IoT Lab 4

Low Power Adaptive LoRa protocols

In the previous labs involving **direct** and **close terminals** using **WiFi** and **WiFi MAC–WiFi** connections, we integrated **confirmed transmissions** and **security mechanisms**.
To provide the same **low-power control and adaptive behavior**, it is necessary to implement **ACK packets** together with an **AES-based encryption scheme**.
The **ESP32-C3** SoCs integrates **hardware AES accelerators**, which allow these security mechanisms to be implemented efficiently with minimal energy overhead.
ACK packets may serve two different purposes:

- **Short ACK packets**, which simply confirm successful reception of a data packet,
- **Control ACK packets**, which additionally carry **meta-parameters** used to adapt and optimize the operation of the **low-power protocol**.
-

This approach ensures both **secure communication** and **energy-efficient adaptive control** across different types of IoT terminals.

Let us remind the **parameters** and the **meta-parameters** used in Low Power Protocols.

- **cycle**: with **`base_cycle`** (in seconds) and **`max_cycle`** factor
- **delta**: with **`min_delta`** and **`max_delta`**
- **threshold**: **`t_low`** and **`t_high`**

The **cycle parameter** defines the duration of the **low-power stage**. The initial **`base_cycle`** value is fixed either in the program code or stored in **non-volatile memory (NVS)** (for example, 10 seconds). The effective cycle duration is obtained by multiplying the **`base_cycle`** by a **cycle factor**.

The **cycle factor** starts at **1** and evolves dynamically based on changes in the sensor values. When there are **two consecutive high-power stages without transmission**, the cycle factor is **multiplied by 2** and stored in **low-power RTC SRAM**. This updated value is then used during the next high-power stage.

Because the **main SRAM is powered off during the low-power stage**, the cycle factor must be saved in RTC memory before entering deep sleep.

Conversely, when there are **two consecutive high-power stages with transmission**, the cycle factor is **divided by 2**, reducing the duration of the next low-power stage.

The **delta parameter**, when applied to a temperature sensor, represents the difference between the **last transmitted value** and the **current measured value**. The delta value is initially set to **`max_delta`** (for example, 0.1). During extended periods without transmission, the delta value may gradually decrease toward **`min_delta`**, increasing measurement sensitivity. In contrast, during periods with frequent transmissions, the delta value may increase toward **`max_delta`** to reduce unnecessary updates.

The **threshold parameter** defines the upper (**`t_high`**) and lower (**`t_low`**) limits of acceptable sensor values.

When the current sensor reading crosses either threshold, the system must:

- Transmit the data packet during the high-power stage,
- Reduce the cycle factor to ensure faster subsequent updates.

The **adaptive adjustment** of both the cycle factor and the delta parameter enables:

- An exponential extension of the **low-power stage duration** (i.e., the cycle time) when conditions are stable,
- Fine-tuning of the delta value to achieve **maximum precision** in sensor measurements.

For correct operation, the **current cycle factor and delta values** must be stored in **low-power RTC SRAM**, which retains its contents during the SoC's **deep-sleep mode**.
However, RTC SRAM loses its contents when the device is completely powered off.
To preserve **meta-parameters** such as **`base_cycle, max_cycle, min_delta, max_delta, t_low, and t_high`** across power cycles, these values must be stored in **non-volatile internal memory (NVS)**.

