uRADMonitor HW106 motherboard

This document describes the uRADMonitor INDUSTRIAL data payload structure for direct data access.

Version

- The data structure corresponds to version HW106

Applications

- Decentralized monitoring
- Private monitoring networks
- LoRaWAN payload decoding
- Off grid monitoring

Description

uRADMonitor INDUSTRIAL is an automated, fixed monitoring station with an array of sensors that tracks a total of 12 important environmental parameters. The hardware version HW108 measures Temperature, Barometric pressure, Relative Humidity, Volatile organic compounds (VOC), Particulate matter PM1, PM2.5, PM10, Ozone O_3, Carbon Monoxide CO, Nitrogen Dioxide NO_2, Sulphur Dioxide SO_2, Noise level. The values are packed in a so called payload, a sequence of bytes that uses minimal bandwidth so it can be used across a multitude of networks including LoRaWAN. The payload is machine friendly, but not plain text. So using it outside the uRADMonitor Network and API requires knowing its structure in order to interpret it correctly.

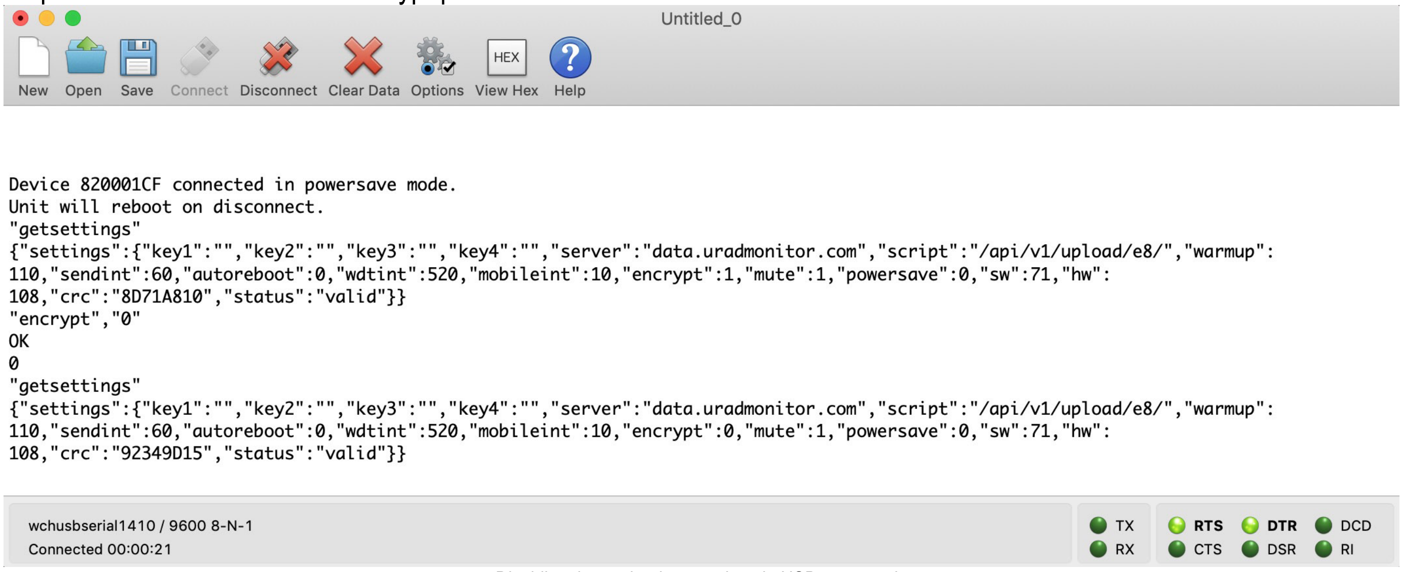
This document presents the payload decoding mechanisms.

Payload Encryption

To protect the integrity of the uRADMonitor Network, the data is encrypted before it is transmitted from the uRADMonitor hardware device to the server. The encryption includes a checksum and a timestamp so it can assure protection against invalid data injection and also repeated injection attacks. The uRADMonitor Data server receives the encrypted payloads then decrypts them before making the data available via the uRADMonitor API.

For Direct Data Access, the payload encryption on the uRADMonitor INDUSTRIAL must first be disabled via USB commands because the decryption algorithm is not publicly available. Once the encryption is disabled, the payload can be decoded as further explained in this document, but the uRADMonitor server will no longer accept the data sent by the device.

