

Roles

As with any other protocol or profile in the Bluetooth specification, GATT starts by defining the roles that interacting devices can adopt:

Client
The GATT client corresponds to the ATT client discussed in Attribute Protocol (ATT). It sends requests to a server and receives responses (and server-initiated updates) from it. The GATT client does not know anything in advance about the serverâs attributes, so it must first inquire about the presence and nature of those attributes by performing service discovery. After completing service discovery, it can then start reading and writing attributes found in the server, as well as receiving server-initiated updates.

Server
The GATT server corresponds to the ATT server discussed in Attribute Protocol (ATT). It receives requests from a client and sends responses back. It also sends server-initiated updates when configured to do so, and it is the role responsible for storing and making the user data available to the client, organized in attributes. Every BLE device sold must include at least a basic GATT server that can respond to client requests,

UUIDs

A universally unique identifier (UUID) is a 128-bit (16 bytes) number that is guaranteed (or has a high probability) to be globally unique. UUIDs are used in many protocols and applications other than Bluetooth, and their format, usage, and generation is specified in ITU-T Rec. X.667, alternatively known as ISO/IEC 9834-8:2005.

For efficiency, and because 16 bytes would take a large chunk of the 27-byte data payload length of the Link Layer, the BLE specification adds two additional UUID formats: 16-bit and 32-bit UUIDs. These shortened formats can be used only with UUIDs that are defined in the Bluetooth specification (i.e., that are listed by the Bluetooth SIG as standard Bluetooth UUIDs).

To reconstruct the full 128-bit UUID from the shortened version, insert the 16- or 32-bit short value (indicated by xxxxxxxx, including leading zeros) into the Bluetooth Base UUID:

```
xxxxxxxx-0000-1000-8000-00805F9B34FB
```

The SIG provides (shortened) UUIDs for all the types, services, and profiles that it defines and specifies. But if your application needs its own, either because the ones offered by the SIG do not cover your requirements or be-

cases, you don't need to use the full 128-bit UUID value every time.

Attributes

Attributes are the smallest data entity defined by GATT (and ATT). They are *addressable* pieces of information that can contain relevant user data (or *metadata*) about the structure and grouping of the different attributes contained within the server. Both GATT and ATT can work only with attributes, so for clients and servers to interact, all information must be organized in this form.

Conceptually, attributes are always located on the server and accessed (and potentially modified) by the client. The specification defines attributes only conceptually, and it does not force the ATT and GATT implementations to use a particular internal storage format or mechanism. Because attributes contain both static definitions of invariable nature and also actual user (often sensor) data that is bound to change rapidly with time (as discussed in Attribute and Data Hierarchy), attributes are usually stored in a mixture of nonvolatile memory and RAM.

Each and every attribute contains information about the attribute itself and then the actual data, in the fields described in the following sections.

Handle

Note

Whenever used in the context of attribute handles, the term *handle range* refers to all attributes with handles contained between two given boundaries. For example, handle range `0x0100-0x010A` would refer to any attribute with a handle between `0x0100` and `0x010A`.

Within a GATT server, the growing values of handles determine the ordered sequence of attributes that a client can access. But gaps between handles are allowed, so a client cannot rely on a contiguous sequence to guess the location of the next attribute. Instead, the client must use the discovery feature (Service and Characteristic Discovery) to obtain the handles of the attributes it is interested in.

Type

The attribute type is nothing other than a UUID (see UUIDs). This can be a 16-, 32-, or 128-bit UUID, taking up 2, 4, or 16 bytes, respectively. The type determines the kind of data present in the value of the attribute, and mechanisms are available to discover attributes based exclusively on their type (see Service and Characteristic Discovery).

pends on the implementation.

Permissions

Permissions are metadata that specify which ATT operations (see ATT operations) can be executed on each particular attribute and with which specific security requirements.

ATT and GATT define the following permissions:

Access Permissions

Similar to file permissions, access permissions determine whether the client can read or write (or both) an *attribute value* (introduced in Value). Each attribute can have one of the following access permissions:

None
The attribute can neither be read nor written by a client.
Readable
The attribute can be read by a client.
Writable
The attribute can be written by a client.

this attribute to be accessed by the client. (See Authentication, Security Modes and Procedures, and Security Modes for more information on authentication and encryption.) These are the allowed encryption permissions, as defined by GATT:

No encryption required (Security Mode 1, Level 1)
The attribute is accessible on a plain-text, non-encrypted connection.
Unauthenticated encryption required (Security Mode 1, Level 2)
The connection must be encrypted to access this attribute, but the encryption keys do not need to be authenticated (although they can be).
Authenticated encryption required (Security Mode 1, Level 3)
The connection must be encrypted with an authenticated key to access this attribute.