4.1 Remote Terminal – sender

The following is the complete code of the sender node:

```
_____________________________________________________________________________________________
import time, ustruct
from machine import I2C, Pin, freq, deepsleep
from sensors import sensors
from lora_init import lora_init
from rtc_tools import *
from nvs_tools import *
from aes_tools import *
# Initialize LoRa communication
lora = lora_init()
aes_key="smartcomputerlab"     # constant - 1 ECB mode
cdef=1; ts_chan=1234; ncycle=1
ACK_wait_time = 2                    # ACK waiting time depends on the protocol and data rate
nvs_key="param"
led = Pin(3, Pin.OUT)

def onReceive(lora_modem,payload):
    global cdef; global ncycle
    if len(payload)==16:                  # the payload: max_cycle, t_high, t_low
        ack=aes_decrypt(payload,aes_key)
        chan,cntr,c_par,d_par=ustruct.unpack('3if',ack)   # ack parameters - to confirm
        if chan==ts_chan:
            print("short ACK received")             # no channel test
        lora.sleep()                      # only for deepsleep
        time.sleep(0.1)
        deepsleep(ncycle*cdef*1000)

    if len(payload)==32:                  # the payload: max_cycle, t_high, t_low
        ack=aes_decrypt(payload,aes_key)
        chan,cntr,c_def,c_max,d_min,d_max,t_low,t_high=ustruct.unpack("4i4f",ack) # meta parameters
        if chan==ts_chan:
            print("long ACK parameters received")             # no channel test
            value=ustruct.pack("2i4f",c_def,c_max,d_min,d_max,t_low,t_high)
            write_nvs_power(nvs_key, value)
            print("new parameters written to nvs")
        lora.sleep()                      # only for deepsleep
        time.sleep(0.1)
        ncycle,npos,nneg=rtc_load_param()
        print(ncycle*cdef)
        deepsleep(ncycle*cdef*1000)

def send_lora_data(ts_chan,ts_wkey,l,t,h):
    try:
        message = f"L:{l:.2f},T:{t:.2f},H:{h:.2f}"
        print("Sending LoRa packet:", message)
        data = ustruct.pack('i16s3f',ts_chan,ts_wkey,l,t,h)  # 32 bytes - short version
        enc_data=aes_encrypt(data,aes_key)
        lora.println(enc_data)
        print("LoRa encrypted packet sent successfully.")
    except Exception as e:
        print("Failed to send LoRa packet:", e)
# Main program

def main():
    global cdef; global ts_chan; global ncycle
    freq(20000000)
    print("Reading ts from internal EEPROM...")
    len,ts_rparam = read_nvs_ts(nvs_key)
    if len:
        ts_chan,ts_wkey=ustruct.unpack("i16s",ts_rparam)
        print("len:",len,"ts_chan:",ts_chan,"ts_wkey:",ts_wkey.decode())
    print("Reading pow from internal EEPROM...")
    len,pow_rparam = read_nvs_power(nvs_key)
    if len:
```

```
        cdef,cmax,dmin,dmax,tlow,thigh=ustruct.unpack("2i4f",pow_rparam)
        print("len:",len,", cdef:",cdef,", cmax:",cmax,", dmin:",dmin,", dmax:",dmax,",
tlow:",tlow,", thigh:",thigh)

    lora.onReceive(onReceive)
    lora.receive()
    while True:
        ncycle,npos,nneg= rtc_load_param()
        ssens= rtc_load_sensor(); sdelta= rtc_load_delta()
        print("ncycle:" +str(ncycle));
        lumi, temp, humi = sensors(sda=8, scl=9)
        print("Luminosity:", lumi, "lux")
        print("Temperature:", temp, "C")
        print("Humidity:", humi, "%")
        print("current: "+str(temp)+" saved: "+str(ssens));  # sensor is temperature
        print(dmin,dmax,sdelta);
        if abs(ssens-temp)>sdelta or temp>thigh or temp<tlow :  # testing temp difference and
thresholds
            print("data SENT")
            rtc_store_sensor(temp)
            led.on()
            if npos :
                if ncycle > 2:
                    ncycle= int(ncycle/2)
                else:
                    if sdelta< dmax:
                        sdelta = sdelta*2         # new delta
                        rtc_store_delta(sdelta)

            npos=npos+1; nneg=0  # positive and negative counters
            rtc_store_param(ncycle,npos,nneg)
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet sent, no ack received")
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet re-sent, no ack received")
            led.off()
        else:
            print("data packet NOT sent")
            if nneg :
                if ncycle < cmax:
                    ncycle = int(ncycle*2)           # maximum factor 64 (64*15sec)
                else :
                    if sdelta> dmin:
                        sdelta = sdelta/2
                        rtc_store_delta(sdelta)

            npos=0; nneg=nneg+1
            rtc_store_param(ncycle,npos,nneg)
            # waiting for ACK frame
        lora.sleep()                        # only for deepsleep
        time.sleep(0.1)
        print(ncycle*cdef)
        print(sdelta)
        deepsleep(ncycle*cdef*1000)               # 10*1000 miliseconds

# Run the main program
main()
```

To do:

To improve the above program we have to decompose the following condition into two parts:

```
if abs(ssens-temp)>sdelta or temp>thigh or temp<tlow :  # testing temp difference and thresholds
…
else
-

if abs(ssens-temp)>sdelta :              # testing temp difference
..
elif temp>thigh or temp<tlow :           # testing thresholds
..
else:
..
```

