apptionLABS

MEATER+ S1 BLE Specification

Table of Contents

Version History

Date	Author	Notes
7/12/2022	jerome@meater.com	Initial Version

Introduction

MEATER Probe is a wireless bluetooth meat thermometer. Please familiarise yourself with the product before implementing the protocol in this document. You can find more information at https://www.meater.com

The MEATER+ is designed to work with the corresponding MEATER Probe.

Our data interface is designed for use with the MEATER app, and not shared widely as an open API. The interface is subject to change in future. Please contact us if you have concerns regarding future changes to the data formats or protocols.

If you have a requirement for something not covered by this document, please contact us and we can help.

Each MEATER+ can only be connected over BLE to a single device, so it’s best to ensure that the MEATER app is not running on any nearby mobile devices.

Operation

The MEATER+ and probe turn on and start advertising when the probe is removed from its charger. It follows Apple’s guidelines for the advertising interval, connection parameters, etc.

Security

MEATER data is not considered sensitive information, so is sent unencrypted. No BLE security features are used, and no proprietary encryption is used. Data formats are proprietary, but we are not intentionally trying to hide data.

Advertising Rates

MEATER+ currently has a fixed rate 152.5ms advertising, this may change in the future.

Advertising: Scan and identify a MEATER+ device using manufacturer data

The following chapter describes how to recognise a device is the MEATER+ amongst other advertising devices.

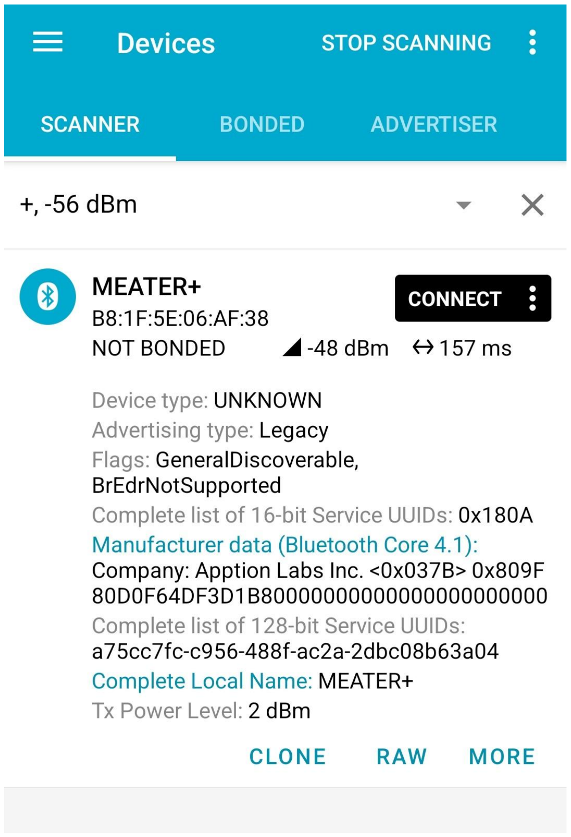
It is possible to do a first search and filtering using the Service UUID, which is in the Scan Response:

UUID: `A75CC7FC-C956-488F-AC2A-2DBC08B63A04`

In this document we always exclude the TYPE “manufacturer data”, recognised as “0xFF” (1 byte). The TYPE byte may or may not be included in the manufacturer data presented to you depending on the app and/or OS or embedded system.
Using “manufacturer data”, all the following conditions must be true:

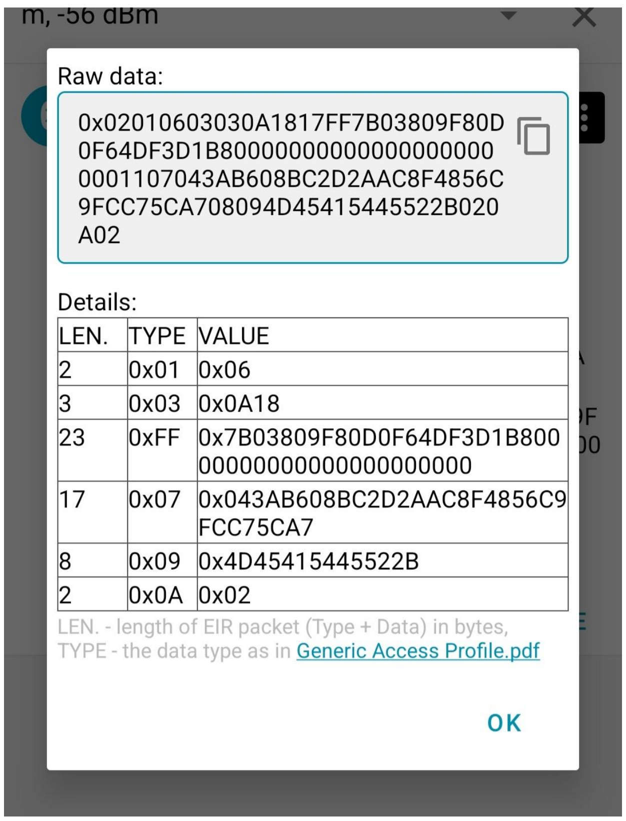
1. MFG==0x037b
   The Advertisement data starts with manufacturer name “apption labs” recognised as 0x037b (2 bytes). This may be presented as “7b03” if considered an array.
2. LENGTH>=11
   Advertisement data payload is of length at at least 11 bytes (can be more). This starts from and includes manufacturer name, 2 bytes as above.
3. PRODUCT==0x80
   Product ID is 0x80, found in the 3rd byte

Here is how this may look like in the Android app “nRF Connect”:



MEATER+
B8:1F:5E:06:AF:38
NOT BONDED -48 dBm ↔157 ms

Device type: UNKNOWN
Advertising type: Legacy
Flags: GeneralDiscoverable, BrEdrNotSupported
Complete list of 16-bit Service UUIDs: 0x180A
Manufacturer data (Bluetooth Core 4.1):
Company: Apption Labs Inc. <0x037B> 0x809F80D0F64DF3D1B80000000000000000000000
Complete list of 128-bit Service UUIDs:
a75cc7fc-c956-488f-ac2a-2dbc08b63a04
Complete Local Name: MEATER+
Tx Power Level: 2 dBm



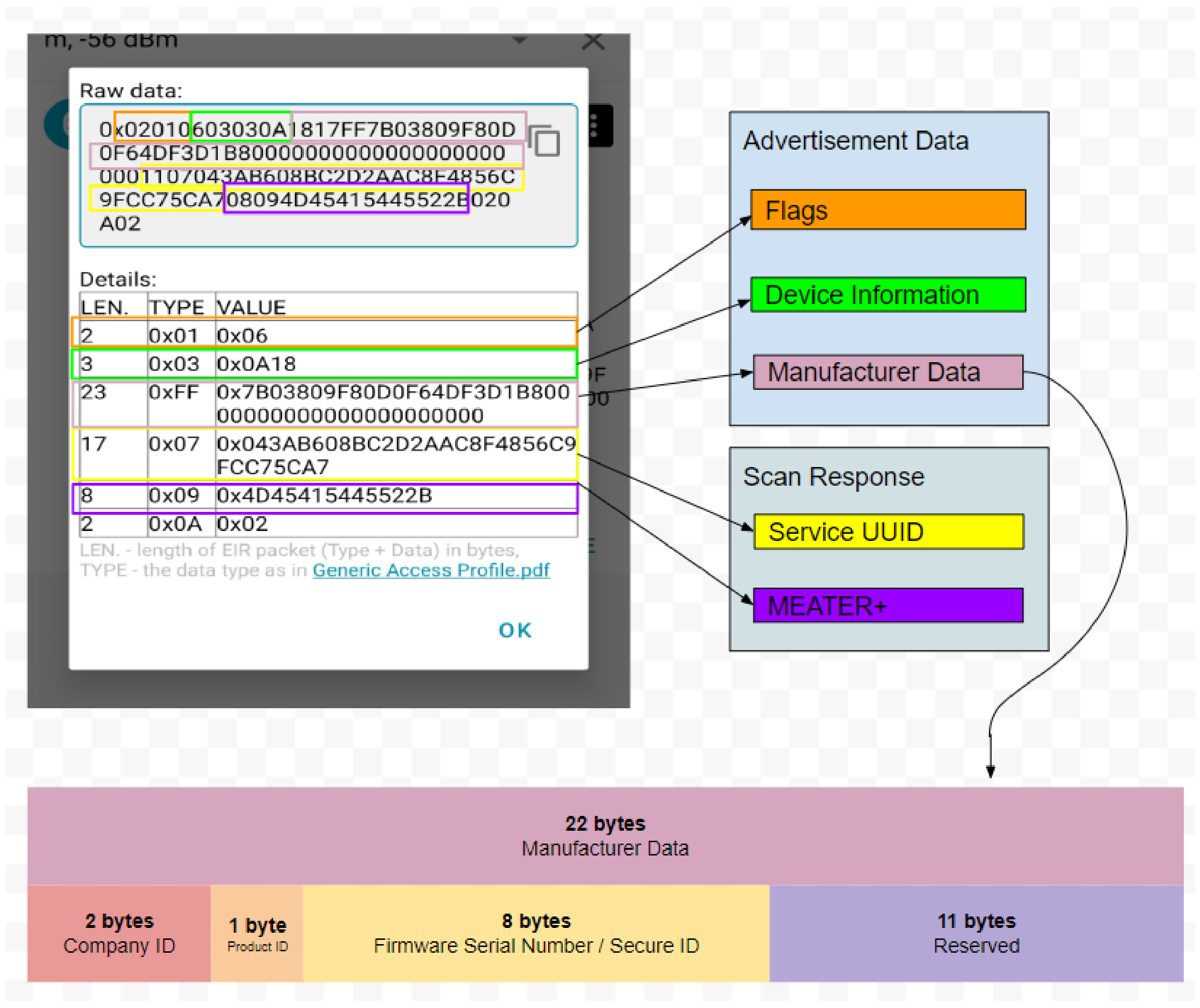
Raw data:

```
0x02010603030A1817FF7B03809F80D
0F64DF3D1B80000000000000000000
0001107043AB608BC2D2AAC8F4856C
9FCC75CA708094D45415445522B020
A02
```

