UC11-N1 Payload Structure

V1.2

Contents

1.Uplink Payload Structure

An uplink message can be sent from end node to gateway. Additionally, the UC11-N1 sends different sensor data in different frames. Therefore, all sensor data must be prefixed with two bytes:
Data Channel: Uniquely identifies each sensor in the UC11-N1 across frames, e.g. “TEMP Sensor”
Data Type: Identifies the data type in the frame, e.g. “Power”.

Note: The device sends multiple sensor data at a time by using the following payload structure:

1 Byte	1 Byte	N Bytes	1 Byte	1 Byte	M Bytes	1 Byte	...
Channel 1	Type 1	Data 1	Channel 2	Type 2	Data 2	Channel 3	...

Channel	Description
1	Battery
3	GPIO1
4	GPIO2
5	Analog Input1
6	Analog Input2

Uplink Packet Example

Frame N: Regular AI1 value uplink.

05 02 c302 c302 c302 c302					
Channel	Type	Ccy Value	Min Value	Max Value	Avg Value
05 means Analog Input1	02 means Analog Input	C3 02 => 02 c3 = 707 means 7.07	C3 02 => 02 c3 = 707 means 7.07	C3 02 => 02 c3 = 707 means 7.07	C3 02 => 02 c3 = 707 means 7.07

Frame N+1: Regular GPIO1 value uplink.

03 00 01		
Channel	Type	Value
03 means GPIO1	00 means Digital Input	01 means high

Frame N+2: Regular GPIO2 pulse value uplink.

04 c8 78 05		
Channel	Type	Value
04 means GPIO2	c8 means Pulse Counter	78 05 => 05 78 => 1400

Frame N+3: Regular RS485(Modbus Master) channel value uplink.

ff 0e 08 25 00000000				
Channel	Type	Channel ID	Data Type	Value of this channel.
ff	0e means the Data of RS485 slave devices	08 means RS485 (Modbus Master) Channel 2	25 => 00100101 101=>5 means Holding Register (Float) 00100=>4 means Data length = 4	00000000

Frame N+4: attribute package: SN , hardware version, software version, AI signal type , battery uplink.

Note: attribute package will be sent out when the device is powered on or rejoins the network.

ff 0b ff ff 01 01					
Channel	Type	Value	Channel	Type	Value
ff=255	0b = 11 (Device Restart Notification)	0xff reserved.	ff=255	01 = 1 (Custom Format Version)	01 = 1 (Version 1)

ff 08 61 22 91 36 34 79		
Channel	Type	Value
ff=255	08 = 8 (Device SN)	61 22 91 36 34 79

ff 09 01 20 ff 0a 01 10					
Channel	Type	Value	Channel	Type	Value
ff = 255	09 (Hardware version)	0120 (V1.2)	ff = 255	0a (Software version)	0110 (V1.10)

ff 14 11 ff 14 20							
Channel	Type	Analog Input	Analog Input Signal Type	Channel	Type	Analog Input	Analog Input Signal Type
ff = 255	14 = 20 (AI signal Type)	1 (AI 1)	1 (0-10 v)	ff = 255	14 = 20 (AI signal Type)	2 (AI 2)	0 (4-20 mA)

01 75 5a		
Channel	Type	Value
01	75 (Battery Capacity)	5a = 90 means 90%

Note: Battery packet will be sent every 6 hours (UC11-N1-DC) or 12 hours (UC11-N1).

2.Downlink Payload Structure

A downlink message can be sent from gateway to end node in order to perform some actions on that device.

When the channel range is 1~253, the format is:

1 Byte	2 Bytes	1 Byte1	1 Byte	2 Bytes	1 Byte
Channel 1	Data 1	0xff (reserved)	Channel2	Data2	0xff (reserved)

When the channel range is above 255, the format is:

1 Byte	1 Byte	N Bytes	1 Byte	1 Byte	M Bytess
255	Type 1	Data 1	255	Type 2	Data 2

Downlink Packet Example

Frame N: Set the data reporting interval as 20mins (1200s), and only enable lora channels with index 0,1,2.

ff 03 00 4b ff 05 01 07					
Channel	Type	Value	Channel	Type	Value
ff = 255	03 (set data collecting interval)	b0 04 =>04 b0 = 1200 (second)	ff = 255	05 (set Channel Mask)	01(set channel as with index within 0-15)

					07= 00000111 (enable channels with index 0,1,2)

3. Data Types

3.1 IPSO Standard Definition

Data Types conform to the IPSO Alliance Smart Objects Guidelines, which identifies each data type with an "Object ID." However, as shown below, a conversion is made to fit the Object ID into a single byte.