To disable payload encryption, connect to the uRADMonitor device via USB (baurate 9600bps) and send the following command: "encrypt","0" . Then check that the change is in place with a second command: "getsettings" . The returned response must show that the encrypt parameter is now 0:



```
Device 820001CF connected in powersave mode.
Unit will reboot on disconnect.
"getsettings"
{"settings":{"key1":"","key2":"","key3":"","key4":"","server":"data.uradmonitor.com","script":"/api/v1/upload/e8/","warmup":
110,"sendint":60,"autoreboot":0,"wdtint":520,"mobileint":10,"encrypt":1,"mute":1,"powersave":0,"sw":71,"hw":
108,"crc":"8D71A810","status":"valid"}}
"encrypt","0"
OK
0
"getsettings"
{"settings":{"key1":"","key2":"","key3":"","key4":"","server":"data.uradmonitor.com","script":"/api/v1/upload/e8/","warmup":
110,"sendint":60,"autoreboot":0,"wdtint":520,"mobileint":10,"encrypt":0,"mute":1,"powersave":0,"sw":71,"hw":
108,"crc":"92349D15","status":"valid"}}
```

wchusbserial1410 / 9600 8-N-1
Connected 00:00:21

Disabling the payload encryption via USB commands

The results are immediate. The encrypted data payload on the INDUSTRIAL HW106 starts with 0065 and has 49bytes, while the plain payload will start with the device ID and only has 47 bytes:

```
006586AF78E80ED1D46F4A87F0D7400C479D220A68EC5F87D496EB1DE02E9AC3D25A720D5C1C601BBB8939E7C5A93C6111
```

Encrypted payload

```
14000055006A00460000003207108BCF1C0D00016A421759512A00D42C00662B0002040214000D0011001362CD3938
```

Non-encrypted payload

LoRaWAN Network Server data

The INDUSTRIAL HW106 with LoRaWAN connectivity will send the payloads to the Network server. If encryption is disabled as presented above, the plain payload will be accessible directly on the Network Server. The next step is to interpret the packed data and extract all measurements.

Payload structure and decoding

The payload is a packed byte sequence, optimized for low bandwidth. We will take the above plain text payload as an example:

```
14000055006A00460000003207108BCF1C0D00016A421759512A00D42C00662B0002040214000D0011001362CD3938
```

In order, the structure is as follows:

```
ID 4B - "14000055"
VERHW 2B - "006A" (decimal 106, meaning INDUSTRIAL HW106)
VERSW 2B - "0046" (decimal 70)
TIMESTAMP 4B - "00000032" (decimal 50 sec)
TEMPERATURE 2B - "0710" DI encoded *
PRESSURE 2B - "8BCF" offset encoded **
HUMIDITY 2B - "1C0D" DI encoded *
VOC 4B - "00016A42" (decimal 92738ohms, or 92KO)
NOISE 2B - "1759" DI encoded *
GASREADTYPE 1B - "51" (binary 01010001) ***
SENSORGASID1 1B - "2A" (Ozone: O3=0x2A, NO2=0x2C, SO2=0x2B, CO=0x04, H2S=0x03, NH3=0x02, CL2=0x31)
SENSORGASCONC1 2B - "00D4" ***
SENSORGASID2 1B - "2C" (Nitrogen Dioxide: O3=0x2A,NO2=0x2C, SO2=0x2B, CO=0x04, H2S=0x03, NH3=0x02, CL2=0x31)
SENSORGASCONC2 2B - "0066" ***
SENSORGASID3 1B - "2B" (Sulphur Dioxide: O3=0x2A, NO2=0x2C, SO2=0x2B, CO=0x04, H2S=0x03, NH3=0x02, CL2=0x31)
SENSORGASCONC3 2B - "0002" ***
SENSORGASID4 1B - "04"  (Carbon Monoxide: O3=0x2A,NO2=0x2C, SO2=0x2B, CO=0x04, H2S=0x03, NH3=0x02, CL2=0x31)
SENSORGASCONC4 2B - "0214" ***
PM1 2B - "000D" (decimal 17 µg/m3)
PM2.5 2B - "0011" ( decimal 22 µg/m3)
PM10 2B - "0013" (decimal 24 µg/m3)
CRC 4B - "62CD3938"
```

* temperature, humidity and noise are small rational numbers (float), encoded as integers. The decoding is as follows:

```c
// an uint16_t stores 16 bits, we use first bit for sign, and following 15 bits for number (0..32767)
// result is divided by 100 for a real with 2 decimals, max is 327.00
double id(uint16_t val) {
  double res = 0;
  uint16_t mask = 1<<15;
  // check negative number
  if (val & mask) {
    val &= ~mask; // remove sign bit
    res = val;
    res = -res;
  } else {
    res = val;
  }
  res /= 100.0; // restore the 2 decimals
  return res;
}
```

