MikroTik Tag advertisement formats

Introduction

TG-BT5-XX tags can operate in 4 different modes:

1. Factory sleep mode
2. Configuration mode
3. Advertising mode
4. Upgrade mode

In advertising mode, the tag will broadcast information about itself in Bluetooth advertising packets. The information depends on the advertising packet type.

At the moment, these are all the supported types that can be configured using the MikroTik Beacon Manager app:

Eddystone-TLM, Eddystone-UID, MikroTik, and iBeacon.

Bluetooth technology uses 2 types of channels (each using different frequencies) during the data exchange.

1. Data channels dedicated to data transmission
2. Advertise channels dedicated to advertising

There are 40 unique bands (channels) and each band has a 2 MHz separation. 37, 38, and 39 channels are used for advertising, and 0-36 are used for data transmission.

During the advertising process, the BLE advertising packet is broadcasted. This packet contains the Preamble, Access Address, PDU and CRS fields.

The Preamble and Access Address fields help the receiver detect frames. CRS field is used to check errors. PDU defines the packet itself.

MikroTik tags support legacy Non-Connectable Non-Scannable Undirected advertising (`ADV_NOCONN_IND`). The Payload, in this case, consists of "AdvA" (a field that contains information about the advertiser's address) and "AdvData" (a field that contains data information) fields.

1 octet = 1 byte = 8 bits

Preamble	1 octet
Access-Address	4 octets
PDU	• PDU Header = 2 octets • PDU Payload = AdvA (6 octets)+AdvData (0...31 octets)
CRS	3 octets

MikroTik packet structure

"AdvData" field structure (max 31 octets/bytes):

Length	length of the payload	1 octet (15)

Type	manufacturer specific data	1 octet (`ff`)
ManufacturerData	company identifier	2 octets (`4F09`)
Version	the version of this advertisement structure	1 octet (uint)
UserData	user-configured part of the payload	1 octet (uint)
Secret	optionally encrypted (AES-ECB) part of the payload	• secret: salt (for encryption) = 2 octets (uint) • secret: acceleration (acceleration in signed 8.8 fixed point format - acceleration of all 3 axis (0=x, 1=y, 2=z)) = 6 octets (uint) • secret: temperature (ambient temperature in Celsius in signed 8.8 fixed point format) = 2 octet (int) • secret: uptime (uptime in seconds) = 4 octets (uint) • secret: flags (bit-mask of flags) = 1 octet (uint) • secret: batteryPercentage (battery level in percent) = 1 octet (uint)

Please note that all multi-byte values are in little-endian. Meaning, that if, for example, you want to get the temperature value and #14 and #15 octets indicate the temperature as "a1 19" ("plus" temperature) → the real temperature value is going to be (0x19a1)/256 = 25.6 C.

A detailed example of how you can convert **hexadecimal** payload's values to **decimal** values will be shown later in the **PDU payload structure** section.

UserData and Secret fields are configured with the help of `flags`. In the "UserData" section, the parameter that controls whether the "Secret" is encrypted or not is called `FLAG_ENCRYPTED`. When `FLAG_ENCRYPTED=0`, it means the secret is not encrypted (1st bit in 6th octet would be set to 0), and when `FLAG_ENCRYPTED=1`, it means the secret is encrypted (1st bit in 6th octet would be set to 1).

In the "Secret" section, there are 6 `flags` (21st octet):

1. `FLAG_REED_SWITCH` (1st bit - if set to 1, shows that the reed switch was closed at the moment of advertising)
2. `FLAG_ACCEL_TILT` (2nd bit - if set to 1, shows that the advertisement was sent by tilting the device)
3. `FLAG_ACCEL_FREE_FALL` (3d bit - if set to 1, shows that the advertisement was sent by dropping the device)
4. `FLAG_IMPACT_X` (4th bit - if set to 1, shows that there was an impact on the x-axis at the moment of advertising)
5. `FLAG_IMPACT_Y` (5th bit - if set to 1, shows that there was an impact on the y-axis at the moment of advertising)
6. `FLAG_IMPACT_Z` (6th bit - if set to 1, shows that there was an impact on the z-axis at the moment of advertising)

For example, if you see that the 21st octet of the hex message is "02" (when converting the value "02" from **hexadecimal** to **binary** it is "0010" → 2nd bit is set to 1) → it means that the device was tilted. If you see "04" (**hex** "04" to **bin** is "0100" → 3d bit is set to 1) → it means that the device was dropped (freefalling triggered). If you see "38" (**hex** to **bin** it is "00111000" → 4th, 5th, and 6th bits are set to 1) → it means that when the advertisement was sent, the accelerometer detected an impact/wake-up on all 3 x/y/z-axis.