Authorization

Determines whether user permission (also known as *authorization*, as discussed in Security Modes and Procedures) is required to access this attribute. An attribute can choose only between requiring or not requiring authorization:

All permissions are independent from each other and can be freely combined by the server, which stores them in a per-attribute basis.

Value

The attribute value holds the actual data content of the attribute. There are no restrictions on the type of data it can contain (you can imagine it as a non-typed buffer that can be cast to whatever the actual type is, based on the attribute type), although its maximum length is limited to 512 bytes by the specification.

As discussed in Attribute and Data Hierarchy, depending on the attribute type, the value can hold additional information about attributes themselves or actual, useful, user-defined application data. This is the part of an attribute that a client can freely access (with the proper permissions permitting) to both read and write. All other entities make up the structure of the attribute and cannot be modified or accessed directly by the client (although the client uses the handle and UUID indirectly in most of the exchanges with the server).

You can think of the whole set of attributes contained in a GATT server as a table (such as TableÂ 4-1), with each row representing a single attribute and each column representing the different parts that actually constitute an attribute.

0x0201	UUID1 (16-bit)	Read only, no security	`0x180A`	2
0x0202	UUID2 (16-bit)	Read only, no security	`0x2A29`	2
0x0215	UUID3 (16-bit)	Read/write, authorization required	âa readable UTF-8 stringâ	23
0x030C	UUID4 (128-bit)	Write only, no security	`{0xFF, 0xFF, 0x00, 0x00}`	4
0x030D	UUID5 (128-bit)	Read/write, authenticated encryption required	`36.43`	8
0x031A	UUID1 (16-bit)	Read only, no security	`0x1801`	2

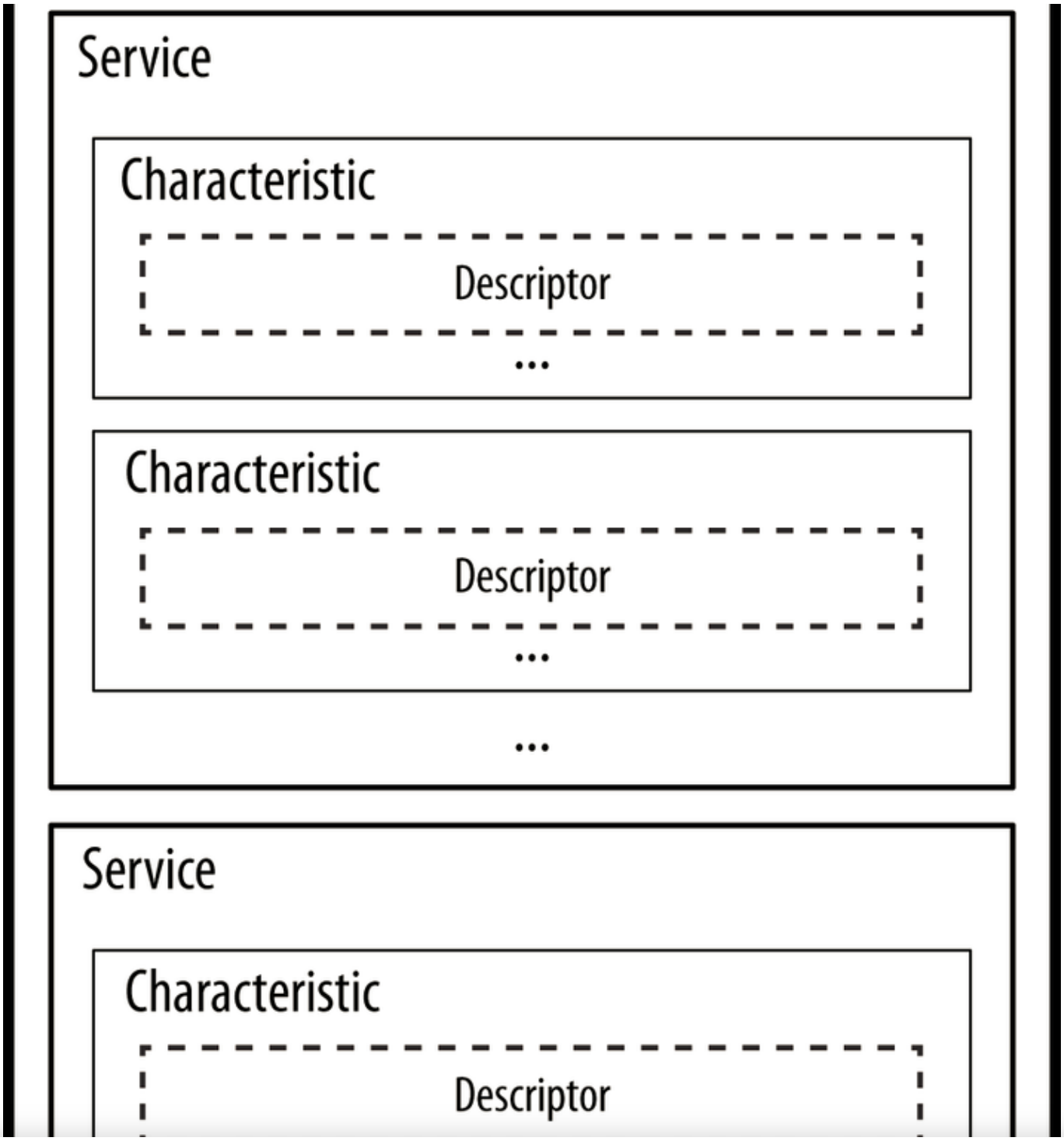
The Value column of the table is intended to mirror the high diversity of formats that attribute values can contain in the different GATT-based profiles. The attributes with handles `0x0201`, `0x0202`, and `0x031A` contain 16-bit integers in their respective value fields. The attribute with handle `0x0215` contains a UTF-8 string, `0x030C` contains a 4-byte buffer, and `0x030D` holds an IEEE-754 64-bit floating point number in its value field.