DATA_TYPE = IPSO_OBJECT_ID - 3200

Type	IPSO	Hex	Data Size	Data Resolution per Byte
Digital Input	3200	0	1	1
Digital Output	3201	1	1	1
Analog Input	3202	2	8(ccy+min+max+avg)	0.01 signed
Analog Output	3203	3	2	0.01 signed
Illuminance Sensor	3301	65	2	1 Lux Unsigned MSB
Presence Sensor	3302	66	1	1
Temperature Sensor	3303	67	2	0.1 °C Signed MSB
Humidity Sensor	3304	68	1	0.5% Unsigned
Voltage	3316	74	2	0.01V unsigned
Current	3317	75	1%	1%
Depth	3319	77	2	1
Concentration	3325	7d	2	1
Pulse Counter	3400	c8	4	1

3.2 Ursalink Custom Format

Type	Type ID	Data Size	Data Resolution per Byte
Ursalink Custom Format Version	1	1	0x01
Data Collection Interval	2	2	1s
Data Reporting Interval	3	2	1s
LoRa Channel Mask	5	3	ID (1Byte) + Value (2Byte) ID: 1~6
Debug Level	7	1	Bit 0: info Bit 1: debug Bit 2: warn Bit 3: err
Product SN	8	6	641090824375 => 0x641090824375
Hardware Version	9	2	0110 => 0x01 0x10
Software Version	10	2	0110 => 0x01 0x10
Device Power On Notification	11	1	0xff reserved. Contents reported after rebooting each time: Ursalink Custom Format Version+SN+Hardware Version +Software Version+the battery level
The Data of RS485 Slave Devices	14	mutable	Channel ID of RS485(1 bit) + Data Type (8 bits) + Value (N Bytes) Data Type (0~2 bits): 0: Coil 1: Discrete 2: Input Register (INT16) Input Register (INT32 with upper 16 bits) Input Register (INT32 with lower 16 bits) 3: Holding Register (INT16) Holding Register (INT32 with upper 16 bits) Holding Register (INT32 with lower 16 bits) 4: Holding Register (INT32) 5: Holding Register (Float) 6: Input Register (INT32)

			7: Input Register (Float) (3~7 Bits): Data Length
Class Type	15	1	0: Class A 2: Class C
AI Signal Type	20	2	Bit0~Bit 3(mode) 0: 4-20 mA 1: 0-10 V Bit4~Bit 7(channel) 1~2

Channel ID of RS485	Description
7	RS485(Modbus Master) Channel 1
8	RS485(Modbus Master) Channel 2
9	RS485(Modbus Master) Channel 3
…	…
14	RS485(Modbus Master) Channel 8

3.3 LoRaWAN Parameter

Device EUI	24E1+SN
APP EUI	24E1+24C0002A0001
App Port	0x85
NetID	0x010203
DevAddr	The last 8 digits of SN.
AppKey	5572404c696e6b4c6f52613230313823
NwkSKey	5572404c696e6b4c6f52613230313823
AppSKey	5572404c696e6b4c6f52613230313823