Details:

LEN.	TYPE	VALUE
2	0x01	0x06
3	0x03	0x0A18
23	0xFF	0x7B03809F80D0F64DF3D1B800 00000000000000000000
17	0x07	0x043AB608BC2D2AAC8F4856C9 FCC75CA7
8	0x09	0x4D45415445522B
2	0x0A	0x02

LEN. - length of EIR packet (Type + Data) in bytes,
TYPE - the data type as in Generic Access Profile.pdf



Raw data:

```
0x020106030A1817FF7B03809F80D
0F64DF3D1B8000000000000000000
0001107043AB608BC2D2AAC8F4856C
9FCC75CA708094D4541544552 2B020
A02
```

Details:

LEN.	TYPE	VALUE
2	0x01	0x06
3	0x03	0x0A18
23	0xFF	0x7B03809F80D0F64DF3D1B800 00000000000000000000
17	0x07	0x043AB608BC2D2AAC8F4856C9 FCC75CA7
8	0x09	0x4D45415445522B
2	0x0A	0x02

LEN. - length of EIR packet (Type + Data) in bytes,
TYPE - the data type as in Generic Access Profile.pdf

Connection Parameters

Connection parameters are updated by sending a request to the host processor after connection. For optimal functionality, it is recommended to follow the settings requested by the MEATER+.

If you wish to use fixed connection parameters instead, we recommend a connection interval of 450ms and a connection timeout of 6000ms. A slow connection interval is needed to reach advertised power consumption of the probe.

Services and Characteristics

Device Information Service

UUID: `180A`

Name	UUID	Properties	Notes
Firmware Revision	`2A26`	R	
Hardware Revision	`2A27`	R	

MEATER Service

UUID: `A75CC7FC-C956-488F-AC2A-2DBC08B63A04`

Name	UUID	Properties	Notes
Log Mode	`575D3BF1-2757-45AD-94D9-875C2F6120D3`	R/W	[Advanced] Used for controlling temperature history logging. You can pause, resume and reset the temperature log.
Temperature Reading	`7EDDA774-045E-4BBF-909B-45D1991A2876`	R/N	Live temperature reading. Updated once per second.
Battery Level	`2ADB4877-68D8-4884-BD3C-D83853BF27B8`	R/N	Current battery level of the probe.
Cook Setup	`CAF28E64-3B17-4CB4-BB0A-2EAA33C47AF7`	R/W	A blob of data, not read by the probe. Use this for storing whatever you need to be able to recover all cook properties later.
Temperature History	`B3E02C20-85BE-4D1E-8DA8-30CD88AAF0D4`	R	[Advanced] Temperature history data since the temperature log was last reset.
Probe Information	`1cbff55e-9a06-4`	R	Probe’s Product ID, Secure