Attribute and Data Hierarchy

Athough the Bluetooth specification defines attributes in the ATT section, that is as far as ATT goes when it comes to them. ATT operates in attribute terms and relies on all the concepts exposed in Attributes to provide a series of precise protocol data units (PDUs, commonly known as *packets*) that permit a client to access the attributes on a server.

GATT goes further to establish a strict hierarchy to organize attributes in a reusable and practical manner, allowing the access and retrieval of information between client and server to follow a concise set of rules that together consitute the framework used by all GATT-based profiles.

FigureÂ 4-1 illustrates the data hierarchy introduced by GATT.



can contain zero or more *characteristics*. These characteristics, in turn, can include zero or more *descriptors*. This hierarchy is strictly enforced for any device claiming GATT compatibility (essentially, all BLE devices sold), which means that all attributes in a GATT server are included in one of these three categories, with no exceptions. No dangling attributes can live outside of this hierarchy, as exchanging data between BLE devices depends on it.

For most types of data in the GATT hierarchy, it is important to differentiate between their *definition* (the whole group of attributes that make it up) and the *declaration*. The declaration is a single attribute that is always placed first (in increasing handle order) within the definition and that introduces most of the metadata about the data that follows. All declarations have read-only permissions with no security required, because they cannot contain sensitive data. They are only structural attributes that allow the client to find out and discover the layout and nature of the attributes on the server.

Services

GATT services group conceptually related attributes in one common section of the attribute information set in the GATT server. The specification refers to all the attributes within a single service as the *service definition*. Therefore, a GATT serverâs attributes are in fact a succession of service

0xNNNN	UUID*primary service* or UUID*secondary service*	Read Only	Service UUID	2, 4, or 16 bytes

In the declaration shown in TableÂ 4-2, UUIDprimary service (0x2800) and UUIDsecondary service (0x2801) refer to standard, SIG-assigned UUIDs that are used as the exclusive type to introduce a service. They are naturally 16-bit UUIDs (because they are fundamental ones defined by the specfication).

The difference between primary and secondary services is important to note. A *primary service* is the standard type of GATT service that includes relevant, standard functionality exposed by the GATT server. A *secondary service*, on the other hand, is intended to be included only in other primary services and makes sense only as its modifier, having no real meaning on its own. In practice, secondary services are rarely used.

The value of the service declaration attribute itself contains a UUID (as mentioned in Value, the value of an attribute can be any data type), this time corresponding to the UUID of the actual service that this declaration introduces.

Although the service declaration must always be the first attribute of the service, many others can follow it before the next service declaration, usu-

Inside a service definition (that is to say, inside a service), you can add one or more references to another services, using *include definitions*. Include definitions consist of a single attribute (the *include declaration*) that contains all the details required for the client to reference the included service.

Included services can help avoid duplicating data in a GATT server. If a service will be referenced by other services, you can use this mechanism to save memory and simplify the layout of the GATT server. In the previous analogy with classes and objects, you could see include definitions as pointers or references to an existing object instance.

TableÂ 4-3 shows the include declaration attribute with all its fields.

