Testing with payload decoder

For testing with a payload decoder of a different project group, some settings need to be changed. Step 1: In the thingsnetwork cloud, go to payload formatters -> uplink. (<https://eu1.cloud.thethings.network/console/applications/lorahan/payload-formatters/uplink>). Select **“Custom Javascript formatter”**. In formatter code, paste in the decoder code of a different group, in our case the **“decoder\_CayenneLPP\_extreme.js”** file from [GitHub - HAN-IoT-LAB/TTN-Decoder: TheThingsNetwork decoder which is linked to the refactored CayenneLPP encoder.](https://github.com/HAN-IoT-LAB/TTN-Decoder) Save your changes with the blue button.

!! Delete the very last line of this file, which is: “module.exports = { SensorTypes, decodeUplink };”.

Or copy the decoder text which is located at the end of this file.

Step 2: Go to End devices -> (device) -> payload formatters. In uplink, select Custom Javascript formatter, scroll down and click ‘paste application formatter’. This will copy the same file as above to the formatter code.

!! Check again if the very last line is not being displayed, otherwise go back to step 1.

To test the other example codes, reset all the settings to what is was before:

1. In the thingsnetwork cloud, go to payload formatters -> uplink. Select “**none”.** Save changes
2. In the thingsnetwork cloud, go to End devices -> (device) -> payload formatters. In uplink, select **“CayenneLPP”.** Save changes.

Decoder file

/\* This code is free software:

\* you can redistribute it and/or modify it under the terms of a Creative

\* Commons Attribution-NonCommercial 4.0 International License

\* (http://creativecommons.org/licenses/by-nc/4.0/)

\*

\* Copyright (c) 2024 March by Klaasjan Wagenaar, Tristan Bosveld and Richard Kroesen

\*/

/\*\*

\* @brief Definitions for sensor types used in IoT device data decoding.

\*

\* This file contains the `SensorTypes` constant, which is a mapping of various sensor types

\* to their respective characteristics. Each sensor type is an object with the following properties:

\* - `type`: Numeric identifier for the sensor type.

\* - `precision`: The factor by which the raw sensor data is divided to obtain a meaningful value.

\* - `signed`: Boolean indicating if the sensor data is signed (true) or unsigned (false).

\* - `bytes`: The number of bytes that represent the sensor's data in the transmission.

\*/

const SensorTypes = {

DIG\_IN: { type: 0, precision: 1, signed: false, bytes: 1 },

DIG\_OUT: { type: 1, precision: 1, signed: false, bytes: 1 },

ANL\_IN: { type: 2, precision: 100, signed: true, bytes: 2 },

ANL\_OUT: { type: 3, precision: 100, signed: true, bytes: 2 },

ILLUM\_SENS: { type: 101, precision: 1, signed: false, bytes: 2 },

PRSNC\_SENS: { type: 102, precision: 1, signed: false, bytes: 1 },

TEMP\_SENS: { type: 103, precision: 10, signed: true, bytes: 2 },

HUM\_SENS: { type: 104, precision: 10, signed: false, bytes: 2 },

ACCRM\_SENS: { type: 113, precision: 1000, signed: true, bytes: 6 },

BARO\_SENS: { type: 115, precision: 10, signed: false, bytes: 2 },

GYRO\_SENS: { type: 134, precision: 100, signed: true, bytes: 6 },

GPS\_LOC: { type: 136, precision: 10000, signed: true, bytes: 12 }

};

/\*\*

\* @SensorTypes Reference Table:

\*

\* | Sensor Name | LPP | IPSO | Decimal Precision | Signed | Size Bytes |

\* |-------------|------|--------|--------------------|--------|--------------|

\* | DIG\_IN | 0 | 3200 | 1 | false | 1 |

\* | DIG\_OUT | 1 | 3201 | 1 | false | 1 |

\* | ANL\_IN | 2 | 3202 | 100 | true | 2 |

\* | ANL\_OUT | 3 | 3203 | 100 | true | 2 |

\* | ILLUM\_SENS | 101 | 3301 | 1 | false | 2 |

\* | PRSNC\_SENS | 102 | 3302 | 1 | false | 1 |

\* | TEMP\_SENS | 103 | 3303 | 10 | true | 2 |

\* | HUM\_SENS | 104 | 3304 | 10 | false | 2 |

\* | ACCRM\_SENS | 113 | 3313 | 1000 | true | 6 |

\* | BARO\_SENS | 115 | 3315 | 10 | false | 2 |

\* | GYRO\_SENS | 134 | 3334 | 100 | true | 6 |

\* | GPS\_LOC | 136 | 3336 | 10000 | true | 12 |

\*/

/\*\*

\* @brief Decodes the uplink data payload based on the specified payload version.