4.Decoder Example

```
// N1: Payload Decoder
function Decoder(bytes, port) {
    var decoded = {};

    for (i = 0; i < bytes.length;) {
        // BATTERY
        if (bytes[i] == 0x01) {
            decoded.battery = bytes[i + 2];
            i += 3;
            continue;
        }

        // GPIO
        if (bytes[i] == 0x03) {
            decoded.gpio1 = bytes[i + 2] === 0 ? "off" : "on";
            i += 3;
            continue;
        }

        if (bytes[i] == 0x04 && bytes[i + 1] == 0x00) {
            decoded.gpio2 = bytes[i + 2] === 0 ? "off" : "on";
            i += 3;
            continue;
        }
        if (bytes[i] == 0x04 && bytes [i + 1] == 0xc8){
         //Pulse Counter
          decoded.counter = readUInt16LE(bytes.slice(i + 2, i + 4)) ;
          i += 4;
          continue;
        }

        // ADC
        if (bytes[i] == 0x05) {
            decoded.adc1 = {};
            decoded.adc1.cur = readInt16LE(bytes.slice(i + 2, i + 4)) / 100;
            decoded.adc1.min = readInt16LE(bytes.slice(i + 4, i + 6)) / 100;
            decoded.adc1.max = readInt16LE(bytes.slice(i + 6, i + 8)) / 100;
            decoded.adc1.avg = readInt16LE(bytes.slice(i + 8, i + 10)) / 100;
            i += 10;
```

```
        continue;
    }

    if (bytes[i] == 0x06) {
        decoded.adc2 = {};
        decoded.adc2.cur = readInt16LE(bytes.slice(i + 2, i + 4)) / 100;
        decoded.adc2.min = readInt16LE(bytes.slice(i + 4, i + 6)) / 100;
        decoded.adc2.max = readInt16LE(bytes.slice(i + 6, i + 8)) / 100;
        decoded.adc2.avg = readInt16LE(bytes.slice(i + 8, i + 10)) / 100;
        i += 10;
        continue;
    }

    // MODBUS
    if (bytes[i] == 0xFF && bytes[i + 1] == 0x0E) {
        var chnId = bytes[i + 2];
        var packageType = bytes[i + 3];
        var dataType = packageType & 7;
        var dataLength = packageType >> 3;
        var chn = 'chn' + chnId;
        switch (dataType) {
            case 0:
                decoded[chn] = bytes[i + 4] ? "on" : "off";
                i += 5;
                break;
            case 1:
                decoded[chn] = bytes[i + 4];
                i += 5;
                break;
            case 2:
            case 3:
                decoded[chn] = readUInt16LE(bytes.slice(i + 4, i + 6));
                i += 6;
                break;
            case 4:
            case 6:
                decoded[chn] = readUInt32LE(bytes.slice(i + 4, i + 8));
                i += 8;
                break;
            case 5:
            case 7:
                decoded[chn] = readFloatLE(bytes.slice(i + 4, i + 8));
```

```
                    i += 8;
                    break;
                }
            }
        }

    return decoded;
}

/* ******************************************
 * bytes to number
 ********************************************/
function readUInt8LE(bytes) {
    return (bytes & 0xFF);
}

function readInt8LE(bytes) {
    var ref = readUInt8LE(bytes);
    return (ref > 0x7F) ? ref - 0x100 : ref;
}

function readUInt16LE(bytes) {
    var value = (bytes[1] << 8) + bytes[0];
    return (value & 0xFFFF);
}

function readInt16LE(bytes) {
    var ref = readUInt16LE(bytes);
    return (ref > 0x7FFF) ? ref - 0x10000 : ref;
}

function readUInt32LE(bytes) {
    var value = (bytes[3] << 24) + (bytes[2] << 16) + (bytes[1] << 8) + bytes[0];
    return (value & 0xFFFFFFFF);
}

function readInt32LE(bytes) {
    var ref = readUInt32LE(bytes);
    return (ref > 0x7FFFFFFF) ? ref - 0x100000000 : ref;
}

function readFloatLE(bytes) {
```

```
    // JavaScript bitwise operators yield a 32 bits integer, not a float.
    // Assume LSB (least significant byte first).
    var bits = bytes[3] << 24 | bytes[2] << 16 | bytes[1] << 8 | bytes[0];
    var sign = (bits >>> 31 === 0) ? 1.0 : -1.0;
    var e = bits >>> 23 & 0xff;
    var m = (e === 0) ? (bits & 0x7fffff) << 1 : (bits & 0x7fffff) | 0x800000;
    var f = sign * m * Math.pow(2, e - 150);
    return f;
}
```

---End---