More examples (for 21 octet's value):

- "08" is the impact on just the x-axis;
- "18" is the impact on the x and y-axis;
- "28" is the impact on the x and z-axis;
- "10" is the impact on just the y-axis;
- "30" is the impact on y and z-axis;
- "20" is the impact on just the z-axis.

MikroTik PDU Payload structure

0	15	Length	length of the payload
1	FF	Type	manufacturer specific data
2	4F	Company identifier	MikroTik
3	09	Company identifier	MikroTik
4	01	Version	the version of this advertisement structure
5	00	UserData	user-configured part of the payload

6	xx*	Secret	secret: salt
7	xx*	Secret	secret: salt
8	xx*	Secret	secret: acceleration on the X-axis
9	xx*	Secret	secret: acceleration on the X-axis
10	xx*	Secret	secret: acceleration on the Y-axis
11	xx*	Secret	secret: acceleration on the Y-axis
12	xx*	Secret	secret: acceleration on the Z-axis
13	xx*	Secret	secret: acceleration on the Z-axis
14	xx*	Secret	secret: temperature
15	xx*	Secret	secret: temperature
16	xx*	Secret	secret: uptime
17	xx*	Secret	secret: uptime
18	xx*	Secret	secret: uptime
19	xx*	Secret	secret: uptime
20	00	Secret	secret: flags
21	xx*	Secret	secret: batteryPercentage

- - can vary

Example

An example of the payload configured in MikroTik's format (non-encrypted) would be:

```
15ff4f090100cea6000000000200a01c91085700005f
```

15ff4f09 (first 4 octets) → Length (0x15 **hex-to-dec** is 21). Type (0xff). Company identifier (0x4f09).

01 (4th octet) → Current version of the payload's structure. Should be the same for every payload (constant data).

00 (5th octet) → Indicates that the payload is not encrypted. "01" would mean it is encrypted.

cea6 (6th and 7th octets) → Salt. Each new payload should have a different salt value generated. You can use this value to check whether the identical payloads are encrypted differently. **The value itself does not contain any useful information**. If you see that the salt value is identical for two payloads received during different time intervals, it would mean that the two payloads received are exactly identical. You can calculate the salt value using the same principle that applies to the uptime calculation (17th to 20th octets) - see below.

0000 (8th and 9th octets) → acceleration on the X-axis at the moment of the broadcast = **0 m/s^2**. Check acceleration calculation for the Z-axis below.

0000 (10th and 11th octets) → acceleration on the Y-axis at the moment of the broadcast = **0 m/s^2**. Check acceleration calculation for the Z-axis below.

0200 (12th and 13th octets) → acceleration on the Z-axis at the moment of the broadcast = **0.0078 m/s^2**. To get the decimal value out of the hex format you will need to follow the steps:

- As noted before, multi-byte values are in little-endian and that means, to calculate the realm value, you will need to switch octets places (switch octets order). So the first step is to swap places for the values from 0x0200 to 0x0002. 0x0002 converted from **hexadecimal** to **decimal** is 02.
- Keep in mind that acceleration is in signed 8.8 fixed point format (two's complement) and that means that you basically need to divide the result by "256". The second step is to divide the value by 256 → (0x0002 **hex** or 02 **dec**)/256 = 0.0078 m/s^2.
- The same calculation principle applies to the acceleration for the X and Y-axis. In our example, they just happen to be 0 → 0x0000/256=0.

a01c (14th and 15th octets) → temperature detected by the tag in Celsius = **28.625 C**. Temperature is in little-endian (as it is a multi-byte value) and it is in signed 16-bit integer [twos complement] 8.8 fixed point format, so the same "formula" applies here as well:

- 0x1ca0/256=28.625 C.

91085700 (16th to 19th octets) → tag's uptime in seconds = **5703825 s**. 0x91085700 is in little-endian, so just swap the octets to 0x00570891 and the result is 5703825 in decimal. That is 1584.395833 hours or 66-day uptime.

00 (20th octet) → trigger (flag) that sent the payload. If it is "**00**" it means that no trigger was detected and that it is just a periodically broadcasted payload (based on the advertisement interval configured for the tag). If the value would be "**04**" it would mean that the device was dropped (freefalling triggered). You can find more information on the "flags" and the "Secret" section above in the **packet structure** section.

5f (21st octet) → battery percentage of the tag = **95 %**. 0x**5f** from hex to dec is 95.

Starting with v**7.11**, you can use the Peripheral Device section or/and Decode-ad feature to view decoded values!