4.1.1 The completed code for terminal

```
-----------------------------------------------------------------------------------------------------
import time, ustruct
from machine import I2C, Pin, freq, deepsleep
from sensors import sensors
from lora_init import lora_init
from rtc_tools import *
from nvs_tools import *
from aes_tools import *
# Initialize LoRa communication
lora = lora_init()
aes_key="smartcomputerlab"     # constant - 1 ECB mode
cdef=1; ts_chan=1234; ncycle=1
ACK_wait_time = 2                    # ACK waiting time depends on the protocol and data rate
nvs_key="param"
led = Pin(3, Pin.OUT)

def onReceive(lora_modem,payload):
    global cdef; global ncycle
    if len(payload)==16:                 # the payload: max_cycle, t_high, t_low
        ack=aes_decrypt(payload,aes_key)
        chan,cntr,c_par,d_par=ustruct.unpack('3if',ack)   # ack parameters - to confirm
        if chan==ts_chan:
            print("short ACK received")             # no channel test
        lora.sleep()                     # only for deepsleep
        time.sleep(0.1)
        deepsleep(ncycle*cdef*1000)

    if len(payload)==32:                 # the payload: max_cycle, t_high, t_low
        ack=aes_decrypt(payload,aes_key)
        chan,cntr,c_def,c_max,d_min,d_max,t_low,t_high=ustruct.unpack("4i4f",ack)   # ack parameters
- to confirm and save
        if chan==ts_chan:
            print("long ACK parameters received")             # no channel test
            value=ustruct.pack("2i4f",c_def,c_max,d_min,d_max,t_low,t_high)
            write_nvs_power(nvs_key, value)
            print("new parameters written to nvs")
        lora.sleep()                     # only for deepsleep
        time.sleep(0.1)
        ncycle,npos,nneg=rtc_load_param()
        print(ncycle*cdef)
        deepsleep(ncycle*cdef*1000)

def send_lora_data(ts_chan,ts_wkey,l,t,h):
    try:
        message = f"L:{l:.2f},T:{t:.2f},H:{h:.2f}"
        print("Sending LoRa packet:", message)
        data = ustruct.pack('i16s3f',ts_chan,ts_wkey,l,t,h)  # 32 bytes - short version
        enc_data=aes_encrypt(data,aes_key)
        lora.println(enc_data)
        print("LoRa encrypted packet sent successfully.")
    except Exception as e:
        print("Failed to send LoRa packet:", e)
# Main program

def main():
    global cdef; global ts_chan; global ncycle
    freq(20000000)
    print("Reading ts from internal EEPROM...")
    len,ts_rparam = read_nvs_ts(nvs_key)
    if len:
        ts_chan,ts_wkey=ustruct.unpack("i16s",ts_rparam)
        print("len:",len,"ts_chan:",ts_chan,"ts_wkey:",ts_wkey.decode())
    print("Reading pow from internal EEPROM...")
    len,pow_rparam = read_nvs_power(nvs_key)
    if len:
        cdef,cmax,dmin,dmax,tlow,thigh=ustruct.unpack("2i4f",pow_rparam)
        print("len:",len,", cdef:",cdef,", cmax:",cmax,", dmin:",dmin,", dmax:",dmax,",
tlow:",tlow,", thigh:",thigh)

    lora.onReceive(onReceive)
    lora.receive()
    while True:
        ncycle,npos,nneg= rtc_load_param()
        ssens= rtc_load_sensor(); sdelta= rtc_load_delta()
        print("ncycle:" +str(ncycle));
        lumi, temp, humi = sensors(sda=8, scl=9)
        print("Luminosity:", lumi, "lux")
        print("Temperature:", temp, "C")
        print("Humidity:", humi, "%")
        print("current: "+str(temp)+" saved: "+str(ssens));               # sensor is temperature
        print(dmin,dmax,sdelta);
        if abs(ssens-temp)>sdelta :  # testing delta and thresholds
            print("data to SEND")
            rtc_store_sensor(temp)
            led.on()
```

```
            if npos :
                if ncycle > 2:
                    ncycle= int(ncycle/2)
                else:
                    if sdelta< dmax:
                        sdelta = sdelta*2         # new delta
                        rtc_store_delta(sdelta)

            npos=npos+1; nneg=0  # positive and negative counters
            rtc_store_param(ncycle,npos,nneg)
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet sent, no ack received")
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet re-sent, no ack received"); led.off()

        elif temp>thigh or temp<tlow :
            ncycle=1; npos=0; nneg=0; rtc_store_param(ncycle,npos,nneg)
            sdelta = dmin; rtc_store(sdelta)
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet sent, no ack received")
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet re-sent, no ack received"); led.off()

        else:
            print("data packet NOT sent")
            if nneg :
                if ncycle < cmax:
                    ncycle = int(ncycle*2)           # maximum factor 64 (64*15sec)
                else :
                    if sdelta> dmin:
                        sdelta = sdelta/2
                        rtc_store_delta(sdelta)

            npos=0; nneg=nneg+1
            rtc_store_param(ncycle,npos,nneg)
            # waiting for ACK frame
        lora.sleep()                        # only for deepsleep
        time.sleep(0.1)
        print(ncycle*cdef)
        print(sdelta)
        deepsleep(ncycle*cdef*1000)               # 10*1000 miliseconds

# Run the main program
main()
```