TableÂ 4-3.Â Include Declaration attribute

Handle	Type	Permissions	Value	Value length
`0xNNNN`	UUID*include*	Read only	Included service handle, end group handle, Included Service UUID	6, 8, or 20 bytes

Again, the UUIDinclude (`0x2802`) is a special SIG-assigned UUID used ex-

provides metadata about the actual user data, and the characteristic value (which is a full attribute that contains the user data in its value field).

Additionally, the characteristic value can be followed by descriptors, which further expand on the metadata contained in the characteristic declaration. The declaration, value, and any descriptors together form the *characteristic definition*, which is the bundle of attributes that make up a single characteristic.

TableÂ 4-4 shows the structure of the first two attributes of every single characteristic.

TableÂ 4-4.Â Characteristic declaration and characteristic value attributes

Handle	Type	Permissions	Value	Value length
`0xNNNN`	UUID*characteristic*	Read only	Properties, value handle (`0xMMMM`), characteristic UUID	5, 7, or 19 bytes
`0xMMMM`	Characteristic UUID	Any	Actual value	Variable

is a standardized, unique UUID used exclusively to denote the beginning of characteristics. As with all other declarations (such as service and include), this attribute has read-only permissions, because clients are allowed only to retrieve its value but in no case modify it.

TableÂ 4-5 lists the different items concatenated within the characteristic declarationâs attribute value.

TableÂ 4-5.Â Characteristic declaration attribute value

Name	Length in bytes	Description
Characteristic Properties	1	A bitfield listing the permitted operations on this characteristic
Characteristic Value Handle	2	The handle of the attribute containing the characteristic value
Characteristic UUID	2, 4, or 16	The UUID for this particular characteristic

These three fields are contained in a characteristic declaration attribute

that can be used with this characteristic. Each of those 10 properties is encoded as a single bit on the bitfield shown in TableÂ 4-6.

TableÂ 4-6.Â Characteristic properties

Property	Location	Description
Broadcast	Properties	If set, allows this characteristic value to be placed in advertising packets, using the Service Data AD Type (see)
Read	Properties	If set, allows clients to read this characteristic using any of the ATT read operations listed in
Write without response	Properties	If set, allows clients to use the Write Command ATT operation on this characteristic (see)
Write	Properties	If set, allows clients to use the Write Request/Response ATT operation on this

		Value Notification ATT operation on this characteristic (see)
Indicate	Properties	If set, allows the server to use the Handle Value Indication/Confirmation ATT operation on this characteristic (see)
Signed Write Command	Properties	If set, allows clients to use the Signed Write Command ATT operation on this characteristic (see)
Queued Write	Extended Properties	If set, allows clients to use the Queued Writes ATT operations on this characteristic (see)
Writable Auxiliaries	Extended Properties	If set, a client can write to the descriptor described in

The client can read those properties to find out which operations it is al-

These two bytes contain the attribute handle of the attribute that contains the actual value of the characteristic. Although itâs often the case, you should never assume that this handle will be contiguous (i.e., `0xNNNN+1`) to the one containing the declaration.

Characteristic UUID

The UUID of the particular characteristic, this can be either a SIG-approved UUID (when making use of the dozens of characteristic types included in the standard profiles) or a 128-bit vendor specific UUID otherwise.

Continuing with the class and object-orientation analogy, characteristics are like individual fields or properties in that class, and a profile is like an application that makes use of one or more classes for a specific need or purpose.

Characteristic value attribute

Finally, the characteristic value attribute contains the actual user data that the client can read from and write to for practical information exchanges. The type for this attribute is always the same UUID found in the characteristicâs declaration value field (as shown in Characteristic declaration attribute). So, characteristic value attributes no longer have types of *services*

Characteristic Descriptors

GATT characteristic descriptors (commonly called simply *descriptors*) are mostly used to provide the client with *metadata* (additional information about the characteristic and its value). They are always placed within the characteristic definition and after the characteristic value attribute. Descriptors are always made of a single attribute, the *characteristic descriptor declaration*, whose UUID is always the descriptor type and whose value contains whatever is defined by that particular descriptor type.

You can find two types of descriptors in the different GATT characteristics:

GATT-defined descriptors
: These are the fundamental, widely used descriptor types that simply add meta information about the characteristic. The following sections describe the most common ones.

Profile or vendor-defined descriptors
: Regardless of whether a profile is specified and published by the SIG or by a particular vendor, these descriptors can contain all types of data, including additional information regarding the characteristic value, such as the encoding used to acquire the value from a sensor or any other particulars that complement the reading itself.

bits, covered in Characteristic declaration attribute and listed in TableÂ 4-6.