Script for decoding

Add a new script under the "**System>Scripts**" tab and import the script there (for non-encrypted payloads).

```
# POSIX regex for filtering advertisement Bluetooth addresses. E.g. "^BC:33:AC"
# would only include addresses which start with those 3 octets.
# To disable this filter, set it to ""
:local addressRegex "2C:C8:1B:4B:BB:0A"

# POSIX regex for filtering Bluetooth advertisements based on their data. Same
# usage as with 'addressRegex'.
:local advertisingDataRegex ""

# Signal strength filter. E.g. -40 would only include Bluetooth advertisements
# whose signal strength is stronger than -40dBm.
# To disable this filter, set it to ""
:local rssiThreshold ""

################################## Bluetooth ##################################
:global invertU16 do={
    :local inverted 0
    :for idx from=0 to=15 step=1 do={
        :local mask (1 << $idx)
        :if ($1 & $mask = 0) do={
            :set $inverted ($inverted | $mask)
        }
    }
    return $inverted
}

:global le16ToHost do={
    :local lsb [:pick $1 0 2]
    :local msb [:pick $1 2 4]

    :return [:tonum "0x$msb$lsb"]
}

:local le32ToHost do={
    :local lsb [:pick $1 0 2]
    :local midL [:pick $1 2 4]
    :local midH [:pick $1 4 6]
    :local msb [:pick $1 6 8]

    :return [:tonum "0x$msb$midH$midL$lsb"]
}

:local from88 do={
    :global invertU16
    :global le16ToHost
    :local num [$le16ToHost $1]

    # Handle negative numbers
    :if ($num & 0x8000) do={
        :set num (-1 * ([$invertU16 $num] + 1))
    }
    # Convert from 8.8. Scale by 1000 since floating point is not supported
    :return (($num * 125) / 32)
}

:local flagStr do={
    :local str ""

    :if ($1 & 0x01) do={ :set $str " switch" }
    :if ($1 & 0x02) do={ :set $str "$str tilt" }
    :if ($1 & 0x04) do={ :set $str "$str free_fall" }
    :if ($1 & 0x08) do={ :set $str "$str impact_x" }
    :if ($1 & 0x10) do={ :set $str "$str impact_y" }
    :if ($1 & 0x20) do={ :set $str "$str impact_z" }

    :if ([:len $str] = 0) do={ :return "" }

    :return [:pick $str 1 [:len $str]]
}

# Find fresh Bluetooth advertisements
:global btOldestAdvertisementTimestamp
:if ([:typeof $btOldestAdvertisementTimestamp] = "nothing") do={
    # First time this script has been run since booting, need to initialize
    # persistent variables
    :set $btOldestAdvertisementTimestamp 0
}
:local advertisements [/iot bluetooth scanners advertisements print detail \
    as-value where \
        epoch > $btOldestAdvertisementTimestamp and \
        address ~ $addressRegex and \
        data ~ $advertisingDataRegex and \
        rssi > $rssiThreshold
]
:local advCount 0
:local lastAdvTimestamp 0
:local advJson ""
:local advSeparator ""

# Remove semicolons from MAC/Bluetooth addresses
:local minimizeMac do={
    :local minimized
    :local lastIdx ([:len $address] - 1)
    :for idx from=0 to=$lastIdx step=1 do={
        :local char [:pick $address $idx]
        :if ($char != ":") do={
            :set $minimized "$minimized$char"
        }
    }
    :return $minimized
}

:foreach adv in=$advertisements do={
    :local address ($adv->"address")
    :local rssi ($adv->"rssi")
    :local epoch ($adv->"epoch")
    :local ad ($adv->"data")
    :local version [:tonum "0x$[:pick $ad 8 10]"]
    :local encrypted [:tonum "0x$[:pick $ad 10 12]"]
    :local salt [$le16ToHost [:pick $ad 12 16]]
    :local accelX [$from88 [:pick $ad 16 20]]
    :local accelY [$from88 [:pick $ad 20 24]]
    :local accelZ [$from88 [:pick $ad 24 28]]
    :local temp [$from88 [:pick $ad 28 32]]
    :local uptime [$le32ToHost [:pick $ad 32 40]]
    :local flags [:tonum "0x$[:pick $ad 40 42]"]
    :local bat [:tonum "0x$[:pick $ad 42 44]"]
    :put ("$advCount: \
        address=$address \
        ts=$epoch \
        rssi=$rssi \
        version=$version \
        encrypted=$encrypted \
        salt=$salt \
        accelX=$accelX \
        accelY=$accelY \
        accelZ=$accelZ \
        temp=$temp \
        uptime=$uptime \
        flags=\"$[$flagStr $flags]\" \
        bat=$bat" \
    )
    :set $advCount ($advCount + 1)
    :set $lastAdvTimestamp $epoch
}
:if ($advCount > 0) do={
    :set $btOldestAdvertisementTimestamp $lastAdvTimestamp
}
```

The only line that you need to alter is the:

```
:local addressRegex "2C:C8:1B:4B:BB:0A"
```

line, where you need to input the MAC address of the tag.

