CORE Bluetooth® Communication

Implementation Notes

V2.1.2

Revision Number	Date	Comments
V1.0	2020-12-06	Description of the Health Thermometer Service
V1.1	2021-04-20	Adding Battery Profile (available since CORE FW 0.3.17)
V2.0	2021-12-10	Adding Core Body Temperature Service (available since CORE FW 0.6.1)
V2.1	2022-10-10	Add section about scanning for CORE and identifying the peripheral, Add sample packet for Core Temperature characteristic

CORE is a venture of greenTEG AG.

Hofwisenstr. 50A
8153 Rümlang, Switzerland
T: +41 44 515 09 15
info@CoreBodyTemp.com
CoreBodyTemp.com

1 Overview

CORE is an easy-to-use wearable product that can accurately monitor core body temperature[1]. To make it easy for developers that want to interact with the CORE in their own Bluetooth applications, we implemented Bluetooth low energy (BLE) GATT servers on the CORE. The Generic Attribute (GATT)[2] profile is implemented in accordance with the Bluetooth Core Specifications[3]. The GATT profile specifies the structure in which the profile data is exchanged: data is encapsulated in *services*, which consist of one or more *characteristics*.

CORE implements a custom GATT Service (CoreTemp Service) as well as some Bluetooth SIG adopted profiles and services. To get the live core body temperature along with meta information such as skin temperature, the **Core Body Temperature Service (CTS)** should be used. It features a control point characteristic that allows to manage external heart rate monitor pairing.

If only the core temperature is needed, the Bluetooth SIG-adopted **Health Thermometer Profile (HTP)** is the preferred communication method. CORE is compliant to the HTP by implementing both the **Health Thermometer Service (HTS)** and the **Device Information Service (DIS)**. Finally, we encourage reading the battery level through the **Battery Service (BAS)** to make sure the user knows when to charge his device.

Any Bluetooth GATT client can connect to the GATT servers running on the CORE. Hence it is possible to integrate the live core body temperature values into your own app.

In the following, an overview of the BLE services and characteristics that are running on CORE are presented. We also provide some sample implementations and guidelines in the different appendixes.

[1] https://corebodytemp.com/pages/core-technology
[2] Check out one of the various online tutorials such as https://devzone.nordicsemi.com/nordic/short-range-guides/b/bluetooth-low-energy/posts/ble-services-a-beginners-tutorial.
[3] https://www.bluetooth.com/specifications/specs/core-specification/

2 Scanning for CORE and identification

Before any data exchange between CORE and the client device (e. g. a smartphone), the client needs to initiate a scan for Bluetooth devices and the program running must identify the CORE sensor among all peripherals found during the scan. In the following, the CORE sensor advertisement data structure is explained.

Like all Bluetooth Low Energy devices, CORE adheres to the Generic Access Protocol (GAP) for discovery and connection. The discovery process of a GAP device involves 'advertising': The GAP peripheral emits small packets containing data useful to the scanning device.

An advertising packet has space for 31 octets of data which consists of fields (AD Types) and their values. Most GAP peripherals use the 'service UUID AD Types' (e. g. 0x06) to declare a list of UUIDs of the important GATT services the device supports. Alongside, the custom 'Manufacturer Specific Data' field can be used to advertise anything of importance to the GAP client.

In addition to the advertisement packet payload, more information can be sent in the Scan Response PDU ('Protocol Data Unit'). Only during a BLE 'Active Scan', this additional packet will be sent back from the peripheral device as a response to a 'Scan Request GAP protocol PDU' packet. The active scan response PDU has also a max size of 31 octets.

The CORE sensor is optimized to act as a peripheral in permanent connection, but it can also be used as a beacon serving core body temperature. To fit enough information, the CORE sensor supports active scan requests and provides an active scan response upon request.

- **Advertisement packet of CORE**

AD_TYPE	*AD Type name*	*Example value (LSB first)*	*Description*
0X01	**Flags**	0x06	Flags - LE General Discoverable Mode
0x03	**Complete list of 16bit UUIDs**	0x0918	CORE runs a GATT server for the Health Thermometer service
0X09	**Complete Local Name**	0x434F5245	"CORE"
0xFF	**Manufacturer Specific Data**	0x0B-F6-00-04-B3-91	see below

Example: parsing manufacturer-specific data payload 0x0B-F6-00-04-B3-91

	LSO			MSO
Payload	*Manufacturer ID*	*Version of Manufacturer Data*	*Status**	*Beacon value***
Data Type	-	Byte 4	8bit	UINT16
Size	16bit	8bit	8bit	16 bit
Binary	0xF60B	0x00	0x04	0x91B3
Value	*(-)*	Version "0"	"4"	37.299°C

*Status flag:

- Bit 0-3: specifies state of the sensor state machine, normally **4 – Measuring or Erasing**, other states are internal
- Bit 4-7: reserved or internal use

**Beacon value: current core body temperature as unsigned 16bit integer in units 0.001°C, so here 37.001 °C.

- **Active scan response of CORE**

AD_TYPE	AD Type name	Example value (LSB first)	Description
0x03	**Complete list of 16bit UUIDs**	0x0A180F18	Device Information Service (0x180A) Battery Service (0x180F)
0x07	**List of 128bit services**	0x21E1DA14B5977CB047 431E5B00210000	Core Temperature Service (0x00002100-5B1E-4347-B07C-97B514DAE121)

Important Note on the intended way to identify CORE as a BLE peripheral

The intended way to identify a CORE sensor is to filter for devices that advertise the 128bit UUID of the **Core Temp service.**

128bit UUID: 00002100-5B1E-4347-B07C-97B514DAE121 – “Core Temp Service”

In legacy firmwares, a different 128bit UUID was advertised instead of the Core Temp Service, the **CORE private API service**

128bit UUID: 00004200-F366-40B2-AC37-70CCE0AA83B1 – “CORE private API service”

The recommended way is to filter for peripherals that advertise either of the two services for backwards compatibility.