4.1.2 The completed code for receiver - gateway

```
from machine import Pin, I2C, SPI
import ustruct, random, ubinascii, urequests
from lora_init import *
from display_sensors import *
from aes_tools import *
import machine,time
from wifi_tools import *
# WiFi credentials
SSID = 'PhoneAP'
PASS = 'smartcomputerlab'
AES_KEY = b'smartcomputerlab'  # Replace with your actual 16-byte AES key
# Initialize LoRa modem
lora_modem = lora_init()
rssi=0; chan=0; wkey=""; lumi=0.0; temp=0.0; humi=0.0; precv=0

def send_data_to_thingspeak(lumi, temp, humi, rssi):
    try:
        sf1="&field1="+str(lumi); sf2="&field2="+str(temp); sf3="&field3="+str(humi);
sf4="&field4="+str(rssi)
        url = "https://thingspeak.com/update?key="+wkey.decode()+sf1+sf2+sf3+sf4
        response = urequests.get(url)
        response.close()
        print("Data sent to ThingSpeak:", lumi, temp, humi, rssi)
    except Exception as e:
        print("Failed to send data:", e)

# --- Receive LoRa Packet ---
ack_num=0
def onReceive(lora_modem,payload):
    global rssi; global chan; global wkey; global lumi; global temp; global humi; global precv;
global ack_num
    rssi = lora_modem.packetRssi()
    if len(payload)==32:
        precv=1
        rssi = lora_modem.packetRssi()
        data=aes_decrypt(payload,AES_KEY)
        chan, wkey, lumi, temp, humi = ustruct.unpack('i16s3f', data)
        print("Received encrypted LoRa packet with RSSI: "+str(rssi))   #, payload.decode())
        print(chan,wkey,lumi,temp,humi)
        display_sensors(8,9,lumi,temp,humi,0)
        ack_num=ack_num+1; print("ack:",ack_num)
        if ack_num%60 :
            rcycle=random.randint(5,15)
            control=0;
            ack=ustruct.pack('3if',chan,control,rcycle,0.1)
            enc_ack=aes_encrypt(ack,AES_KEY)
            lora_modem.println(enc_ack)  # sending ACK packet
            print("send short encrypted ack AES packet")
        else:    # sends long ACK encrypted packet every 60 packets
            cntr=0; c_def=1; c_max=64; d_min=0.01;d_max=0.2;t_low=16.0; t_high=26.0
            ack=ustruct.pack("4i4f",chan,cntr,c_def,c_max,d_min,d_max,t_low,t_high)
            enc_ack=aes_encrypt(ack,AES_KEY)
            lora_modem.println(enc_ack)  # sending ACK packet
            print("send long encrypted ack AES packet")

        lora_modem.receive()

def main():
    global rssi; global lumi; global temp; global humi; global precv  # packet received
    lora_modem.onReceive(onReceive)
    lora_modem.receive()
    while True :
        if precv:
            if connect_WiFi(SSID, PASS):
                print("WiFi connected")
            send_data_to_thingspeak(lumi, temp, humi, rssi)
            time.sleep(1)
            disconnect_WiFi()
            precv=0
            time.sleep(15)

        print("waiting for packet")
        time.sleep(1)


main()
```