\*

\* This function decodes the input payload by examining the payload version. For encoding version 1.

\*

\* @param input A structure containing the payload to be decoded.

\* @return Returns an object containing the decoded data, the version of the coding used,

\* and arrays for warnings and errors. The returned object has the following structure:

\* {

\* decoder\_version: <version>, // Integer representing the payload version

\* data: <decoded\_data>, // Object containing the decoded payload

\* warnings: <warnings\_array>, // Array of strings representing any warnings

\* errors: <errors\_array>, // Array of strings representing any errors encountered

\* }

\*/

function decodeUplink(input) {

let bytes = input.bytes;

let payload\_version = input.fPort;

let decoded = {};

if (payload\_version == 1) {

decoded = processPayloadVersion\_ONE(bytes, decoded);

} else {

decoded.errors = ["Payload Version not supported: " + payload\_version];

}

return {

decoder\_version: payload\_version,

data: decoded,

warnings: [],

errors: [],

};

}

/\*\*

\* @brief Decodes a value from a byte array based on the specified parameters.

\*

\* This function extracts a value starting from index `i` in the `bytes` array, interpreting

\* a sequence of `byteLength` bytes according to whether the value is signed or unsigned,

\* and then adjusting it by a given `precision`.

\*

\* @param bytes The array of bytes from which the value is to be extracted.

\* @param i The starting index in the `bytes` array from which to begin decoding the value.

\* @param isSigned A boolean indicating whether the value to be decoded is signed (true) or unsigned (false).

\* @param precision The factor by which the raw decoded value should be divided to obtain the final value.

\* This allows for the representation of fractional values without using floating point numbers

\* in the encoded data.

\* @param byteLength The number of bytes that make up the value to be decoded.

\*

\* @return An object containing two properties: `value` and `index`. `value` is the decoded number adjusted

\* by the `precision`, and `index` is the new index in the `bytes` array after decoding the value,

\* which can be used for subsequent decoding operations.

\*/

function decodeValue(bytes, i, isSigned, precision, byteLength) {

let value = 0;

for (let byteIndex = byteLength - 1; byteIndex >= 0; byteIndex--) {

value = (value << 8) | bytes[i+byteIndex];

}

i += byteLength;

if (isSigned && (value & (1 << (8 \* byteLength - 1)))) {

value = value - (1 << (8 \* byteLength));

}

return { value: value / precision, index: i };

}

/\*\*

\* @brief Processes and decodes payload version 1.

\*

\* This function iterates through the bytes of the payload, decoding each segment according to the sensor type

\* it represents. The decoded values are then added to the `decoded` object with keys

\* representing the sensor type and channel.

\*

\* @param bytes An array of bytes representing the payload to be decoded.

\* @param decoded An initially empty object that will be populated with the decoded sensor values.

\* @return Returns the `decoded` object populated with keys and values representing the decoded sensor data.

\*

\*/

function processPayloadVersion\_ONE(bytes, decoded) {

for (let i = 0; i < bytes.length;) {

let type = bytes[i++];

let channel = bytes[i++];

let decodeResult;

switch (type) {

case SensorTypes.DIG\_IN.type:

decodeResult = decodeValue(bytes, i, SensorTypes.DIG\_IN.signed,

SensorTypes.DIG\_IN.precision, SensorTypes.DIG\_IN.bytes);

i = decodeResult.index;

decoded['digital\_' + channel] = decodeResult.value;

break;

case SensorTypes.DIG\_OUT.type:

decodeResult = decodeValue(bytes, i, SensorTypes.DIG\_OUT.signed,

SensorTypes.DIG\_OUT.precision, SensorTypes.DIG\_OUT.bytes);

i = decodeResult.index; // Update index

decoded['digital\_' + channel] = decodeResult.value;

break;

case SensorTypes.ANL\_IN.type:

decodeResult = decodeValue(bytes, i, SensorTypes.ANL\_IN.signed,

SensorTypes.ANL\_IN.precision, SensorTypes.ANL\_IN.bytes);

i = decodeResult.index;

decoded['analog\_' + channel] = decodeResult.value;

break;

case SensorTypes.ANL\_OUT.type:

decodeResult = decodeValue(bytes, i, SensorTypes.ANL\_OUT.signed,

SensorTypes.ANL\_OUT.precision, SensorTypes.ANL\_OUT.bytes);

i = decodeResult.index;

decoded['analog\_' + channel] = decodeResult.value;

break;

case SensorTypes.ILLUM\_SENS.type:

decodeResult = decodeValue(bytes, i, SensorTypes.ILLUM\_SENS.signed,

SensorTypes.ILLUM\_SENS.precision, SensorTypes.ILLUM\_SENS.bytes);