3 Bluetooth GATT profiles of CORE

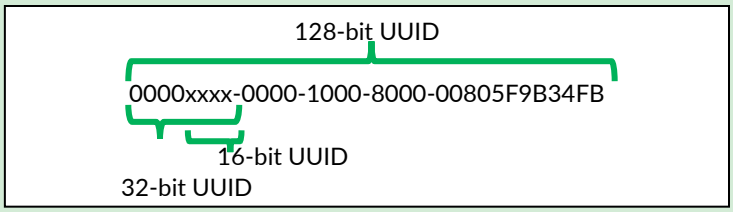
The standard services are in accordance with the Bluetooth SIG specifications. The latest specifications can be found online under https://www.bluetooth.com/specifications/specs/. Core Body Temperature Service is a custom specification issued by CORE and it is available on www.github.com/CoreBodyTemp/CoreBodyTemp.

An Overview of the services and characteristics that are implemented by CORE can be found in Figure 1.

CORE BLE Profile	Core Body Temperature Service 128bit-uuid: 00002100-5B1E-4347-B07C-97B514DAE121	Core Body Temperature Characteristic
		Control Point Characteristic
	Health Thermometer Service 16bit-uuid 0x1809	Temperature Characteristic
	Device Information Service 16bit-uuid 0x180A	Manufacturer Name
		Model Number
		...
	Battery Service 16bit-uuid 0x180F	Battery Value

Figure 1 Overview of the Services and Characteristics in the CORE Bluetooth GATT profile.

A note on UUIDs, the identification numbers of BLE services:



Each attribute of the GATT profile (services, characteristics, and descriptors) is assigned a unique number to identify them. To save transmitting airtime and memory space, short 16-bit UUIDs (uuid16) are predefined by the Bluetooth SIG (and reserved for future use, also 32-bit UUIDs might be used at some point). For example, the uuid16 for the health thermometer service is 0x1809. All current uuid16 numbers can be found in the *"16-bit UUID Numbers Document"*, which is accessible here: https://www.bluetooth.com/specifications/assigned-numbers/.

The 16-bit UUID are then directly transmitted over the air, so that a Bluetooth collector can detect which services etc. a device is advertising.
The second type of UUID is 128 bit long (uuid128), and sometimes referred to as a vendor specific UUID. It is a unique number used to transmit custom/proprietary Bluetooth services, e.g. the CoreTemp Service.
Every short UUID can be transformed into a long UUID. To determine the long UUID (uuid128) of e. g. a GATT service, the short 16bit-UUIDs are inserted into the base UUID (see above). More on service discovery can be found here: https://www.bluetooth.com/specifications/assigned-numbers/service-discovery/.

3.1 Core Body Temperature Service

This is the custom GATT service we created to leverage the full potential of the CORE device via a public Bluetooth Low Energy GATT service. Because it is a custom service, it does not feature a short UUID16 to be identified. Instead, the service UUID is set to

UUID128 00002100-5B1E-4347-B07C-97B514DAE121.

There are two characteristics that are exposed in the Core Body Temperature Service:

- **Core Body Temperature**

UUID128	00002101-5B1E-4347-B07C-97B514DAE121
payload	The **payload** of the Core Body Temperature characteristic, contains (depending on the flag field, not all values may be present): LSO MSO **Payload** | *Flag* | *Core Body Temperature* | *Skin Temperature* | *Core Reserved* | *Quality & State* **Data Type** | 8bit | SINT16 | *SINT16* | SINT16 | 8bit **Size** | 8bit | 16bit | 16bit | 16bit | 8 bit
Notes	Notifications on this characteristic can be activated by setting the corresponding CCCD value. See the full specification for details.

Example: parsing Core Body Temperature characteristic data payload 0x07-92-0E-C2-0D-CE-00-14

	LSO				MSO
Payload	*Flag*	*Core Body Temperature*	*Skin Temperature*	*Core Reserved*	*Quality & State*
Binary	0x07	0x0E92	0x0DC2	0x00CE	0x14
Value	*all fields are present, Temperature Unit is °C*	37.30°C	35.22°C	206	Quality 'excellent' Heart Rate supported, not receiving signal

- **CoreTemp Control Point**

UUID128	00002102-5B1E-4347-B07C-97B514DAE121
payload	The **payload** of the Core Body Temperature characteristic, contains (depending on the flag field, not all values may be present): LSO MSO **Payload** | *OpCode* | *Parameter Value* **Data Type** | 8bit | *Variable, e. g. an ANT+ id to pair a heart rate monitor.* **Size** | 8bit | *Variable*
Notes	Indications on this characteristic must be activated by setting the corresponding CCCD value. By writing the respective OpCode to the Control Point characteristic, the procedure is started on the server and then after completion / failure, it is indicated back to the client using OpCode 0x80. This Characteristic is intended to use for the management of heart rate monitors that are paired to the CORE device to improve performance. See the full specification for details.