Characteristic User Description Descriptor

As the name implies, this descriptor contains a user-readable description for the characteristic within which it is placed. This is a UTF-8 string that could read, for example, âTemperature in the living room.â

Client Characteristic Configuration Descriptor

This descriptor type (often abbreviated CCCD) is without a doubt the most important and commonly used, and it is essential for the operation of most of the profiles and use cases. Its function is simple: it acts as a switch, enabling or disabling server-initiated updates (covered in more detail in Server-Initiated Updates), but only for the characteristic in which it finds itself enclosed.

> **Why Provide a Notification Switch?**
>
> As discussed earlier, a client knows nothing in advance about a serverâs attributes, so it will need to perform discovery to find out which services, characteristics, and descriptors are present on the server. The server sends server-initiated updates (handle value notifications and handle value indications, introduced in ATT operations) asynchronously whenever a characteristicâs value changes, formatted in a packet that only contains an attribute handle and an array of bytes with the value.

A CCCDâs value is nothing more than a two-bit bitfield, with one bit corresponding to notifications and the other to indications. A client can set and clear those bits at any time, and the server will check them every time the characteristic that encloses them has changed value and might be susceptible to an update over the air.

Every time a client wants to enable notifications or indications for a particular characteristic that supports them, it simply uses a Write Request ATT packet to set the corresponding bit to 1. The server will then reply with a Write Response and start sending the appropriate packets whenever it wants to alert the client of a change in value.

Additionally, CCCDs have two special properties that separate them from other attributes:

Their values are unique per connection
In *multi-connection* scenarios, in which a central is connected to multiple peripherals and also acting as a GATT server, each peripheral will receive its own copy of the CCCDâs value when reading it with ATT.
Their values are preserved across connections with bonded devices

regardless of the time elapsed between connections.

Many protocol stacks have special mechanisms to deal with CCCDs, both from a clientâs and a serverâs perspective, because they are so critical to correct operation and to guarantee timely data updates between peers.

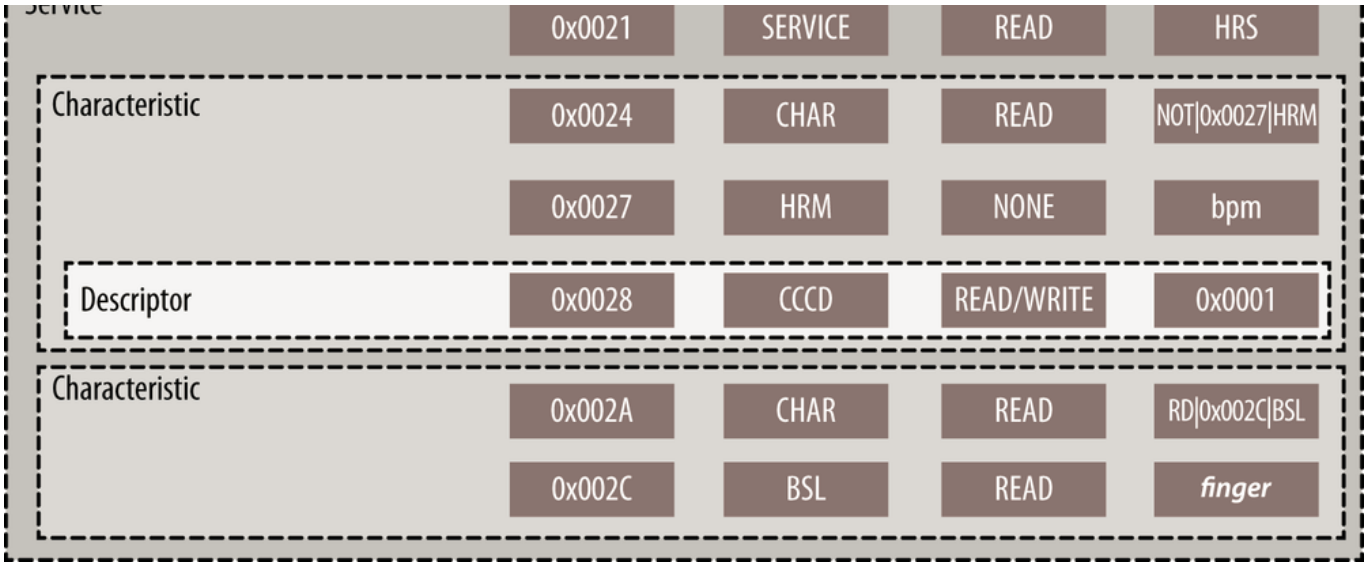
Characteristic presentation format descriptor

When present, this descriptor type contains the actual format of the enclosing characteristic value in its seven-byte attribute value. The list of formats available include Booleans, strings, integers, floating-point, and even generic untyped buffers.