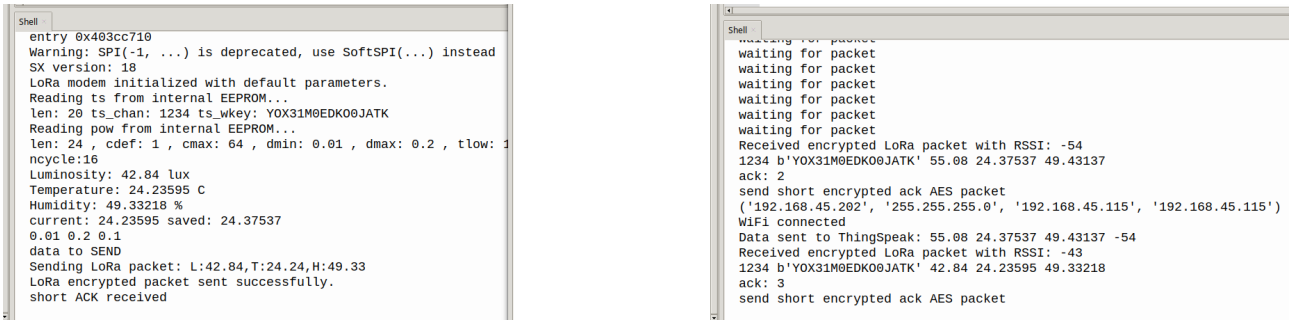

Fig 4.1 IDE terminals output : sender and receiver

Required modules (tools):

In sender (terminal) module:

```
from rtc_tools import *
from nvs_tools import *
from aes_tools import *
```

In receiver (gateway) module

```
from aes_tools import *
from wifi_tools import *
```

```
        chan,cntr,c_par,d_par=ustruct.unpack('3if',ack)    # ack parameters - to confirm

        chan,cntr,c_def,c_max,d_min,d_max,t_low,t_high=ustruct.unpack("4i4f",ack)
```

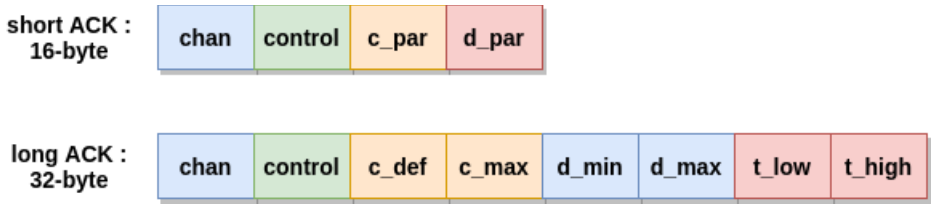

Fig 4.2
Formats of ACK packets send by the gateway to the terminal node.

Fig 4.3a PPK2 diagram: **`high_power` stage** with LoRa transmission phase (**SF=7,SB=125KHz,CR=5**) with received ACK packet (no retransmission and no wait time-out); **cost ~36mC**.