3.2 Health Thermometer Service – UUID16 0x1809

The CORE device implements two of the health thermometer service characteristics:

- **Temperature Measurement characteristic**

<table>
<tr><td>UUID16</td><td>0x2a1c</td></tr>
<tr><td>payload</td><td>The 5-byte payload of the Temperature Measurement characteristic consists, according to the characteristic specification, of:

<table>
<tr><td>Payload</td><td>Flag</td><td colspan="4">32bit IEEE 11073-20601 float[4] (Temperature Value)</td></tr>
<tr><td>Detail</td><td></td><td>Exponent</td><td colspan="3">Mantissa</td></tr>
<tr><td>Size</td><td>8bit</td><td>8bit</td><td>8 bit</td><td>8 bit</td><td>8 bit</td></tr>
<tr><td></td><td>MSB←</td><td></td><td></td><td></td><td>→ LSB</td></tr>
</table>

Flag:
• Bit 0: specifies unit of the temperature (0=”Celsius”).
• Bit 1: specifies presence of timestamp field (0 for CORE)
• Bit 2: specifies presence of temperature type field. (1 for CORE)</td></tr>
<tr><td>Notes</td><td>Ready-to use sample code that parse the IEEE 11073 32bit float format can be found e. g. here:

• sample converter written in in C: https://github.com/signove/antidote/blob/master/src/util/bytelib.c#L240-L268
• sample converter written in kotlin: https://github.com/SiliconLabs/EFRConnect-android/blob/e986690414c885207e23d5224e7dd2ab03d1a0d9/mobile/src/main/java/com/siliconlabs/bledemo/HealthThermometer/Models/TemperatureReading.kt

Both are third party source example and should be used with care and with no warranty, it has not been tested or approved in any way.

The IEEE-11073 special value for NaN (0x007FFFFF) is transmitted if the current temperature reading is invalid (e. g. if CORE is “off body”).

Setting up a server-initiated notification on the core body temperature can be done through the client characteristic descriptor (CCCD 0x2902). Currently, the notification interval is set to 10 seconds (CORE firmware: 0.3.19).</td></tr>
</table>

- **Temperature Type characteristic**

UUID16	0x2a1d
payload	Value for body position, set to 0x02 (general) for CORE.

3.3 Battery – UUID16 0x180F

- **Battery Level characteristic**

UUID16	0x2a19
payload	8bit Unsigned Int corresponding to the battery level in percent (*range:0..100)*

[4] Also commonly referred to as the *medical device numeric format (MDNF)*. Specification can be found here, for example: https://www.bluetooth.com/wp-content/uploads/2019/03/PHD_Transcoding_WP_v16.pdf

4 Other Services

- Generic Access Service and Generic Attribute Service

UUID16	0x1800 – Generic Access
	Characteristics: 2a00 – Device Name (CORE [x] with x set to the device state a, m, etc5) 2a01 – Appearance 2a04 - Peripheral Preferred Connection Parameters 2a06 - Alert Level
UUID16	0x1801 – Generic Attribute
	0x2a05 – Service Changed characteristic. Used for the proprietary service (0x4200) This is an optional characteristic that can be included in GATT servers and it communicates to the client any potential changes in the contents of its attribute information.
UUID16	0x4200 – Proprietary Bluetooth service of CORE (UUID128: 00004200-f366-40b2-ac37-70cce0aa83b1) with characteristics.

5 Testing the BLE communications

Important Notes:

To test your CORE device, first connect with the CORE smart phone app and ensure the CORE is on. The CORE app can be found on your phone's corresponding online store[6][7][8]. For support, please visit https://corebodytemp.com/.

Other apps can be useful to help with implementing a Bluetooth client. Those toolbox-apps are presented in Appendix B.

While the phone apps provide the full functionality such as displaying live and historic data of your core body temperature, updating the firmware etc, the Android wearOS app only reads from the standard Bluetooth profiles implemented on the CORE. Thus, it can be used as a reference implementation for your own app. Some excerpts from the source code can be found in Appendix C.

Trouble Shooting:

Problem	**Possible Resoultion**
Core is not found when scanning for Bluetooth devices	Make sure you are not connected to your CORE with a different Bluetooth client. Make sure Bluetooth is not restricted on your device (e. g. battery saving mode)
(Android) Notification setup not working for the temperature measurement characteristic	Make sure any bluetooth command is answered by the device, i. e. the callback is received, before trying to write the Gatt descriptor (see https://stackoverflow.com/questions/52598927/read-and-notify-ble-android)

Copy chapter 3 from CORE Wireless Connectivity Guide.docx

[5] Depending on the state, the core body temperature measurement may be put on hold. Check with the greenTEG CORE app whether the core body temperature is being updated, if you are not sure.
[6] Android https://play.google.com/store/apps/details?id=com.greenteg.core.app
[7] Android wear https://play.google.com/store/apps/details?id=com.greenteg.core.wearos
[8] iOS https://apps.apple.com/us/app/id1521866309

6 Glossary

Term	Requirement
BLE	Bluetooth Low Energy
LSO	Least significant octet
HRM	Heart Rate Monitor
MSO	Most significant octet
UUID	Universally Unique Identifier
Server	*Also:* The Peripheral. In the context of this Specification: The core body temp device that sends data to the Client.
Client	*Also:* The Central. In the context of this Specification: The display device such as mobile phone app or a smartwatch.