Example Service

This section presents an example of a particular service found in many commercial products today. The Heart Rate Service (HRS) exposes the userâs heart rate to a monitoring device.

FigureÂ 4-2 illustrates an instance of the HRS on a fictitious server. This would not be the only service contained in the server, so you can see this as a partial slice of the complete set of attributes that a client could access.



FigureÂ 4-2.Â GATT Heart Rate Service

Here is a handle-by-handle description of the HRS service illustrated in FigureÂ 4-2:

Handle `0x0021`

This attribute contains the service declaration (see Services) for the Heart Rate Service. These are the service declaration attributeâs fields:

UUID
The UUID is the standard 16-bit UUID for a primary service declaration, UUIDprimary service (`0x2800`).
Value

ration attributeâs fields:

UUID
The UUID is the standard 16-bit UUID for a characteristic declaration, UUIDcharacteristic (`0x2803`).
Value
The characteristic properties for this characteristic are notify only, the characteristic value handle is `0x0027`, and the characteristic value UUID is the UUID for Heart Rate Measurement (`0x2A37`).

Handle `0x0027`

This attribute contains the characteristic value (see Characteristic value attribute), in this case the heart rate measurement itself. These are the characteristic value attributeâs fields:

UUID
The same UUID present in the last two bytes of the characteristic definitionâs attribute value.
Permissions

Handle `0x0028`

This attribute contains a CCCD (a key descriptor described in Client Characteristic Configuration Descriptor). These are the CCCD attributeâs fields:

UUID
The UUID for any CCCD is always the standard 16-bit UUIDCCCD (`0x2902`).
Permissions
A CCCD must always be readable and writable. The security level required to execute these operations is defined by the profile or application.
Value
As already established, the CCCDâs value is a bitfield, in this case `0x0001`, denoting that notifications are enabled for this particular HRM characteristic.

Handle `0x002A`

This attribute contains another characteristic declaration (see Characteristic declaration attribute), this time for Body Sensor Location characteristic. These are the characteristic declaration

The characteristic properties for this characteristic are read only, the characteristic value handle is `0x002C`, and the characteristic value UUID is the UUID for the Body Sensor Location (`0x2A38`).

Handle `0x002C`

This attribute contains the characteristic value (see Characteristic value attribute), in this case the body sensor location. These are the characteristic value attributeâs fields:

UUID
The same UUID present in the last two bytes of the characteristics definitionâs attribute value.
Permissions
This attributeâs value is read only: a client can only check where the sensor is located, but not modify its location (that is up to the server).
Value
The actual body sensor location (fictitiously represented as âfingerâ for clarity).

Several tools exist to discover and display the different services on a server in a format similar to FigureÂ 4-2, which can be useful during appli-

Attribute Caching

Attributes discusses how attribute handles allow a client to individually address all available attributes on a server. Discovering the list of available handles and the contents of their respective attributes can be a time-consuming (and power-consuming) process detailed further in Service and Characteristic Discovery. But for now, this section describes what a client can do to avoid performing the discovery procedure every time it reconnects to a server, and under what circumstances it can do so.

Servers typically tend to maintain a stable set of attributes, and their basic structure does not, in most cases, change over the lifetime of a server device. But the implementation imposes no rigid restrictions in this regard, and a server is indeed free to completely overhaul its attributes and even replace them with a radically new set at any time (through a firmware update or perhaps with the installation of applications on the server device, for example). Certain rules and constraints are therefore required, so that a client can rely on the validity of previously discovered handles without risk of them having changed on the server and thus no longer being valid.

As a general rule, the specification recommends that clients cache (i.e., store for subsequent transactions and even connections) the handles of the attributes they are interested in. Attribute values, especially in the cases where they correspond to actual user data, are highly volatile, so it

Clients can ascertain if the result of discovery can be cached for future use by observing the following conditions:

> No Service Changed characteristic present on the server
> Clients can freely and permamently cache all handles found with no restrictions. The server guarantees they will not change during the lifetime of the device.
> Service Changed characteristic present on the server
> In this case, a client needs to subscribe to the server-initiated updates by writing into the corresponding CCCD enclosed within the Service Changed characteristic (see Characteristic Descriptors). This will allow the server to alert the client of any structural changes. If the client and the server are bonded as described in Security Modes and Procedures, the client can cache attribute handles across connections and expect them to remain identical. If the devices are not bonded, the client will need to perform discovery every time it reconnects to the server.

GATT Service provides more details about using the Service Changed characteristic.

