CORE Bluetooth® and ANT+ Communication

Implementation Notes

V3.1

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
V3.0	2023-03-21	Add notes on ANT+ communication, add section about displaying Data from CORE
V3.1	2024-08-08	Add documentation about CORE Heat Strain Index broadcast.

CORE is a venture of greenTEG AG.

Contents

1 Overview

CORE is an easy-to-use wearable product that can accurately monitor core body temperature[1]. Our wireless connectivity suite includes both an interface for ANT+ and for Bluetooth (starting from Bluetooth Version 4.2).

For ANT+ developers, CORE supports the ANT+ Device Profile “CoreTemp” open standard. To get the live core body temperature along with other information such as skin temperature and measurement quality, the data page 1 “Temperature” can be used. Additionally, heart rate monitors can be paired using the common page 74: Open channel command.

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 Communication via Bluetooth

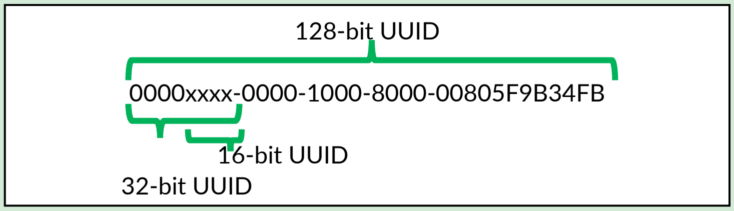
The standard services are in accordance with the Bluetooth SIG specifications. The latest specifications can be found online under https://www.bluetooth.com/specifications/specs/. Core Body Temperature Service is a custom specification issued by CORE and it is available on www.github.com/CoreBodyTemp/CoreBodyTemp.

An Overview of the Bluetooth GATT 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/.

2.1 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.

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

Table 1 Advertisement packet of CORE.

	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

Table 2 Example parsing manufacturer-specific data payload 0x0B-F6-00-04-B3-91.

*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.

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**	0x21E1DA14B5977CB047431E5B00210000	Core Temperature Service (0x00002100-5B1E-4347-B07C-97B514DAE121)

Table 3 Active scan response of CORE.

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.

2.2 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: The Core Body Temperature (Table 4) and the CoreTemp Control Point (Table 6) characteristic. An example parsing of the payload from the core body temperature characteristic can be found in Table 5. Note that the Heat Strain Index is not valid (flag bit 5 is set to 0 in that case), if the core temperature is not valid.

<table>
<tr><td>UUID128</td><td>00002101-5B1E-4347-B07C-97B514DAE121</td></tr>
<tr><td>payload</td><td>The <b>payload</b> of the Core Body Temperature characteristic, contains (depending on the flag field, not all values may be present):
LSO MSO
<table>
<tr><td><b>Payload</b></td><td><i>Flag</i></td><td><i>Core Body Temperature</i></td><td><i>Skin Temperature</i></td><td><i>Core Reserved</i></td><td><i>Quality &amp; State</i></td><td><i>Heart Rate</i></td><td><i>Heat Strain Index</i></td></tr>
<tr><td><b>Data Type</b></td><td>8bit</td><td>SINT16</td><td><i>SINT16</i></td><td>SINT16</td><td>8bit</td><td>UINT8</td><td>UINT8</td></tr>
<tr><td><b>Size</b></td><td>8bit</td><td>16bit</td><td>16bit</td><td>16bit</td><td>8 bit</td><td>8bit</td><td>8bit</td></tr>
<tr><td><b>Unit</b></td><td>N/A</td><td>0.01°C</td><td>0.01°C</td><td>N/A</td><td>N/A</td><td>BPM</td><td>0.1 a.u.</td></tr>
</table>
</td></tr>
<tr><td>Notes</td><td>Notifications on this characteristic can be activated by setting the corresponding CCCD value.
See the full specification for details.</td></tr>
</table>

Table 4 Core Body Temperature characteristic.