7 Further Resources:

Note: The Specification for the ***Core Body Temperature Service*** is available upon request.

Appendix A:– Excerpts Chapter 3 from the Health Thermometer Service document

Appendix B– General Android Apps illustrating Bluetooth communication.

Appendix C– Excerpts from CoreBodyTemp's sample wearOS app

Appendix D– Excerpts from David Vassallo's Blog *"BLE Health Devices: First Steps with Android"*

Appendix A:

The following is excerpts from the Health Thermometer Service document Chapter 3. For the latest and most up to date information please consult the official document available from at:
https://www.bluetooth.com/specifications/gatt/ under the "Health thermometer service

1 Chapter 3 Service Characteristics

The following characteristics are exposed in the Health Thermometer Service. Unless otherwise specified, only one instance of each characteristic is permitted within this service.

Characteristic Name	Requirement	Mandatory Properties	Optional Properties	Security Permissions
Temperature Measurement	M	Indicate		None.
Client Characteristic Configuration descriptor	M	Read, Write		None.
Temperature Type	O	Read		None.
Intermediate Temperature	O	Notify		None.
Client Characteristic Configuration descriptor	C.1	Read, Write		None.
Measurement Interval	O	Read	Indicate, Write	Read: None. Writable with authentication.
Client Characteristic Configuration descriptor	C.2	Read, Write		None.
Valid Range descriptor	C.3	Read		None.

Table 3.1: Health Thermometer Service characteristics

C.1: Mandatory if Intermediate Temperature characteristic is supported, otherwise excluded.

C.2: Mandatory if Measurement Interval supports indications, otherwise excluded.

C.3: Mandatory if Measurement Interval is supported and Writable, otherwise excluded. Notes:

- Security Permissions of "None" means that this service does not impose any requirements.
- Properties not listed as Mandatory or Optional are Excluded.

1.1 Temperature Measurement

The Temperature Measurement characteristic is used to send a temperature measurement. Included in the characteristic are a Flags field (for showing the units of temperature and presence of optional fields), the temperature measurement value and, depending upon the contents of the Flags field, the time of that measurement and the temperature type.

The Temperature Type field in the Temperature Measurement characteristic is intended to be used when the value of temperature type is non-static (e.g., configured using a switch or a simple user interface on a thermometer). For temperature type values that are static, the Temperature Type characteristic (see Section 3.2) should be used.

Characteristic Behavior

When the *Client Characteristic Configuration* descriptor is configured for indication and a temperature measurement is available, this characteristic shall be indicated while in a connection.

If a temperature measurement is available and a connection is not currently established, the Server shall become connectable to allow the Collector to create a link.

The Temperature Measurement characteristic contains time-sensitive data, thus the requirements for time-sensitive data and data storage defined in Section 3.5 apply.

Flags Field

The Flags field shall be included in the Temperature Measurement characteristic.

Reserved for Future Use (RFU) bits in the Flags field shall be set to 0.

Temperature Measurement Value Field

This Temperature Measurement Value field shall be included in the Temperature Measurement characteristic.

If the unit of the Temperature Measurement is in Celsius, bit 0 of the Flags field shall be set to 0. Otherwise, the unit shall be Fahrenheit and bit 0 of the Flags field shall be set to 1.

The Temperature Measurement Value field may contain special float value NaN (0x007FFFFF) defined in IEEE 11073-20601 [4] to report an invalid result from a computation step or missing data due to the hardware's inability to provide a valid measurement.

Time Stamp Field

The Time Stamp field shall be included in the Temperature Measurement characteristic if the device supports storing of data. Otherwise it is optional.

If a time stamp is supported, the Server shall set bit 1 of the Flags field to 1 and include the Time Stamp field in the Temperature Measurement characteristic. Otherwise bit 1 of the Flags field shall be set to 0 and the Time Stamp field shall not be included.

The value of the Time Stamp field is derived from a source of date and time at the time of measurement. If the Time Stamp feature is supported, a source of date and time is mandatory.

The date and time of the device may be updated by various means such as via a simple user interface on the device, via an external time service, etc.

The time stamp field is defined to use the same format as the Date Time characteristic [3]; however, a value of 0 for the month or day fields shall not be used for this service.

Temperature Type Field

If the Temperature Type value is supported and is non-static, the Server shall set bit 2 of the Flags field to 1 and include the Temperature Type field in the Temperature Measurement characteristic. Otherwise bit 2 of the Flags field shall be set to 0 and the Temperature Type field shall not be included.

Characteristic Descriptors

Client Characteristic Configuration Descriptor

The *Client Characteristic Configuration* descriptor shall be included in the Temperature Measurement characteristic.

1.2 Temperature Type

The Temperature Type characteristic is one of two methods that are used to describe the type of temperature measurement in relation to the location on the human body at which the temperature was measured.

There are two exclusive methods to enable a Thermometer to provide temperature type information to a Collector. Either one method or the other is used, but not both. The Temperature Type characteristic is intended to be supported when the value is static while in a connection. For temperature type values that are non-static while in a connection (e.g. configured using a switch or a simple user interface on a thermometer), the Temperature Type characteristics shall not be supported, rather the Temperature Type field in the Temperature Measurement characteristic (refer to Section 3.1) shall be supported.