GATT Attribute Data in Advertising Packets

TableÂ 3-3 discusses the Service Data AD Type, but that section does not describe the format it uses to enclose server attributes inside an advertising packet. The Core Specification Supplement in the Specification Adopted Documents page specifies the fields that the GATT server must insert in the payload of an advertising packet to make a particular serviceâs data available to scanners.

As shown in TableÂ 4-7, to be able to broadcast service data, a GATT server must include two different fields in the advertising packetâs Service Data section.

TableÂ 4-7.Â Service Data AD Type

Field	Length in bytes	Description
UUID	2, 4, or 16	The actual UUID identifying the data
Service Data	Variable	The data associated with the service identified by the UUID

The contents of the Service Data field can correspond to the complete or partial value of a particular characteristic or descriptor within the corre-

exchanges to take place. They are all based on the different operations that ATT provides (introduced in ATT operations).

To a certain extent, most of the features listed in this chapter are exposed in one way or another in most GATT APIs. GATT server APIs add the ability to populate the actual server with attributes, but that is heavily implementation dependant and beyond the scope of this chapter.

Exchange MTU

This succinct two-packet procedure allows each ATT peer to let the other end know about the maximum transmission unit (MTU, or effectively maximum packet length) it can hold in its buffers and can therefore accept.

This procedure is used only whenever either the client or the server (or both) can handle MTUs longer than the default `ATT_MTU` of 23 bytes (see Logical Link Control and Adaptation Protocol (L2CAP)) and wants to inform the other end that it can send packets longer than the default values that the specification requires. L2CAP will then fragment these bigger packets into small Link Layers packets and recombine them from small Link Layers packets.

Service and Characteristic Discovery

As mentioned elsewhere in this chapter, the client has no knowledge

For *primary service discovery*, GATT offers the two following options:

Discover all primary services
Using this feature, clients can retrieve a *full list* of all primary services (regardless of service UUIDs) from the remote server. This is commonly used when the client supports more than one service and therefore wants to find out about the full service support on the server side. Because the client can specify a handle range when issuing the required request, it must set `0x0001-0xFFFF` as the handle range to implement this feature, covering the full attribute range of the server.
Discover primary service by service UUID
Whenever the client knows which service it is looking for (usually because it supports only that single service itself), it can simply look for *all instances* of a particular service using this feature, also with the requirement of setting the handle range to `0x0001-0xFFFF`.

Each of these procedures yield handle ranges that refer to the attributes that belong to a single service. The *discover all primary services* feature also obtains the individual service UUIDs.

When the client has already found services on the server, it can proceed to

query refers to the boundaries of an existing service, previously obtained using service discovery. As with service discovery, the client also receives a set of handle ranges, along with UUIDs when applicable.

In terms of *characteristic discovery*, GATT offers the following options:

Discover all characteristics of a service
Once a client has obtained the handle range for a service it might be interested in, it can then proceed to retrieve a *full list* of its characteristics. The only input is the handle range, and in exchange, the server returns both the handle and the value of all characteristic declaration attributes enclosed within that service (see Characteristic declaration attribute).
Discover characteristics by UUID
This procedure is identical to the previous one, except the client discards all responses that do not match the particular characteristic UUID it targets.

Once the boundaries (in terms of handles) of a target characteristic have been established, the client can go on to *characteristic descriptor discovery*:

Discover all characteristic descriptors

All the features in this section can be performed over open, unsecured connections, because discovery is allowed for all clients without any restrictions.

Reading Characteristics and Descriptors

To obtain the current value of a characteristic value or a descriptor, the client has the following choices:

Read characteristic value or descriptor
This feature can be used to simply read the contents of a characteristic value or a descriptor by using its handle. Only the first `ATT_MTU-1` bytes of the contents can be read, because that is the maximum number of bytes that can fit in the response packet (1 byte is reserved for the ATT operation code).
Read long characteristic value or descriptor
If the value is too long to be read with the previous feature, this feature includes an offset along with the handle in the request, so that the characteristic value or the descriptor contents can be read in successive chunks. Multiple request/response pairs might be required, depending on the length of the attribute value being read.

all the characteristics of a specific type. The client simply provides a handle range and a UUID and receives an array of values of characteristics enclosed in that range.

Read mutiple characteristic values

Conversely, if a client already has the handles for a set of characteristics it wants to obtain the value from, it can then send a request with this set of handles and subsequently receive the values for all corresponding characteristics.

Reading characteristics and descriptors is subject to the security permissions and the server can deny permission if the connectionâs security level does not match the established requirements (see Security).