LSO MSO

Payload	*Flag*	*Core Body Temperature*	*Skin Temperature*	*Core Reserved*	*Quality & State*	*Heart Rate*	*Heat Strain Index*
Binary	0x37	0x0F19	0x0D5C	0x002F	0x11	0x00	0x27
Value	*all fields are valid, Temperature Unit is °C*	38.65°C	35.22°C	47	Quality 'poor' Heart Rate supported, not receiving signal	0	3.9

Table 5 Example: parsing Core Body Temperature characteristic data payload 0x37-19-0F-A4-5C-0D-2F-00-11-00-27. Note that print() in Python renders this payload as b"7\x19\x0f\xa4\r/\x00\x11\x00'"

<table>
<tr><td>UUID128</td><td>00002102-5B1E-4347-B07C-97B514DAE121</td></tr>
<tr><td>payload</td><td>The <b>payload</b> of the Core Body Temperature characteristic, contains (depending on the flag field, not all values may be present):
LSO MSO
<table>
<tr><td><b>Payload</b></td><td><i>OpCode</i></td><td><i>Parameter Value</i></td></tr>
<tr><td><b>Data Type</b></td><td>8bit</td><td><i>Variable, e. g. an ANT+ id to pair a heart rate monitor.</i></td></tr>
<tr><td><b>Size</b></td><td>8bit</td><td><i>Variable</i></td></tr>
</table>
</td></tr>
<tr><td>Notes</td><td>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.</td></tr>
</table>

Table 6 CoreTemp Control Point characteristic.

2.3 Health Thermometer Service – UUID16 0x1809

The CORE device implements two of the health thermometer service characteristics: the temperature measurement characteristic (Table 7) and the Temperature Type characteristic (Table 8).

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

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

Table 7 Temperature Measurement characteristic.

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

Table 8 Temperature Type characteristic.

2.4 Battery – UUID16 0x180F

The CORE sensor also implements the battery service as a complementary information.

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

Table 9 Battery Level characteristic

[44] 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

2.5 Other Services

The CORE sensor also implements the Generic Access Service and Generic Attribute Service to be compliant with the GATT specification.

UUID16	0x1800 – Generic Access
	Characteristics: 2a00 – Device Name (CORE [x] with x set to the device state a, m, etc[5]) 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.

Table 10 Generic Access Service and Generic Attribute Service.

2.6 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 A.

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 .

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 GATTt descriptor (see https://stackoverflow.com/questions/52598927/read-and-notify-ble-android)

Table 11 Trouble shooting Bluetooth connections.

[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

3 Communication via ANT+

The detailed specification of the CORE Temp Device Profile are available upon request. The CORE sensor is configured as a master device and has a device type of 127 (0x7F). The device Number ranges from 1-65535, which may be extended by 4bits to comply to the “extended device number specification”. The device number can be derived from the decimal representation of the two LSO of the Bluetooth device number and vice versa.

There are two main data pages: the general information (page 0) and the CORE Temp page (page 1), which are recommended to be sent at 2 Hz and upon request of the slave. Live temperature data from CORE can be extracted from page 1. Additionally, standard pages 80 and 81 are available, as described in Table 12.

	Payload Description
Page 0	The **payload** of page 0 – General Information - contains:

Byte	*Description*	*Length*	*Value*
0	Data Page Number	1 Byte	Data Page Number = 0 (0x00)
1	Reserved	1 Byte	Set to 0xFF
2	Data Quality	1 Byte	Optional (0xFF if not used) Bit 0:1 – 00-Poor, 01-Fair, 10-Good, 11-Excellent Bit 2:7 – RFU, set to 0
3	Transmission Info	1 Byte	*details in “ANT+ Device Profile – CORE Temp”*
4-7	Supported Pages	4 Bytes (LSO..MSO)	Bit 0 – Page 0 Support Bit 1 – Page 1 Support Bit 2:31 RFU, set to 0.