	721-a178-1e2d84246dd1		ID, version string.
Charger battery	22db81c4-d125-4e8f-99a4-3609e4c9a017	R/N	Current battery level of the charger.
Charger's Status and Commands	e03c6ccc-2aa7-40a4-8a66-c98b599b737a	R/W/N	Charger Status and Commands.
Probe's RSSI	370aabe7-4837-4bee-aadc-cd1836dbce53	R/N	Probe's RSSI.
Charger's temperature	BB9B2404-FCFB-4B73-8ACD-B1B08DA3749D	R/N	Charger's electronic temperature.
Service Change	0x2A05	R/N	For debug only

Temperature Reading Characteristic

This characteristic has the `read` and `notify` properties. When subscribed to notifications, it is updated every 1 second.

Temperatures are currently returned as 16x Celsius, so for example 10°C would be 160.

The data is formatted using the following C struct:

```
typedef struct {
    uint16_t internal;
    uint16_t ambient;
    uint16_t initialAmbientOffset;
    uint16_t lowestAmbientOffset;
} BLETemperatureReading;
```

The following C code can be used to convert the above data into simple temperature values:

```
typedef int32_t Temperature;

// Use for obtaining a signed value for the internal temperature from
a raw reading from the probe
#define BIT_MASK_12BIT_ALL_ONE 0x0fff
```

```
static inline Temperature
SignedInternalFromTemperatureReading(BLETemperatureReading reading)
{
    int minus1 = -1;
    if (reading.internal > 2048) {
        // 12 bit signed value needs extending to 32bit or whatever
the computer has as "int"
        return (reading.internal & BIT_MASK_12BIT_ALL_ONE) | (minus1
^ BIT_MASK_12BIT_ALL_ONE);
    }
    return reading.internal;
}

// Converts a raw ambient reading to a temperature
static inline Temperature
AmbientOffsetFromTemperatureReading(BLETemperatureReading reading)
{
    return MAX(0, ((reading.ambient - MIN(3 * 16,
reading.initialAmbientOffset)) * 16 * 589) / 1487.0f);
}
```

Then, with those functions in place, you can convert data from the probe:

```
BLETemperatureReading reading = {raw data from the probe}

Temperature internal = SignedInternalFromTemperatureReading(reading);
Temperature ambient =
MAX(0,internal+AmbientOffsetFromTemperatureReading(reading));
```

Depending on the precision you need, you can also use the following functions to convert a raw `Temperature` value to Celsius and Fahrenheit:

```
// Converts a temperature to celsius (int, rounded)
static inline int32_t TemperatureToCelsius(Temperature temperature)
{
    if (temperature >= 0) {
        return (temperature + 8) / 16;
    }
    return (temperature - 8) / 16;
}
```

```
// Converts a temperature to celsius (float)
static inline float TemperatureToCelsiusFloat(Temperature
temperature)
{
    return temperature / 16.0f;
}

// Converts a temperature to fahrenheit (int, rounded)
static inline int32_t TemperatureToFahrenheit(Temperature
temperature)
{
    return ((((temperature * 9 + 3) / 5) / 16) + 32);
}

// Converts a temperature to fahrenheit (float)
static inline float TemperatureToFahrenheitFloat(Temperature
temperature)
{
    return ((temperature / 16.0) * (9 / 5.0)) + 32;
}
```

Cook Setup

MEATER's Cook setup is often subject to change, thus will not be documented here.

Battery Level

Probe battery levels are integer values between 0 and 10, inclusive. For current probes, the curve of battery level is not linear, we suggest:

- 0-5 == 0 bars/empty
- 6 == 1 bar
- 7 == 2 bars
- 8-10 == 3 bars/full

Log Mode

The log mode characteristic is used for pausing, resuming and resetting the temperature log. You'll reset the log when a new cook is started, and you'll pause it before reading the temperature log. Once read, you'll need to resume the temperature log. If you do not resume the temperature log after reading it, it will not continue to update. It is imperative that the temperature log is resumed afterwards.

Reading the temperature log will be covered later in this document.