Writing Characteristics and Descriptors

To write to the value of a characteristic value or a descriptor, the client has the following choices:

Write characteristic value or descriptor

This feature is used to write to a characteristic value or descriptor. The client provides a handle and the contents of the value (up to `ATT_MTU-3` bytes, because the handle and the ATT operation code are included in the packet with the data) and the server will acknowledge the write operation with a response

includes an offset and the data itself, and then finally writing them all atomically with an *execute write* operation.

Additionally, and only applicable for characteristic values, these features are available:

Write without response
This feature is the converse equivalent of notifications (detailed in Server-Initiated Updates) and uses Write Command packets. Write Commands are unacknowledged packets that include a handle and a value, and they can be sent at any time and in any amount without any flow control mechanism kicking in (except of course for the native Link Layer flow control, since all traffic is subject to it). The server is free to silently discard them if it cannot process them or if the attribute permissions prevent it from accepting it. The client will never know, but that is by mutual agreement. The only way for the client to find out whether the value was written is to read it after the fact.
Reliable writes
Similar to the *read multiple characteristic values* feature, when a client wants to queue write operations to multiple characteristic values, it issues a final packet to commit the pending write operations and execute them.

to a clientâs request) packets that can flow from the server to the client. These updates send timely alerts of changes in a characteristic value without the client having to regularly poll for them, saving both power and bandwidth.

There are two types of server-initiated updates:

Characteristic Value Notification
Notifications are packets that include the handle of a characteristic value attribute along with its current value. The client receives them and can choose to act upon them, but it sends no acknowledgement back to the server to confirm reception. Along with *write without response*, this is the only other packet that does not comply with the standard request/response flow control mechanism in ATT, as the server can send any number of these notifications at any time. This feature uses the *handle value notification* (HVN) ATT packet.

Characteristic Value Indication
Indications, on the other hand, follow the same handle/value format but require an explicit acknowledgment from the client in the form of a *confirmation*. Note that although the server cannot send further indications (even for different characteristics) until it receives confirmation from the client

The client must enable both types of server-initiated updates by writing to the corresponding CCCD before the server can start sending them. Client Characteristic Configuration Descriptor describes this process extensively.

Security

Security Modes and Procedures discusses how the GAP authentication procedure can be used to transition from one security mode to a higher, more secure one by using the different means available within the Security Manager and GAP. GATT transactions can act as triggers of such an authentication procedure. As discussed in Permissions, each attribute within a GATT server has fine-grained, independent permissions for both reading and writing, and these permissions are enforced at the ATT level.

Generally speaking, attributes that are *declarations* require no special security to be accessed. This is true for both service and characteristic declarations, but not for descriptor declarations, which contain the relevant data directly in them, rather than in a separate attribute. This is done so that clients that have not yet paired or bonded with a server can at least perform basic service and characteristic discovery, without having to resort to performing security procedures. The attribute layout and data hierarchy of a server is not considered to be sensitive information and is therefore freely available to all clients.

When accessing a characteristic value or a descriptor declaration (also

Denotes that the link is not encrypted and that the server does not have a long-term key (LTK, first introduced in Security Keys) available to encrypt the link, or that the link is indeed encrypted, but the LTK used to perform the encryption procedure is not authenticated (generated with man-in-the-middle protection; see Authentication) while the permissions required authenticated encryption.

Insufficient Encryption

Denotes that the link is not encrypted but a suitable LTK is available.

GAP and GATT roles are not linked in any way and can be mixed and matched freely, but security procedures are always initiated by the GAP central (see Security Manager (SM)). Therefore, depending on which peer is acting as a central and which as a peripheral, it can be up to either the GATT client or the GATT server to initiate the pairing, bonding, or encryption procedure in order to raise the security level of the connection. Once the security level matches the one required by the attributeâs permissions, the client can send the request again to be executed on the server.

GATT Service

Just as GAP has its own SIG-specified service that is mandatory for all de-

delimits a particular area of attributes in the server. This is the area that has been affected by structural changes and needs to be rediscovered by the client. The client will have to perform service and characteristic discovery in that area, because the attributes it can have cached might no longer be valid.

TableÂ 4-8.Â Service changed characteristic value

Handle	Type	Permissions	Value	Value length
`0xNNNN`	UUID*service changed*	None	Affected handle range	4

A client must enable indications on the corresponding CCCD for this characteristic before doing anything else, so that it can become aware of any changes on the serverâs attribute structure.

If the server suffers a structural change in attribute layout, it will then immediately send a handle value indication to the client and wait for the corresponding confirmation. In this way, it can be sure that the client understands that cached attribute handles in that range might no longer be valid. If the attributes change outside the lifetime of a connection with a bonded device, the server will send the indication right after the connec-