If the thermometer is for general use, a value of 0x02 Body (general) may be used.

Characteristic Behavior

The Temperature Type characteristic returns the current temperature type value when read.

Characteristic Behavior

The Intermediate Temperature characteristic is notified frequently during the course of a measurement so that a receiving device can effectively update the display on its user interface during the measurement process. The update rate is defined by the Server and should not be greater than a rate necessary to provide an acceptable user display on the Client. Typical update intervals range from 0.25 seconds to 2 seconds.

When the *Client Characteristic Configuration* descriptor is configured for notification and an intermediate temperature value is available, this characteristic shall be notified while in a connection.

If an intermediate temperature value is available and a connection is not currently established, the Server shall become connectable to allow the Collector to create a link.

To avoid transmitting unnecessary data, the Time Stamp and Temperature Type fields should not be used in the Intermediate Temperature characteristic.

Once the measurement process is complete and a stable temperature measurement is available, the Server shall stop notifications of the Intermediate Temperature characteristic and, if indications are enabled, shall indicate the Temperature Measurement characteristic.

Flags Field

The Flags field shall be included in the Intermediate Temperature characteristic.

RFU bits in the Flags field shall be set to 0.

Time Stamp Field

The Time Stamp field may be included in the Intermediate Temperature characteristic.

If a time stamp is supported, the Server shall set bit 1 of the Flags field to 1 and include the Time Stamp field in the Intermediate Temperature characteristic. Otherwise bit 1 of the Flags field shall be set to 0 and the Time Stamp field shall not be included.

The value of the Time Stamp field shall be derived from the date and time of the device at the time of measurement.

The date and time of the device may be updated by various means such as via a simple user interface on the device, via an external time service, etc.

The time stamp field is defined to use the same format as the Date Time characteristic [3], however, a value of 0 for the month or day fields is not permitted for this service.

Temperature Type Field

If the Temperature Type value is supported, the Server shall set bit 2 of the Flags field to 1 and include the Temperature Type field in the Intermediate Temperature characteristic. Otherwise bit 2 of the Flags field shall be set to 0 and the Temperature Type field shall not be included.

1.3 Measurement Interval

The Measurement Interval characteristic is used to enable and control the interval between consecutive temperature measurements. Note: This interval is not related to the intermediate temperature feature described in Section 3.3 and is only applicable to the Temperature Measurement characteristic.

The Valid Range descriptor defined in Section 3.4.2.2 enables a Client to determine the supported range of the Measurement Interval characteristic value.

Characteristic Descriptors

Client Characteristic Configuration Descriptor

If the properties of this characteristic allow indications, then the *Client Characteristic Configuration* descriptor shall be included in the Measurement Interval characteristic, otherwise this descriptor shall not be included.

Valid Range Descriptor

This descriptor is Mandatory if the Measurement Interval characteristic is Writable.

The Valid Range descriptor enables a Client to determine the supported range of the Measurement Interval characteristic value.

The Valid Range descriptor returns the range of supported measurement interval values (i.e., the lower inclusive and upper inclusive values) when read.

A value of 0 is not valid for the lower inclusive value of the descriptor for this profile.

When used with the Measurement Interval characteristic, the Valid Range descriptor contains two unsigned 16-bit integers representing the valid range of values that the Measurement Interval characteristic can support in units of seconds.

See Section 3.6 for further requirements related to this descriptor.

1.4 Valid Range

The following section contains the generic description of the Valid Range characteristic descriptor. This section may be removed from future versions of this document and moved into another document as the descriptor may be used in other profiles and services. The specific use and requirements for this descriptor are contained in earlier sections of this specification.

The *Valid Range* declaration is an optional characteristic descriptor that defines the valid range of values for the *Characteristic Value*. The descriptor is used to communicate the valid range of values that can be written to a *Characteristic Value* by a client. The characteristic descriptor may occur in any position within the characteristic definition after the *Characteristic Value*. Only one *Valid Range* declaration shall exist in a characteristic definition.

The characteristic descriptor is contained in an Attribute and the *Attribute Type* shall be set to the UUID for ‹‹Valid Range›› and the *Attribute Value* shall be set to the characteristic descriptor value. The *Attribute Permissions* are specified by the profile or may be implementation specific if not specified otherwise.

Attribute Handle	**Attribute Type**	**Attribute Value**		**Attribute Permissions**
0xNNNN	0xuuuu – UUID for ‹‹Valid Range››	Minimum Value	Maximum Value	Higher layer profile or implementation specific

Table 3.2: Valid Range Declaration

The definition of the *Valid Range* descriptor *Attribute Value* is the following:

Field Name	Value Size	Description
Minimum Value	Size of the Characteristic Value	Minimum valid Characteristic Value
Maximum Value	Size of the Characteristic Value	Maximum valid Characteristic Value.

Table 3.3: Valid Range Definition

The minimum and maximum *Characteristic Values* are inclusive to the values specified in the Minimum Value and Maximum Value fields of the descriptor. *Characteristic Values* outside the valid range defined by this descriptor are considered invalid values and an error condition. The size of the *Minimum Value* and *Maximum Value* defined in the descriptor are based on the size of the *Characteristic Value* described by the descriptor.

The value of the *Maximum Value* shall always be greater than or equal to the value of the *Minimum Value*.