Temperature, humidity and noise from the above example translate to:
Temperature = id(0x0710) = id(1808) = 18.08 °C
Humidity = id(0x1C0D) = id(7181) = 71.81 RH
Noise = id(0x1759) = id(5977) = 59.77 dB

** pressure is represented in Pascals, and the normal atmospheric values are a number close to 100000 that exceeds 2bytes to store. The normal fluctuation interval however, is a smaller value. Therefore an offset is used to decrease the number needed to store pressure in Pascals, while still being able to keep track of the ground level fluctuations correctly. An offset of 65535 is applied.

The above example translates too:
Pressure = 0x8BCF = 35791+ 65535 = 101326 Pa = 1013hPa

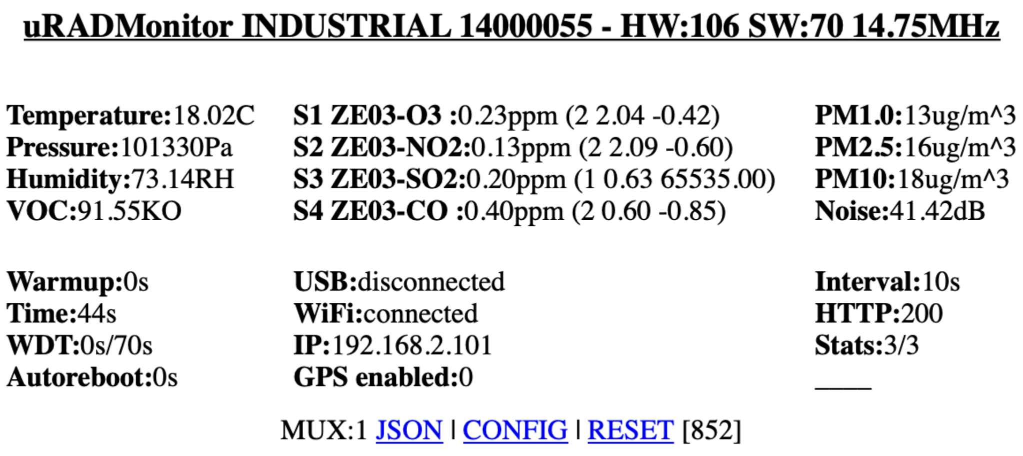
*** Binary mask for the 4 electrochemical sensors that defines the sensor access - or how the sensor data is being retrieved, were each sensor is allocated 2 bits in order, from left to right

The above example translates to: 01010001, and to each number we add +1:
S1/Ozone=01+1=2 S2/Nitrogen Dioxide=01+1=2 S3/Sulphur Dioxide=00+1=1 S4/Carbon Monoxide=01+1=2
The meaning for these values is as follows:
1 = valid sensor ID (we know gas type) + digital sensor access (the sensor output was digital) = best accuracy
2 = valid sensor ID (we know gas type) but digital returns 0 due to low gas concentration, therefore we take the analogue sensor output via an ADC analogue circuit
3 = unknown sensor ID, but digital output, the readings are interpreted on the server, according to config database
4 = unknown sensor ID and no digital output, we read the gas concentration via analogue circuit and send it to the server for processing
Note: Starting with HW106 , only values 1 and 2 will occur with the exception of damaged / empty sensor slots when no ID will be available.
The gas concentration is calculated as follows: Concentration (PPM) = PayloadValue * Multiplier
Multiplier is a function of the sensor access parameter presented above, and has the following values:

```
double mul(int8_t index) {
        switch (sensorZE03Access[index]) {
                case 1: return 10.0;
                case 2: return 1000.0;
                case 3: return 1.0;
                case 4: return 1000.0;
                default: return 1.0; //no change
        }
}
```

For the payload examples, the calculation goes as following:
Gas1 (O3) Concentration = 0x00D4 / mul(2) = 212 / 1000 = 0.212 ppm
Gas2 (NO2) Concentration = 0x0066 / mul(2) = 102 / 1000 = 0.102 ppm
Gas3 (SO2) Concentration = 0x0002 / mul(1) = 2 / 10 = 0.2 ppm
Gas4 (CO) Concentration = 0x0214 / mul(2) = 532 / 1000 = 0.532 ppm