Page 1	The **payload** of page 1 – CORE Temp - contains:

Byte	*Description*	*Length*	*Value*	Units	Range
0	Data Page Number	1 Byte	Data Page Number = 1 (0x01)	N/A	N/A
1	Heat Strain Index	1 Byte	Signed Integer 1 Byte (SINT8) in units of 0.1 (0xFF invalid)	0.1 a.u.	0 to 25.4
2	Event count	1 Byte	Increments with each measurement	N/A	N/A
3	Skin Temp. LSO	1.5 Bytes	Optional: Signed Integer representing Skin Temperature (0x800 invalid – or not supported) Skin temperature = Value / 20	0.05°C	-102.35 to 102.35
4 (bits 4:7)	Skin Temp. MSN				
4 (bits 0:3)	Reserved LSN	1.5 Bytes	(0x800 invalid – or not supported)	N/A	0 to 0xFFF
5	Reserved MSB				
6:7	CORE Temperature	2 Bytes (LSO..MSO)	SINT16 representing current CORE Temperature (0x8000 invalid)	0.01°C	-327.67 to 327.67

Page 80	The **payload** of page 80 (0x50) – Manufacturer’s Identification – contains:

Byte	*Description*	*Length*	*Value*
0	Data Page Number	1 Byte	Data Page Number = 80 (0x50)
1	Reserved	1 Byte	Set to 0xFF
2	Reserved	1 Byte	Set to 0xFF
3	HW Revision	1 Byte	To be set by the manufacturer.
4:5	Manufacturer ID	2 Bytes (LSO..MSO)	Contact the ANT+ Alliance for a current list of manufacturing IDs, or to be assigned a manufacturing ID.
6:7	Model Number	2 Bytes (LSO..MSO)	To be set by the manufacturer.

<table>
<tr><td></td><td></td></tr>
<tr><td>Page 81</td><td>The <b>payload</b> of page 81 (0x51) – Product Information – contains:
<table>
<tr><th>Byte</th><th><i>Description</i></th><th><i>Length</i></th><th><i>Value</i></th></tr>
<tr><td><b>0</b></td><td>Data Page Number</td><td>1 Byte</td><td>Data Page Number = 81 (0x51)</td></tr>
<tr><td><b>1</b></td><td>Reserved</td><td>1 Byte</td><td>Set to 0xFF</td></tr>
<tr><td><b>2</b></td><td>Reserved</td><td>1 Byte</td><td>Set to 0xFF</td></tr>
<tr><td><b>3</b></td><td>SW Revision</td><td>1 Byte</td><td>To be set by the manufacturer.</td></tr>
<tr><td><b>4:7</b></td><td>Serial Number</td><td>4 Bytes (LSO..MSO)</td><td>The lowest 32 bits of the serial number. Value 0xFFFFFFFF to be used for devices without serial number</td></tr>
</table></td></tr>
<tr><td>Notes</td><td>The data pages 0 and 1 are sent with 2Hz and upon request. Information about Heart rate pairing support, local time and UTC time is encoded in Byte 3 “Transmission Info” of page 0.</td></tr>
</table>

Table 12 Condensed payload description of the CORE Temp data pages. For details, please refer to the official ANT+ Device Profile specification.

In addition to the data pages, the common page 74: Open Channel Command (0x4A) is supported by the CORE Temp profile, which will be used for heart rate monitor pairing. Please refer to section 4.2 for further details. Finally, the common page 70: Request data page is also accepted as an acknowledged message from a display to request any data page.

4 App Integration and Data Display

Depending on the use case, different ways of displaying information through the Bluetooth Core Body Temperature Characteristic or the ANT+ CoreTemp Profile are suggested below. The details are explained in the sections below.

	Recommended displayed information	**Optional displayed information**
Activity Tracking, (live)	• Current Core body temperature • Current Heat Strain Index (HSI)	• Quality Index, e. g. grey/semitransparent value or blinking if quality is reduced. • Skin temperature (current) • Heat Zone (color coded)
Activity Tracking, (summary)	• Core body temperature (max, average) • Core body temperature time-series graph • Heat Strain Indes time series graph	• Indication of quality index in the timeseries graph, e. g. with color code or dashed line. • Skin temperature time series graph • Time in Heat Zones bar chart.