The units and data type of the Minimum Value and Maximum Value fields of the descriptor shall be the same as the characteristic.

Bit Ordering

The bit ordering used for the Valid Range descriptor shall be little-endian.

Appendix B

nRF Toolbox

The “nRF Toolbox” Application from Nordic Semiconductor can be used to verify the BLE connection and includes a sample Health Thermometer App. It has accompanying sample source code. This App can be downloaded from:

- Android
  - App: https://play.google.com/store/apps/details?id=no.nordicsemi.android.nrftoolbox
  - Source: https://github.com/NordicSemiconductor/Android-nRF-Toolbox/tree/master/app/src/main/java/no/nordicsemi/android/nrftoolbox/ht
- iOS
  - App : https://apps.apple.com/us/app/nrf-toolbox/id820906058
  - Source : https://github.com/NordicSemiconductor/IOS-nRF-Toolbox

Please note that this app, while running in the background of your phone, may randomly pair with your CORE and thus make it “invisible” to other apps such as the CORE app. To avoid this, withdraw the Bluetooth permission for the nRF Toolbox while you are not using it.

ERF Connect

The EFR Connect Mobile Application includes similar features as the nRF Toolbox, including a Health Thermometer demo. It can be downloaded from:

- Android
  - App: https://play.google.com/store/apps/details?id=com.siliconlabs.bledemo&hl=en
  - Source https://github.com/SiliconLabs/EFRConnect-android
- iOS
  - App: https://apps.apple.com/us/app/blue-gecko/id1030932759
  - Source: https://github.com/SiliconLabs/EFRConnect-ios

2 Appendix C

The full source code of the CORE wearOS app is available on GitHub: https://github.com/CoreBodyTemp/wearos-app

A rough outline from scanning for Bluetooth devices to establishing a connection and setting up notifications on the temperature measurement is illustrated with code snippets below.

Note: Some code snippets may only make sense in the context of the full project.

- Scanning for BLE devices:

```
/**
 * Enables or disables scanning for Bluetooth LE devices.
 * @param enable If true, enable scanning. False otherwise.
 */
private void scanLeDevice(final boolean enable) {
    if (mBluetoothLeScanner == null) {
        mBluetoothLeScanner =
                BluetoothAdapter.getDefaultAdapter().getBluetoothLeScanner();
    }
    if (enable) {
        if(!mBluetoothAdapter.isEnabled())
        {
            Intent enableBtIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        }
        mSettings = new
ScanSettings.Builder().setScanMode(SCAN_MODE_BALANCED).build();  //default
settings

        setScanning(true);
        Log.d(TAG, "scanLeDevice: startLeScan.");
        mBluetoothLeScanner.startScan(mFilters, mSettings, mScanCallback);

        mScanningTimeout.sendEmptyMessageDelayed(MSG_SCANNING_TIMEOUT,
scanningTimeout_Ms);

    } else {
        setScanning(false);
        if(mBluetoothLeScanner != null && mBluetoothAdapter.isEnabled()) {
            mBluetoothLeScanner.stopScan(mScanCallback);
        }
    }
}
```

- Connecting to the Gatt server on the CORE:

```
/**
 * Connects to the GATT server hosted on the Bluetooth LE device.
 *
 * @param address The device address of the destination device.
 *
 * @return Return true if the connection is initiated successfully. The connection
result
 *         is reported asynchronously through the
 *         {@code
BluetoothGattCallback#onConnectionStateChange(android.bluetooth.BluetoothGatt,
int, int)}
 *         callback.
 */
public boolean connect(final String address) {
    if (mBluetoothAdapter == null || address == null) {
        Log.w(TAG, "BluetoothAdapter not initialized or unspecified address.");
        return false;
    }

    if (!mBluetoothAdapter.isEnabled()) {
        Log.w(TAG, "BluetoothAdapter is not enabled");
        return false;
    }

    // Previously connected device.  Try to reconnect.
    if (address.equals(mBluetoothDeviceAddress)
            && mBluetoothGatt != null) {
        Log.d(TAG, "Trying to use an existing mBluetoothGatt for connection.");
        if (mBluetoothGatt.connect()) {
            return true;
        } else {
            return false;
        }
    }

    final BluetoothDevice device = mBluetoothAdapter.getRemoteDevice(address);
    if (device == null) {
        Log.w(TAG, "Device not found.  Unable to connect.");
        return false;
    }

    mBluetoothGatt = device.connectGatt(this, true, mGattCallback);
    Log.d(TAG, "Trying to create a new connection.");
    mBluetoothDeviceAddress = address;
    return true;
}
```

- Implementation of the corresponding BluetoothGattCallback where changes of the connection and receiving data etc. is reported:

```
private final BluetoothGattCallback mGattCallback = new BluetoothGattCallback() {
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt, int status, int
newState) {
        String intentAction;
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            intentAction = ACTION_GATT_CONNECTED;
            broadcastUpdate(intentAction);
            Log.i(TAG, "Connected to GATT server.");
            // Attempts to discover services after successful connection.
            Log.i(TAG, "Attempting to start service discovery:" +
                    mBluetoothGatt.discoverServices());

        } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
            intentAction = ACTION_GATT_DISCONNECTED;
            AppPreferences.setLastCbtValue(BluetoothLeService.this, 0);
            Log.i(TAG, "Disconnected from GATT server.");
            broadcastUpdate(intentAction);
        }
    }

    @Override
    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_GATT_SERVICES_DISCOVERED);
        } else {
            Log.w(TAG, "onServicesDiscovered received: " + status);
        }
    }

    @Override
    public void onCharacteristicRead(BluetoothGatt gatt,
                                     BluetoothGattCharacteristic characteristic,
                                     int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
        }
    }

    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
                                        BluetoothGattCharacteristic
characteristic) {
        broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
    }
};
```