For a scale comparison on the numbers, you can load the device's internal webpage, see the image below - please note the timestamp of the screenshot is not aligned to the payload above:



uRADMonitor INDUSTRIAL 14000055 - HW:106 SW:70 14.75MHz

```
Temperature:18.02C     S1 ZE03-O3 :0.23ppm (2 2.04 -0.42)        PM1.0:13ug/m^3
Pressure:101330Pa      S2 ZE03-NO2:0.13ppm (2 2.09 -0.60)        PM2.5:16ug/m^3
Humidity:73.14RH       S3 ZE03-SO2:0.20ppm (1 0.63 65535.00)     PM10:18ug/m^3
VOC:91.55KO            S4 ZE03-CO :0.40ppm (2 0.60 -0.85)        Noise:41.42dB

Warmup:0s              USB:disconnected                          Interval:10s
Time:44s               WiFi:connected                            HTTP:200
WDT:0s/70s             IP:192.168.2.101                          Stats:3/3
Autoreboot:0s          GPS enabled:0                             ____

                     MUX:1 JSON | CONFIG | RESET [852]
```

Mini webserver exposes the measured parameters

Code wise, the parameters are obtained as follows:

```
uint8_t buf[44] = { 0x14, 0x00, 0x00, 0x55, 0x00, 0x6A, 0x00, 0x46, 0x00, 0x00, 0x00, 0x32, 0x07, 0x10,
0x8B, 0xCF, 0x1C, 0x0D, 0x00, 0x01, 0x6A, 0x42, 0x17,  0x59, 0x51, 0x2A, 0x00, 0xD4, 0x2C, 0x00, 0x66, 0x2B,
0x00, 0x02, 0x04, 0x02, 0x14, 0x00, 0x0D, 0x00, 0x11, 0x00, 0x13, 0x62, 0xCD, 0x39, 0x38 };

uint32_t deviceID = buf[0] << 24 | buf[1] << 16 | buf[2] << 8 | buf[3];
uint16_t verHW = buf[4] << 8 | buf[5];
uint16_t verSW = buf[6] << 8 | buf[7];
uint32_t localtime = buf[8] << 24 | buf[9] << 16 | buf[10] << 8 | buf[11];
uint16_t temperature = id(buf[12] << 8 | buf[13]);
uint16_t pressure = (buf[14] << 8 | buf[15]) + 65535;
uint16_t humidity = id(buf[16] << 8 | buf[17]);
uint32_t voc = buf[18] << 24 | buf[19] << 16 | buf[20] << 8 | buf[21];
uint16_t noise = id(buf[22] << 8 | buf[23]);
uint8_t access1 = 1 + (((buf[24] >> 6) & 0x3);
uint8_t access2 = 1 + (((buf[24] >> 4) & 0x3);
uint8_t access3 = 1 + (((buf[24] >> 2) & 0x3);
uint8_t access4 = 1 + (((buf[24] >> 0) & 0x3);
uint8_t gasid1 = buf[25];
uint16_t conc1 = buf[26] << 8 | buf[27];
uint8_t gasid2 = buf[28];
uint16_t conc2 = buf[29] << 8 | buf[30];
uint8_t gasid3 = buf[31];
uint16_t conc3 = buf[32] << 8 | buf[33];
uint8_t gasid4 = buf[34];
uint16_t conc4 = buf[35] << 8 | buf[36];
uint16_t pm1 = buf[37] << 8 | buf[38];
uint16_t pm25 = buf[39] << 8 | buf[40];
uint16_t pm10 = buf[41] << 8 | buf[42];
uint32_t crc = (buf[43] << 24 | buf[44] << 16 | buf[45] << 8 | buf[46]);
```

The last 4 bytes in the payload contain a CRC32 checksum. The CRC is computed as follows and applied on the entire sequence of bytes, except for the CRC itself (47 - 4 bytes):

```
uint32_t crc32(uint8_t *buffer, uint16_t length) {
        uint32_t crc = 0xFFFFFFFFUL; // initial seed
        uint16_t i, j;
        for (i=0; i<length; i++) {
         crc = crc ^ *(buffer++);
                for (j=0; j<8; j++) {
                        if (crc & 1)
                         crc = (crc>>1) ^ 0xEDB88320UL ;
                        else
                         crc = crc >>1 ;
                }
        }
        return crc ^ 0xFFFFFFFFUL;
}
```

Here is the CRC test result:

```
int main() {
        uint32_t crc = (buf[43] << 24 | buf[44] << 16 | buf[45] << 8 | buf[46]);

        uint32_t computed_crc = crc32(buf, 43);

        (crc == computed_crc)?printf("Match"):printf("Error");

        return 0;
}
```

This checksum is used to verify the integrity of the data.