Table 13 Display information guidelines.

While core body temperature is the primary output of CORE, the CORE Heat Strain Index (HSI) is of equal importance, if not higher. The HSI represents the strain on the body and enables users to adjust their workout and race effort based on an easy zone system called CORE Heat Zones. The four Heat Zones break down the complex processes of heat strain on the body into actionable guidelines for the users (Table 14).

➔ Read more on https://corebodytemp.com/pages/cores-heat-zones

Heat Zone	**Heat Zone HSI range**	**Guidelines for Pacing**	**Guidelines for Heat Training**
Zone 4	> 7	Reduce HSI rapidly	Harmful
Zone 3	3 – 6.9	Adjust pacing, hydrate and cool	Optimal heat training
Zone 2	1 – 2.9	Use discretion	Partial heat training
Zone 1	0 – 0.9	Pacing not affected	No heat training

Table 14 CORE Heat Zone definition and user guidelines.

Limitations:

- No historical data can be downloaded with the open profiles (BLE Core Body Temperature Service, ANT+ CoreTemp Profile). Proprietary Bluetooth API available upon request.
- The battery saver mode cannot be set over the Core Body Temperature Service. This setting is only available in the proprietary Bluetooth API.

4.1 Activity Tracking (Live)

Live data can be obtained via notifications on the Core Body Temperature Characteristic (Bluetooth) or by parsing the data pages 0,1 (ANT+). There is no way to download historical data from CORE via the CTS. We recommend parsing the data into a FIT file on the display device (see section 4.1.3). An example of data quality visual displays can be found in the *"ANT+ Device Profile – CORE Temp"* and is copied to Table 15below.

Data Quality	**Text Color on Black background**	**Text Color on White**	**Optional Dashes to indicate quality**
Bit 00 – Poor	Dark gray	Light gray	36.6° alternate flash 3-.--°
Bit 01 – Fair	Gray	Gray	36.6° alternate flash 36.--°
Bit 10 - Good	Light gray	Dark gray	36.6° alternate flash 36.6-°
Bit 11 - Excellent	White	Black	36.6° solid / no flash

Table 15 Example Data Quality visual displays (from "ANT+ Device Profile - CORE Temp")

We recommend to display the live Heat Strain Index in a separate data field. The display can be enhanced by color coding the live Heat Strain Index according to the heat zone (definition of Heat Zones see Table 14).