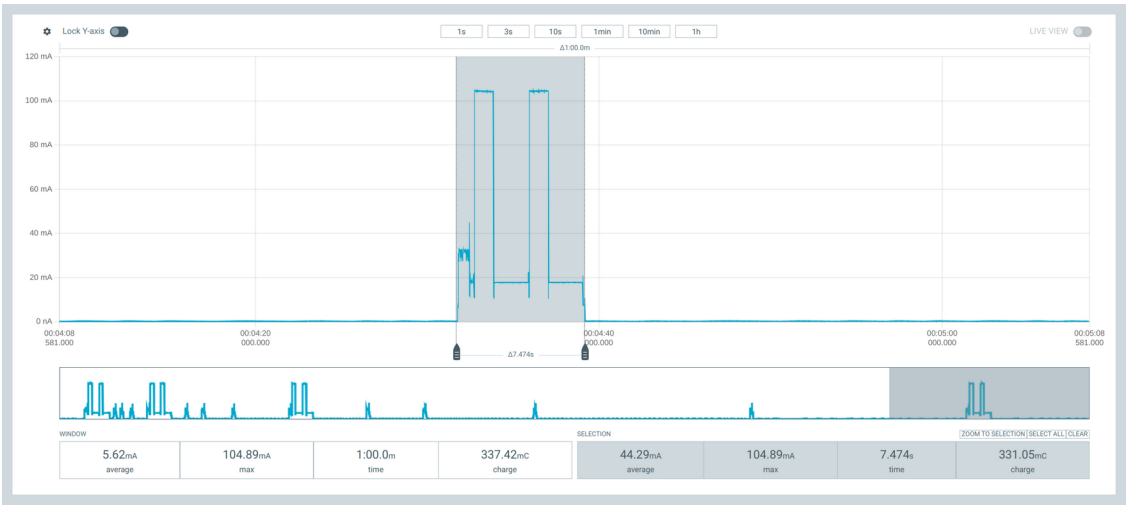

Fig 4.3b PPK2 diagram: **`high_power` stage** with LoRa transmission phase (**SF=11,SB=125KHz,CR=8**) with no received ACK packet (retransmission and wait time-out – 2sec); **cost ~331mC**.

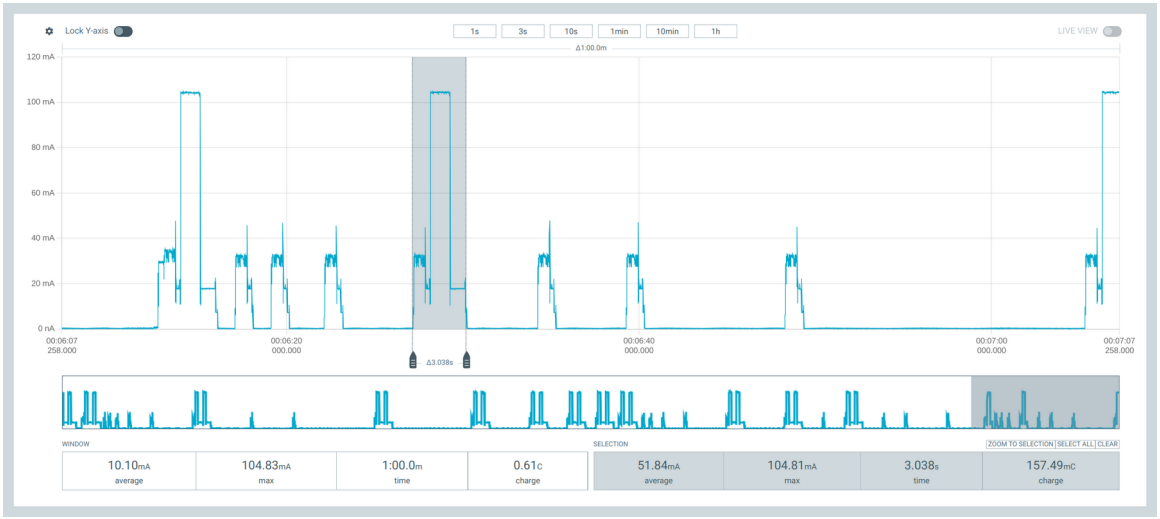

Fig 4.3c PPK2 diagram: **`high_power` stage** with LoRa transmission phase (**SF=11,SB=125KHz,CR=8**) with received ACK packet (no retransmission and no wait time-out – 2sec); **cost ~157.49mC**.