- After a successful service discovery (mBluetoothGatt.discoverServices(), called in GattCallback above), we can set up notifications on the temperature measurement characteristic:

```
/**
 * Enables or disables notification on a give characteristic.
 *
 * @param characteristic Characteristic to act on.
 * @param enabled If true, enable notification.  False otherwise.
 */
public void setCharacteristicNotification(BluetoothGattCharacteristic
characteristic,
                                          boolean enabled) {
    if (mBluetoothAdapter == null || mBluetoothGatt == null) {
        Log.w(TAG, "BluetoothAdapter not initialized");
        return;
    }
    mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);

    // This is specific to Temperature Measurement.
    if (UUID_TEMPERATURE_MEASUREMENT.equals(characteristic.getUuid())) {
        BluetoothGattDescriptor descriptor = characteristic.getDescriptor(
                UUID.fromString("00002902-0000-1000-8000-00805f9b34fb"));
        if (enabled) {

descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
        } else {

descriptor.setValue(BluetoothGattDescriptor.DISABLE_NOTIFICATION_VALUE);
        }
        mBluetoothGatt.writeDescriptor(descriptor);
    }
}
```

3 Appendix D

The following is example code from David Vassallo's Blog "**BLE Health Devices: First Steps with Android**". Full text and further explanation can be found at:
http://blog.davidvassallo.me/2015/09/02/ble-health-devices-first-steps-with-android/

```
package com.sixpmplc.ble_demo;

import android.annotation.TargetApi;
import android.app.Activity;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothGatt;
import android.bluetooth.BluetoothGattCallback;
import android.bluetooth.BluetoothGattCharacteristic;
import android.bluetooth.BluetoothGattDescriptor;
import android.bluetooth.BluetoothGattService;
import android.bluetooth.BluetoothManager;
import android.bluetooth.BluetoothProfile;
import android.bluetooth.le.BluetoothLeScanner;
import android.bluetooth.le.ScanCallback;
import android.bluetooth.le.ScanFilter;
import android.bluetooth.le.ScanResult;
import android.bluetooth.le.ScanSettings;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.os.Build;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.support.v7.app.ActionBarActivity;
import android.util.Log;
import android.widget.TextView;
import android.widget.Toast;

import java.util.ArrayList;
import java.util.List;

@TargetApi(21)
public class MainActivity extends ActionBarActivity {
    private BluetoothAdapter mBluetoothAdapter;
    private int REQUEST_ENABLE_BT = 1;
    private Handler mHandler;
    private static final long SCAN_PERIOD = 30000;
    private BluetoothLeScanner mLEScanner;
    private ScanSettings settings;
    private List<ScanFilter> filters;
    private BluetoothGatt mGatt;
    private BluetoothDevice mDevice;

    // setup UI handler
    private final static int UPDATE_DEVICE = 0;
    private final static int UPDATE_VALUE = 1;
    private final Handler uiHandler = new Handler() {
        public void handleMessage(Message msg) {
            final int what = msg.what;
            final String value = (String) msg.obj;
            switch(what) {
```

```
            case UPDATE_DEVICE: updateDevice(value); break;
            case UPDATE_VALUE: updateValue(value); break;
        }
    }
};

private void updateDevice(String devName){
    TextView t=(TextView)findViewById(R.id.dev_type);
    t.setText(devName);
}

private void updateValue(String value){
    TextView t=(TextView)findViewById(R.id.value_read);
    t.setText(value);
}

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mHandler = new Handler();
    if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
        Toast.makeText(this, "BLE Not Supported",
                Toast.LENGTH_SHORT).show();
        finish();
    }
    final BluetoothManager bluetoothManager =
            (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
    mBluetoothAdapter = bluetoothManager.getAdapter();

}

@Override
protected void onResume() {
    super.onResume();
    if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
        Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
    } else {
        if (Build.VERSION.SDK_INT >= 21) {
            mLEScanner = mBluetoothAdapter.getBluetoothLeScanner();
            settings = new ScanSettings.Builder()
                    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
                    .build();
            filters = new ArrayList<ScanFilter>();
        }
        scanLeDevice(true);
    }
}

@Override
protected void onPause() {
    super.onPause();
    if (mBluetoothAdapter != null && mBluetoothAdapter.isEnabled()) {
        scanLeDevice(false);
    }
}

@Override
protected void onDestroy() {
    if (mGatt == null) {
```

```
            return;
        }
        mGatt.close();
        mGatt = null;
        super.onDestroy();
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == REQUEST_ENABLE_BT) {
            if (resultCode == Activity.RESULT_CANCELED) {
                //Bluetooth not enabled.
                finish();
                return;
            }
        }
        super.onActivityResult(requestCode, resultCode, data);
    }

    private void scanLeDevice(final boolean enable) {
        if (enable) {
            mHandler.postDelayed(new Runnable() {
                @Override
                public void run() {
                    if (Build.VERSION.SDK_INT < 21) {
                        mBluetoothAdapter.stopLeScan(mLeScanCallback);
                    } else {
                        mLEScanner.stopScan(mScanCallback);

                    }
                }
            }, SCAN_PERIOD);