The log mode values are stored as an enum. You need to send just the value to the log mode characteristic, and the probe will take the appropriate action.

```
typedef enum {
    ProbeTemperatureLogStateReset = 0,
    ProbeTemperatureLogStateResume = 1,
    ProbeTemperatureLogStatePaused = 2
} ProbeTemperatureLogState;
```

Temperature Log

The probe temperature log is always a fixed size of 512 bytes. If you receive 0 bytes from this characteristic, it means that the probe isn't ready and you should try again later.

A temperature log stores a maximum of 120 temperature values to represent the history of the cook since the last time the log was reset. A reset can happen:

- At the start of a cook, when the reset value is sent to the Log Mode characteristic
- When the probe is placed into the charger
- When a probe firmware reset happens

Alongside the temperature values, a temperature log also contains a `scale` and a `count` value.

At the beginning, or immediately after a reset, the scale is set to 5. This means that a temperature is appended to the log every 5 seconds. Once 120 values have been stored (after 10 minutes, or rather `120 values * 5 seconds`), the temperature log is compacted.

Log compaction means that the `scale` value is multiplied by 2, to make 10 seconds. Also, every other temperature value in the log is discarded. This leaves 60 temperatures in the log. A new temperature is appended to the log every 10 seconds.
The next time the temperature log reaches 120 values, it is compacted again. This time, the scale value is multiplied again by 2 (to make 20 seconds), and every other temperature value in the log is discarded.

This process is repeated throughout the cook. It does mean that as cooks get longer, precision is lost. Apps connected over BLE to the probe will also generate their own "high resolution" temperature log, which does not need to be compacted. This allows apps to display logs with better resolution than the probes can support.

Reading the temperature log
You must follow a very specific set of steps when reading the temperature log from the probe.

1. Pause the temperature log
2. Read the temperature log
3. Resume the temperature log

Each step is imperative, and you must perform each one. If you do not pause the temperature log, the log you read will be outdated or corrupted. If you do not resume the temperature log, the probe will not update the log until it is resumed.

The temperature log is represented by the following struct:

```
// A pair of temperatures recorded in the probe's cook temperature
history
typedef struct {
    uint16_t internal; // 16x Celsius format
    uint16_t ambient; // Raw value from the sensor
} BLETempRecording;

// The container structure for the probe's cook temperature history
typedef struct {
    uint32_t unused;
    uint16_t timeScale; // The units between temperature recordings
(defaults to 5, will increase over time)
    uint16_t count; // The number of temperature recordings
    uint16_t initialAmbientReading; // The first ambient raw value
    uint16_t lowestAmbientReading; // The lowest ambient raw value
    BLETempRecording values[120]; // The temperature recordings
    uint32_t padding[5]; // Unused
} BLETempRecordingSet;
```

Probe Information

The Probe Information is structure such as

```
        typedef struct _MeaterPlus_ProbeInfo_t
        {
            uint8_t  probeNumber : 8;
            uint64_t probePairedSerialNumber : 64;
            char     probeVersion[15];
        }MeaterPlus_ProbeInfo_t;
```

`probeNumber` – Probe Number refers to the Product ID. The standard Probe for MEATER+ has Product ID = 0.
`probePairedSerialNumber` – In this instance, “serial number” refers to our Firmware Serial Number that we also call Secure ID. This does not relate with the “Serial Number” printed on the back of the charger. MEATER uses the Secure ID to validate the authenticity of its products.
`probeVersion` – Probe’s firmware version string such as for example “1.2.3_0”.

Charger battery

MEATER+ battery level is normalized integer, 0~100%, mapped to the capacity of a standard Alkaline AAA battery at room temperature ~ 20C.

Charger’s Status and Commands

This characteristic has multiple purpose:
- report whether probe is connected
- status and command during OTA operation

The most simplified Status would be:
Value is 0: Probe is Connected, characteristics related to the Probe can be read and contain valid data.
All other values: Probe is Disconnected, ignore Probe-Info, Probe RSSI, Probe battery which are not valid while probe is disconnected.

Probe’s RSSI

RSSI from Probe to Charger, valid range -126 to -1;
Values out of range such as ‘0’ and ‘-127’ indicate an error and should be treated as ‘unknown’.
RSSI is only valid while the probe is connected, see Charger Status.

Charger’s temperature

The temperature of the charger, purpose is to monitor for overheating if placed near a grill/oven.
Temperatures are currently returned as 16x Celsius, so for example 10°C would be 160.