4.1.3 Adding external EEPROM (AT24C32) for meta-parameters

To complete the provision of meta-parameters we can use an external EEPROM module ‘such as AT24C32, pre-loaded with the initial meta-parameters.
This module may be initially connected (bus I2C) to the terminal node or may be connected permanently to the gateway.
The following is the completed terminal node code:

```
-----------------------------------------------------------------------------------------------
import time, ustruct
from machine import I2C, Pin, freq, deepsleep
from sensors import sensors
from lora_init import lora_init
from rtc_tools import *
from nvs_tools import *
from aes_tools import *
from at24_to_nvs import *
# Initialize LoRa communication
lora = lora_init()
aes_key="smartcomputerlab"     # constant - 1 ECB mode
cdef=1; ts_chan=1234; ncycle=1
ACK_wait_time = 2                    # ACK waiting time depends on the protocol and data rate
nvs_key="param"
led = Pin(3, Pin.OUT)

def onReceive(lora_modem,payload):
    global cdef; global ncycle
    if len(payload)==16:                 # the payload: max_cycle, t_high, t_low
        ack=aes_decrypt(payload,aes_key)
        chan,cntr,c_par,d_par=ustruct.unpack('3if',ack)   # ack parameters - to confirm
        if chan==ts_chan:
            print("short ACK received")             # no channel test
        lora.sleep()                     # only for deepsleep
        time.sleep(0.1)
        deepsleep(ncycle*cdef*1000)

    if len(payload)==32:                 # the payload: max_cycle, t_high, t_low
        ack=aes_decrypt(payload,aes_key)
        chan,cntr,c_def,c_max,d_min,d_max,t_low,t_high=ustruct.unpack("4i4f",ack)   # ack parameters
- to confirm and save
        if chan==ts_chan:
            print("long ACK parameters received")             # no channel test
            value=ustruct.pack("2i4f",c_def,c_max,d_min,d_max,t_low,t_high)
            write_nvs_power(nvs_key, value)
            print("new parameters written to nvs")
        lora.sleep()                     # only for deepsleep
        time.sleep(0.1)
        ncycle,npos,nneg=rtc_load_param()
        print(ncycle*cdef)
        deepsleep(ncycle*cdef*1000)

def send_lora_data(ts_chan,ts_wkey,l,t,h):
    try:
        message = f"L:{l:.2f},T:{t:.2f},H:{h:.2f}"
        print("Sending LoRa packet:", message)
        data = ustruct.pack('i16s3f',ts_chan,ts_wkey,l,t,h)  # 32 bytes - short version
        enc_data=aes_encrypt(data,aes_key)
        lora.println(enc_data)
        print("LoRa encrypted packet sent successfully.")
    except Exception as e:
        print("Failed to send LoRa packet:", e)
# Main program

def main():
    global cdef; global ts_chan; global ncycle
    at24_to_nvs()           #  here we may load new meta-parameters if the AT24 is connected
    freq(20000000)
    print("Reading ts from internal EEPROM...")
    len,ts_rparam = read_nvs_ts(nvs_key)
    if len:
        ts_chan,ts_wkey=ustruct.unpack("i16s",ts_rparam)
        print("len:",len,"ts_chan:",ts_chan,"ts_wkey:",ts_wkey.decode())
    print("Reading pow from internal EEPROM...")
    len,pow_rparam = read_nvs_power(nvs_key)
    if len:
        cdef,cmax,dmin,dmax,tlow,thigh=ustruct.unpack("2i4f",pow_rparam)
        print("len:",len,", cdef:",cdef,", cmax:",cmax,", dmin:",dmin,", dmax:",dmax,",
tlow:",tlow,", thigh:",thigh)

    lora.onReceive(onReceive)
    lora.receive()
    while True:
        ncycle,npos,nneg= rtc_load_param()
        ssens= rtc_load_sensor(); sdelta= rtc_load_delta()
        print("ncycle:" +str(ncycle));
        lumi, temp, humi = sensors(sda=8, scl=9)
        print("Luminosity:", lumi, "lux")
```

```
        print("Temperature:", temp, "C")
        print("Humidity:", humi, "%")
        print("current: "+str(temp)+" saved: "+str(ssens));                # sensor is temperature
        print(dmin,dmax,sdelta);
        if abs(ssens-temp)>sdelta :  # testing delta and thresholds
            print("data to SEND")
            rtc_store_sensor(temp)
            led.on()
            if npos :
                if ncycle > 2:
                    ncycle= int(ncycle/2)
                else:
                    if sdelta< dmax:
                        sdelta = sdelta*2         # new delta
                        rtc_store_delta(sdelta)

            npos=npos+1; nneg=0  # positive and negative counters
            rtc_store_param(ncycle,npos,nneg)
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet sent, no ack received")
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet re-sent, no ack received"); led.off()

        elif temp>thigh or temp<tlow :
            ncycle=1; npos=0; nneg=0; rtc_store_param(ncycle,npos,nneg)
            sdelta = dmin; rtc_store(sdelta)
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet sent, no ack received")
            send_lora_data(ts_chan,ts_wkey,lumi, temp, humi)
            lora.receive()
            time.sleep(ACK_wait_time)
            print("data packet re-sent, no ack received"); led.off()

        else:
            print("data packet NOT sent")
            if nneg :
                if ncycle < cmax:
                    ncycle = int(ncycle*2)           # maximum factor 64 (64*15sec)
                else :
                    if sdelta> dmin:
                        sdelta = sdelta/2
                        rtc_store_delta(sdelta)

            npos=0; nneg=nneg+1
            rtc_store_param(ncycle,npos,nneg)
            # waiting for ACK frame
        lora.sleep()                        # only for deepsleep
        time.sleep(0.1)
        print(ncycle*cdef)
        print(sdelta)
        deepsleep(ncycle*cdef*1000)               # 10*1000 miliseconds

# Run the main program
main()
```