Save the script with whichever name you like, for example, **decode**.

Run the script via the command line interface ("**New Terminal**" button in Winbox/Webfig):

Example

```
[admin@MikroTik] > system script run decode
0: address=2C:C8:1B:4B:BB:0A ts=1662553431348 rssi=-45 version=1 encrypted=0 salt=57919 accelX=3 accelY=-35
accelZ=-70 temp=25535 uptime=1046174 flags="" bat=99
1: address=2C:C8:1B:4B:BB:0A ts=1662553436349 rssi=-40 version=1 encrypted=0 salt=24154 accelX=-19 accelY=-23
accelZ=0 temp=25546 uptime=1046179 flags="" bat=99
2: address=2C:C8:1B:4B:BB:0A ts=1662553446351 rssi=-37 version=1 encrypted=0 salt=37822 accelX=-15 accelY=35
accelZ=15 temp=25550 uptime=1046189 flags="" bat=99
```

As you can see from the example above, the script will "translate" all payloads from a **hexadecimal** format to a **decimal** format and print them into the terminal.

You can also alter the script further to structure a message out of the "already decoded" values and post it to an EMAIL, MQTT, or HTTP server of your choice **but!** please keep in mind that it might load the device more. So you need to test the performance when running the script. It will be easier on RouterOS resources when the decoding is done on the server side.

ⓘ Because of the fact that floating point is not supported → every calculation behind a decimal point will be "rounded up" to a whole number. This is why the script will calculate the temperature and acceleration values **scaled by 1000** (multiplied by **1000**).
So, if you see the temperature as **temp=25546**, the real temperature is **25.546 C** (25546/1000) and if you see **accelZ=15**, the real acceleration against the z-axis will be **0.015** m/s^2 (15/1000).

iBeacon packet structure

iBeacon is one of the supported advertising packet types. You can find more information about the protocol following the link.

The PDU Payload, in this case, consists of "AdvA" (that is 6 octets long) and "AdvData" (a field that contains data information) fields. Legacy Bluetooth devices can only support 31 byte-long beacon messages. UUID is 16 byte-long (MikroTik default UID=b2b98de4-c81c-47c2-b14e-791b3e5587ec).

"AdvData" field structure:

ManufacturerData	company identifier	4 octets (`1aff4c00`)
BeaconType	a secondary identifier	1 octet (const)
RemainingDataLength	defines the remaining length for the payload in bytes	1 octet (const)
UserData	user-configured part of the payload	• Proximity UUID (universally unique identifier) = 16 octets (uint) • Major Number (specific group identifier) = 2 octets (uint) • Minor Number (specific beacon identifier) = 2 octets (uint)
TxPower	indicates the signal strength at one meter from the device	1 octet (int)

iBeacon PDU Payload structure

0	`1a`	ManufacturerData	company identifier
1	ff	ManufacturerData	company identifier
2	4c	ManufacturerData	company identifier
3	00	ManufacturerData	company identifier
4	02	BeaconType	a secondary identifier
5	21	RemainingDataLength	defines the remaining length for the payload in bytes
6	xx*	UserData	Proximity UUID
7	xx*	UserData	Proximity UUID
8	xx*	UserData	Proximity UUID
9	xx*	UserData	Proximity UUID
10	xx*	UserData	Proximity UUID
11	xx*	UserData	Proximity UUID
12	xx*	UserData	Proximity UUID
13	xx*	UserData	Proximity UUID
14	xx*	UserData	Proximity UUID
15	xx*	UserData	Proximity UUID
16	xx*	UserData	Proximity UUID
17	xx*	UserData	Proximity UUID
18	xx*	UserData	Proximity UUID
19	xx*	UserData	Proximity UUID
20	xx*	UserData	Proximity UUID
21	xx*	UserData	Proximity UUID
22	xx*	UserData	Major Number
23	xx*	UserData	Major Number
24	xx*	UserData	Minor Number
25	xx*	UserData	Minor Number
26	xx*	TxPower	indicates the signal strength at one meter from the device

* - can vary

Eddystone-TLM packet structure

Eddystone-TLM is one of the supported advertising packet types. You can find more information about the protocol following the link.