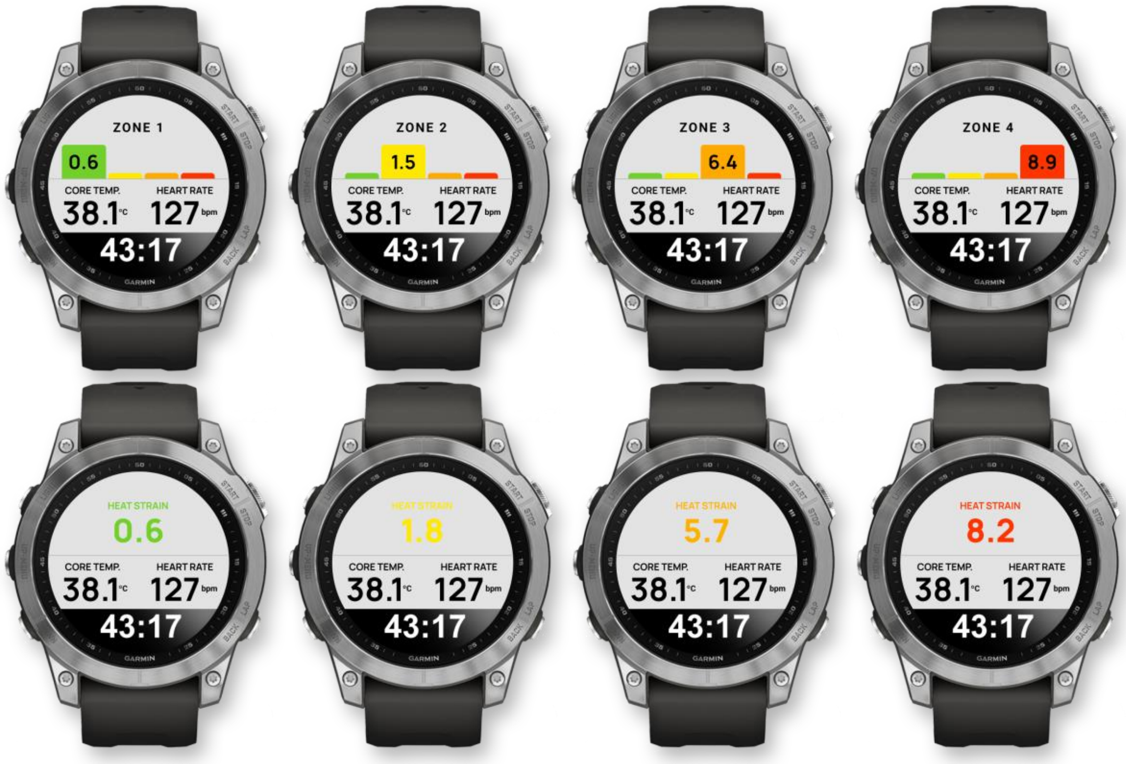


Figure 2 Two display variants of the heat strain index for inspiration.

4.1.1 Bluetooth connection

The Core Body Temperature Characteristic and the Health Thermometer Characteristic both support notifications by switching the respective CCCD. The live values of the metrics of interest can be extracted from this Bluetooth characteristic.

It is recommended to primarily use the Core Body Temperature Characteristic to get the most out of CORE live data transmission. The update rates for the notifications are currently set to 1Hz. However, there is no guaranteed update rate for core body temperature. It may be considerably lower than 1Hz due to the slow nature of the physical signal (The core body temperature is supposed to be relatively sluggish as opposed to for example heart rate). This means, that for some period, only the value for skin temperature will be updated, whereas the core body temperature value won't change.

4.1.2 ANT+ connection

The data page 1 (Temperature, including skin temperature) is broadcast at 2 Hz. At a lower frequency, the data page 0 0 (General Information, including signal quality) is sent instead of data page 1. It is recommended to follow the guidelines for data presentation in the introduction to this section in Table 13.

4.1.3 Fit File writing

A core body temperature display device can store data it receives over Bluetooth or ANT+ using the FIT protocol for ease of sharing the data with different platforms. The data shall be stored in a FIT activity file which must contain the file_id, activity, session, lap, record messages and device messages. The field_id and corresponding field name must be strictly used, as well as the unit (°C). Otherwise, there is no guarantee the data will be parsed correctly by the different fit viewing tools such as WKO5, TrainingPeaks and GoldenCheetah.

Important Guidelines for Fit File writing

All values must be recorded in Celsius.

For consistency, the standard names and the native numbers as declared in the FIT SDK must be used. Some values **do not have a native Fit Number assigned!**

Fit Message	*Required*	*Field Name*	*Units*	*Data Type*	*Native Field Id*
session	N	avg_core_temperature	°C	DATA_TYPE_FLOAT	208
	N	min_core_temperature	°C	DATA_TYPE_FLOAT	209
	N	max_core_temperature	°C	DATA_TYPE_FLOAT	210
	N	CIQ_device_info	N/A	*Optional, recommend to follow format in Garmin (see* Table 17*)*	26
lap	N	avg_core_temperature	°C	DATA_TYPE_FLOAT	158
	N	min_core_temperature	°C	DATA_TYPE_FLOAT	159
	N	max_core_temperature	°C	DATA_TYPE_FLOAT	160
record	Y	core_temperature	°C	DATA_TYPE_FLOAT	139
	N	skin_temperature	°C	DATA_TYPE_FLOAT	***None***
	N	core_data_quality	Q	UINT8 (0 to 3 without heart rate, 16-20 with heart rate)	***None***
	Y	heat_strain_index	a.u.	DATA_TYPE_FLOAT	***None***
	N	core_reserved	kcal	UINT16	***None***

Table 16 Field Names and Native Field Id for CORE sensor data.