The above program imports **`at24_to_nvs.py`** module:

```
from nvs_tools import *
from at24_tools import *
from check_at24 import *

def at24_to_nvs():
    nvs_key = "param"
    if check_eeprom():
        I2C_SCL = 9; I2C_SDA = 8
        i2c = machine.I2C(0, scl=machine.Pin(I2C_SCL), sda=machine.Pin(I2C_SDA))
        eeprom = AT24C32(i2c)
        ts_addr = 0x00  # Starting address in EEPROM for ThingSpeak meta-parameters
        pow_addr = 0x80   # 255 - starting address for power mata-parameters
        print("Reading from AT24C32...")
        ts_rparam = eeprom.read_at24(ts_addr, 20)    # len ts_rparam
        pow_rparam = eeprom.read_at24(pow_addr, 24)  # len pow_rparam
        print("Writing to NVS...")
        write_nvs_ts(nvs_key, ts_rparam)
        write_nvs_power(nvs_key, pow_rparam)
    else:
        print("no AT24CXX module found..")
    print("Reading from NVS...")
    len,ts_rparam = read_nvs_ts(nvs_key)
```

```
    if len:
        chan,wkey=ustruct.unpack("i16s",ts_rparam)
        print("len:",len,"ts_chan:",chan,"ts_wkey:",wkey.decode())
    len,pow_rparam = read_nvs_power(nvs_key)
    if len:
        cdef,cmax,dmin,dmax,tlow,thigh=ustruct.unpack("2i4f",pow_rparam)
        print("len:",len,", cdef:",cdef,", cmax:",cmax,", dmin:",dmin,", dmax:",dmax,",
tlow:",tlow,", thigh:",thigh)
```

As we see the **`at24_to-nvs.py`** module needs **`at24_tools.py`** and **`check_at24.py`** modules

---

```
# at24_tools.py
import machine
import time
import ustruct

class AT24C32:
    def __init__(self, i2c, address=0x50):
        self.i2c = i2c
        self.address = address
        self._capacity = 4096  # AT24C32 has 4KB capacity

    def write_at24(self, addr, buff):
        if not isinstance(buff, (bytes, bytearray)):
            raise ValueError("Buffer must be of type 'bytes' or 'bytearray'")
        if addr < 0 or addr + len(buff) > self._capacity:
            raise ValueError("Address out of range")
        for i in range(len(buff)):
            self.i2c.writeto(self.address, bytes([addr >> 8, addr & 0xFF, buff[i]]))
            time.sleep(0.01)  # EEPROM write delay
            addr += 1

    def read_at24(self, addr, length):
        if addr < 0 or addr + length > self._capacity:
            raise ValueError("Address out of range")
        self.i2c.writeto(self.address, bytes([addr >> 8, addr & 0xFF]))
        return self.i2c.readfrom(self.address, length)

    def capacity(self):
        return self._capacity
```

```
from machine import Pin, I2C
import time

# Initialize I2C (GPIO21 = SDA, GPIO22 = SCL)
i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=100000)

# Common I2C address range for AT24CXX EEPROMs
EEPROM_ADDRESSES = [0x50 + i for i in range(8)]  # 0x50 - 0x57

def check_eeprom():
    print("Scanning I2C bus for AT24CXX EEPROM...")
    devices = i2c.scan()
    if not devices:
        print("No I2C devices found.")
        return False

    found = False
    for addr in EEPROM_ADDRESSES:
        if addr in devices:
            try:
                # Try reading 1 byte from address 0x00
                i2c.writeto(addr, b'\x00')          # Set memory pointer to 0x00
                data = i2c.readfrom(addr, 1)         # Read 1 byte
                print(f"EEPROM found at address 0x{addr:02X}. Data at 0x00: {data[0]:02X}")
                found = True
            except Exception as e:
                print("Device responded but did not behave like EEPROM.")

    if not found:
        print("No AT24CXX EEPROM found on the I2C bus.")
    return found
```