            if (Build.VERSION.SDK_INT < 21) {
                mBluetoothAdapter.startLeScan(mLeScanCallback);
            } else {
                mLEScanner.startScan(filters, settings, mScanCallback);
            }
        } else {
            if (Build.VERSION.SDK_INT < 21) {
                mBluetoothAdapter.stopLeScan(mLeScanCallback);
            } else {
                mLEScanner.stopScan(mScanCallback);
            }
        }
    }


    private ScanCallback mScanCallback = new ScanCallback() {
        @Override
        public void onScanResult(int callbackType, ScanResult result) {
            Log.i("callbackType", String.valueOf(callbackType));
            String devicename = result.getDevice().getName();

            if (devicename != null){
                if (devicename.startsWith("TAIDOC")){
                    Log.i("result", "Device name: "+devicename);
                    Log.i("result", result.toString());
                    BluetoothDevice btDevice = result.getDevice();
                    connectToDevice(btDevice);
                }
            }
```

```
    }

    @Override
    public void onBatchScanResults(List<ScanResult> results) {
        for (ScanResult sr : results) {
            Log.i("ScanResult - Results", sr.toString());
        }
    }

    @Override
    public void onScanFailed(int errorCode) {
        Log.e("Scan Failed", "Error Code: " + errorCode);
    }
};

private BluetoothAdapter.LeScanCallback mLeScanCallback =
        new BluetoothAdapter.LeScanCallback() {
            @Override
            public void onLeScan(final BluetoothDevice device, int rssi,
                                 byte[] scanRecord) {
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        Log.i("onLeScan", device.toString());
                        connectToDevice(device);
                    }
                });
            }
        };

public void connectToDevice(BluetoothDevice device) {
    if (mGatt == null) {
        Log.d("connectToDevice", "connecting to device: "+device.toString());
        this.mDevice = device;
        mGatt = device.connectGatt(this, false, gattCallback);
        scanLeDevice(false);// will stop after first device detection
    }
}

private final BluetoothGattCallback gattCallback = new BluetoothGattCallback() {


    @Override
    public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState) {
        Log.i("onConnectionStateChange", "Status: " + status);
        switch (newState) {
            case BluetoothProfile.STATE_CONNECTED:
                Log.i("gattCallback", "STATE_CONNECTED");

                //update UI
                Message msg = Message.obtain();

                String deviceName = gatt.getDevice().getName();
                switch (deviceName){
                    case "TAIDOC TD1261":
                        deviceName = "Thermo";
                        break;
                    case "TAIDOC TD8255":
                        deviceName = "SPO2";
                        break;
```

```
                case "TAIDOC TD4279":
                    deviceName = "SPO2";
                    break;
            }

            msg.obj = deviceName;
            msg.what = 0;
            msg.setTarget(uiHandler);
            msg.sendToTarget();

            gatt.discoverServices();
            break;
        case BluetoothProfile.STATE_DISCONNECTED:
            Log.e("gattCallback", "STATE_DISCONNECTED");
            Log.i("gattCallback", "reconnecting...");
            BluetoothDevice mDevice = gatt.getDevice();
            mGatt = null;
            connectToDevice(mDevice);
            break;
        default:
            Log.e("gattCallback", "STATE_OTHER");
    }

}

@Override
public void onServicesDiscovered(BluetoothGatt gatt, int status) {
    mGatt = gatt;
    List<BluetoothGattService> services = gatt.getServices();
    Log.i("onServicesDiscovered", services.toString());
    BluetoothGattCharacteristic therm_char = services.get(2).getCharacteristics().get(0);

    for (BluetoothGattDescriptor descriptor : therm_char.getDescriptors()) {
        descriptor.setValue( BluetoothGattDescriptor.ENABLE_INDICATION_VALUE);
        mGatt.writeDescriptor(descriptor);
    }

    //gatt.readCharacteristic(therm_char);
    gatt.setCharacteristicNotification(therm_char, true);

}

@Override
public void onCharacteristicRead(BluetoothGatt gatt,
                                 BluetoothGattCharacteristic
                                         characteristic, int status) {
    Log.i("onCharacteristicRead", characteristic.toString());
    byte[] value=characteristic.getValue();
    String v = new String(value);
    Log.i("onCharacteristicRead", "Value: " + v);
    //gatt.disconnect();
}

@Override
public void onCharacteristicChanged(BluetoothGatt gatt,
                                    BluetoothGattCharacteristic
                                            characteristic) {
    float char_float_value = characteristic.getFloatValue(BluetoothGattCharacteristic.FORMAT_FLOAT,1);
    Log.i("onCharacteristicChanged", Float.toString(char_float_value));
```

```
            String deviceName = gatt.getDevice().getName();
            String value = null;
            switch (deviceName){
               case "TAIDOC TD1261":
                  value = Float.toString(char_float_value);
                  break;
               case "TAIDOC TD8255":
                  value = Float.toString(char_float_value*10);
                  break;
            }

            //update UI
            Message msg = Message.obtain();
            msg.obj = value;
            msg.what = 1;
            msg.setTarget(uiHandler);
            msg.sendToTarget();



            //gatt.disconnect();
         }

      };
   }
```