The PDU Payload, in this case, consists of "AdvA" (that is 6 octets long) and "AdvData" (a field that contains data information) fields. MikroTik default CompleteUUID=03 03 aa fe; ServiceData=11 16 aa fe.

"AdvData" field structure:

CommonPayload	part of the advertisement payload that is common for all Eddystone's frame types	- CompleteUUID (universally unique identifier) = 4 octets (const) - ServiceData (*16 bit UUID* data type) = 4 octets (const) - FrameType (Value = `0x20`) = 1 octet (const)
TlmPayload	Eddystone-TLM frame payload	- Version (TLM version) = 1 octet (const) - BatteryVoltageMv (Battery voltage, 1 mV/bit) = 2 octets (uint) - TemperatureC (Beacon temperature in Celsius) = 2 octets (int) - AdvertisementCount (Advertising PDU count) = 4 octets (uint) - UptimeCounter (Time since power-on or reboot) = 4 octets (uint)

Eddystone-TLM PDU Payload structure

0	`03`	CommonPayload	CompleteUUID
1	03	CommonPayload	CompleteUUID
2	aa	CommonPayload	CompleteUUID
3	fe	CommonPayload	CompleteUUID
4	11	CommonPayload	ServiceData
5	16	CommonPayload	ServiceData
6	aa	CommonPayload	ServiceData
7	fe	CommonPayload	ServiceData
8	20	CommonPayload	FrameType
9	00	TlmPayload	Version
10	xx*	TlmPayload	BatteryVoltageMv
11	xx*	TlmPayload	BatteryVoltageMv
12	xx*	TlmPayload	TemperatureC
13	xx*	TlmPayload	TemperatureC
14	xx*	TlmPayload	AdvertisementCount
15	xx*	TlmPayload	AdvertisementCount
16	xx*	TlmPayload	AdvertisementCount
17	xx*	TlmPayload	AdvertisementCount
18	xx*	TlmPayload	UptimeCounter
19	xx*	TlmPayload	UptimeCounter
20	xx*	TlmPayload	UptimeCounter
21	xx*	TlmPayload	UptimeCounter

* - can vary

Eddystone-UID packet structure

Eddystone-UID is one of the supported advertising packet types. You can find more information about the protocol following the link.

The PDU Payload, in this case, consists of "AdvA" (that is 6 octets long) and "AdvData" (a field that contains data information) fields. MikroTik default CompleteUUID=03 03 aa fe; ServiceData=17 16 aa fe.

"AdvData" field structure:

CommonPayload	part of the advertisement payload that is common for all Eddystone's frame types	• CompleteUUID (universally unique identifier) = 4 octets (const) • ServiceData (*16 bit UUID* data type) = 4 octets (const) • FrameType (value = `0x00`) = 1 octet (const)
UidPayload	Eddystone-UID frame payload	• Ranging Data (calibrated Tx power at 0 m) = 1 octet (int) • Nspace (unique self-assigned beacon ID namespace) = 10 octets (uint) • Instance (unique ID within the namespace) = 6 octets (uint) • RFU1 (reserved for future use, value=`0x00`) = 1 octet (const) • RFU2 (reserved for future use, value=`0x00`) = 1 octet (const)

Eddystone-UID PDU Payload structure

0	`03`	CommonPayload	CompleteUUID
1	03	CommonPayload	CompleteUUID
2	aa	CommonPayload	CompleteUUID
3	fe	CommonPayload	CompleteUUID
4	17	CommonPayload	ServiceData
5	16	CommonPayload	ServiceData
6	aa	CommonPayload	ServiceData
7	fe	CommonPayload	ServiceData
8	00	CommonPayload	FrameType
9	xx*	UidPayload	Ranging Data
10	xx*	UidPayload	Nspace
11	xx*	UidPayload	Nspace
12	xx*	UidPayload	Nspace
13	xx*	UidPayload	Nspace
14	xx*	UidPayload	Nspace
15	xx*	UidPayload	Nspace
16	xx*	UidPayload	Nspace
17	xx*	UidPayload	Nspace
18	xx*	UidPayload	Nspace
19	xx*	UidPayload	Nspace
20	xx*	UidPayload	Instance
21	xx*	UidPayload	Instance

22	xx*	UidPayload	Instance
23	xx*	UidPayload	Instance
24	xx*	UidPayload	Instance
25	xx*	UidPayload	Instance
26	00	UidPayload	RFU1
27	00	UidPayload	RFU2

* - can vary