Byte nr.	*Example value*	*Example value hex*	*Definition*
0	127	0x7F	ANT+ device type “CORE”
1	19	0x13	Wearing position “Armpit”
2	175	0xAF	Device ID m3
3	2	0x02	Device ID m2
4	35	0x23	Device ID m1
5	29	0x1D	Device ID m0 -> Device ID 0x(1D-23-02-AF)
6	47	0x2F	Manufacturer ID LSB
7	1	0x01	Manufacturer ID MSB -> Manufacturer ID 0x012F
8	1	0x01	Battery status (1 New - 2 Good - 3 Ok - 4 Low - 5 Critical - 6 RFU - 7 Invalid)
9	6	0x06	Supplemental SW revision
10	8	0x08	Main SW revision -> Software Revision 0.8.6
11	3	0x03	Hardware revision
12	1	0x01	Model Number LSB
13	0	0x00	Model Number MSB

Table 17 CIQ_device_info field for Garmin Fit Files that record data from CORE.

4.2 Heart Rate Monitor Pairing

Usually without heart rate being fed into the algorithm of CORE there is a lag in the measured core body temperature. CORE uses the heart rate signal to improve the speed of the core body temperature measurement. This can be especially helpful during high activity such as sport. Heart rate is a “energy vector” in the body, from which changes in the core body temperature can be anticipated faster and more accurately than without. CORE can be paired to all standard ANT+ and Bluetooth Heart rate monitors. We recommend to prefer pairing ANT+ over BLE.

4.2.1 Bluetooth connection

If the communication is based on Bluetooth, using the CoreTemp Control Point Characteristic allows for pairing of external Heart Rate Monitors to the CORE sensor.

It is recommended to support at least OpCode 0x02 to add a heart rate monitor to the list of paired devices, providing the ANT+ ID as a parameter value.

4.2.2 ANT+ connection

It is currently not possible to pair CORE to a heart rate monitor over the CORE ANT+ profile. As a fallback, the users should be instructed to use the CORE app to pair CORE to a heart rate monitor.

5 Glossary

Term	Requirement
BLE	Bluetooth Low Energy
LSN	Least significant nibble (lower 4 bits of an octet)
LSO	Least significant octet
MSN	Most significant nibble (upper 4 bits of an octet)
MSO	Most significant octet
HRM	Heart Rate Monitor
UUID	Universally Unique Identifier
CTS	Core Body Temperature Service
RFU	Reserved for future use
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.

6 Further Resources:

- Official CORE app on the Apple Store and the Google Play Store.
  https://apps.apple.com/us/app/id1521866309
  https://play.google.com/store/apps/details?id=com.greenteg.core.app

- CORE Firmware Changelog:
  https://corebodytemp.com/pages/tech-core-changelog

- The ***CoreTemp BLE Service Specification*** is available online or upon request:
  https://github.com/CoreBodyTemp/CoreBodyTemp

- The Specification of the additional official GATT Services supported by CORE are available on
  https://www.bluetooth.com/specifications/specs/

- The ***CoreTemp ANT+ Profile Specification*** is available from CORE upon request through our webpage (https://corebodytemp.com/pages/contact)

- WearOS sample app (Bluetooth based communication), source accessible on:
  https://github.com/CoreBodyTemp/wearos-app

- ConnectIQ-CoreTemp sample app (ANT+ based communication), source accessible on https://github.com/CoreBodyTemp/ConnectIQ-CoreTemp

Appendix A– General Android Apps illustrating Bluetooth communication.

Appendix – Excerpts from CoreBodyTemp’s sample wearOS app

Appendix – Excerpts from David Vassallo’s Blog *“BLE Health Devices: First Steps with Android”*

Appendix A

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

7 Appendix B

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);
    }
}
```

8 Appendix C

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();
        }

    };
}
```