i = decodeResult.index;

decoded['illumination\_' + channel] = decodeResult.value;

break;

case SensorTypes.PRSNC\_SENS.type:

decodeResult = decodeValue(bytes, i, SensorTypes.PRSNC\_SENS.signed,

SensorTypes.PRSNC\_SENS.precision, SensorTypes.PRSNC\_SENS.bytes);

i = decodeResult.index;

decoded['presence\_' + channel] = decodeResult.value;

break;

case SensorTypes.TEMP\_SENS.type:

decodeResult = decodeValue(bytes, i, SensorTypes.TEMP\_SENS.signed,

SensorTypes.TEMP\_SENS.precision, SensorTypes.TEMP\_SENS.bytes);

i = decodeResult.index;

decoded['temperature\_' + channel] = decodeResult.value;

break;

case SensorTypes.HUM\_SENS.type:

decodeResult = decodeValue(bytes, i, SensorTypes.HUM\_SENS.signed,

SensorTypes.HUM\_SENS.precision, SensorTypes.HUM\_SENS.bytes);

i = decodeResult.index;

decoded['humidity\_' + channel] = decodeResult.value;

break;

case SensorTypes.ACCRM\_SENS.type:

let accDecodeX = decodeValue(bytes, i, SensorTypes.ACCRM\_SENS.signed, SensorTypes.ACCRM\_SENS.precision, SensorTypes.ACCRM\_SENS.bytes / 3);

i = accDecodeX.index;

let accDecodeY = decodeValue(bytes, i, SensorTypes.ACCRM\_SENS.signed, SensorTypes.ACCRM\_SENS.precision, SensorTypes.ACCRM\_SENS.bytes / 3);

i = accDecodeY.index;

let accDecodeZ = decodeValue(bytes, i, SensorTypes.ACCRM\_SENS.signed, SensorTypes.ACCRM\_SENS.precision, SensorTypes.ACCRM\_SENS.bytes / 3);

i = accDecodeZ.index;

decoded['accelerometer\_' + channel] = {

x: accDecodeX.value,

y: accDecodeY.value,

z: accDecodeZ.value

};

break;

case SensorTypes.BARO\_SENS.type:

let baroDecode = decodeValue(bytes, i, SensorTypes.BARO\_SENS.signed, SensorTypes.BARO\_SENS.precision, SensorTypes.BARO\_SENS.bytes);

i = baroDecode.index;

decoded['barometer\_' + channel] = baroDecode.value;

break;

case SensorTypes.GYRO\_SENS.type:

let gyroDecodeX = decodeValue(bytes, i, SensorTypes.GYRO\_SENS.signed, SensorTypes.GYRO\_SENS.precision, SensorTypes.GYRO\_SENS.bytes / 3);

i = gyroDecodeX.index;

let gyroDecodeY = decodeValue(bytes, i, SensorTypes.GYRO\_SENS.signed, SensorTypes.GYRO\_SENS.precision, SensorTypes.GYRO\_SENS.bytes / 3);

i = gyroDecodeY.index;

let gyroDecodeZ = decodeValue(bytes, i, SensorTypes.GYRO\_SENS.signed, SensorTypes.GYRO\_SENS.precision, SensorTypes.GYRO\_SENS.bytes / 3);

i = gyroDecodeZ.index;

decoded['gyroscope\_' + channel] = {

x: gyroDecodeX.value,

y: gyroDecodeY.value,

z: gyroDecodeZ.value

};

break;

case SensorTypes.GPS\_LOC.type:

let gpsDecodeX = decodeValue(bytes, i, SensorTypes.GPS\_LOC.signed, SensorTypes.GPS\_LOC.precision, SensorTypes.GPS\_LOC.bytes / 3);

i = gpsDecodeX.index;

let gpsDecodeY = decodeValue(bytes, i, SensorTypes.GPS\_LOC.signed, SensorTypes.GPS\_LOC.precision, SensorTypes.GPS\_LOC.bytes / 3);

i = gpsDecodeY.index;

let gpsDecodeZ = decodeValue(bytes, i, SensorTypes.GPS\_LOC.signed, SensorTypes.GPS\_LOC.precision / 100, SensorTypes.GPS\_LOC.bytes / 3);

i = gpsDecodeZ.index;

decoded['gps\_' + channel] = {

x: gpsDecodeX.value,

y: gpsDecodeY.value,

z: gpsDecodeZ.value

};

break;

default: // Unknown data type

decoded.errors = ["Unknown type: " + type];

i = bytes.length; // Skip the rest of the payload

break;

}

}

return decoded;

}