IoT 5

Tools

5.1 aes_tools.py

```
-----------------------------------------------------------------------------------------------
# aes_tools.py
import ucryptolib

# AES encryption function using ucryptolib
def aes_encrypt(data,aes_key):
    cipher = ucryptolib.aes(aes_key, 1)  # 1 = ECB mode
    encrypted = cipher.encrypt(data)     # data size  must be multiple 16 bytes
    return encrypted
# AES decryption function
def aes_decrypt(encrypted_data,aes_key):
    cipher = ucryptolib.aes(aes_key, 1)  # 1 = ECB mode
    data = cipher.decrypt(encrypted_data)   # data size  must be multiple 16 bytes
    return data
-----------------------------------------------------------------------------------------------
```

5.2 nvs_tools.py

```
# nvs_tools.py
import machine, ustruct
import esp32

# Function to write data to internal flash memory using NVS (Non-Volatile Storage)
def write_nvs_ts(key, value):
    nvs_key = esp32.NVS("thingspeak")  # Open the NVS namespace "thingspeak"
    nvs_key.set_blob(key, value)  # Store a byte array (blob) with a key
    nvs_key.commit()  # Commit the changes

# Function to read data from internal flash memory using NVS
def read_nvs_ts(key):
    nvs_key = esp32.NVS("thingspeak")  # Open the NVS namespace "thingspeak"
    try:
        buff = bytearray(32)
        value = nvs_key.get_blob(key,buff)  # Retrieve the byte array (blob) using the key
        return value,buff
    except OSError:
        print(f"Key '{key}' not found in EEPROM.")
        return None

# Function to write data to internal flash memory using NVS (Non-Volatile Storage)
def write_nvs_power(key, value):
    nvs_key = esp32.NVS("power")  # Open the NVS namespace "thingspeak"
    nvs_key.set_blob(key, value)  # Store a byte array (blob) with a key
    nvs_key.commit()  # Commit the changes

# Function to read data from internal flash memory using NVS
def read_nvs_power(key):
    nvs_key = esp32.NVS("power")  # Open the NVS namespace "thingspeak"
    try:
        buff = bytearray(32)
        value = nvs_key.get_blob(key,buff)  # Retrieve the byte array (blob) using the key
        return value,buff
    except OSError:
        print(f"Key '{key}' not found in EEPROM.")
        return None
-----------------------------------------------------------------------------------------------
```

5.3 rtc_tools.py

```
# rtc_tools.py
import machine
import ustruct

rtc = machine.RTC()
# Function to store four integer/float values in RTC memory
def rtc_store_param(cycle, pos, neg):  # these values define next cycle length factor
    data = rtc.memory()
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    data = ustruct.pack('3i2f',cycle,pos,neg,s,d)
    rtc.memory(data)

def rtc_load_param():
    # Retrieve the packed data from RTC memory
    c=1; p=0; n=0; s=20.0; d=0.1
    data = rtc.memory()
    if not data:
```

```
        data = ustruct.pack('3i2f',c,p,n,s,d)
        rtc.memory(data)
    # Unpack the integers from the byte array
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    return c, p, n

def rtc_store_delta(delta):       # stores last sent sensor and delta values
    data = rtc.memory()
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    data = ustruct.pack('3i2f',c,p,n,s,delta)
    rtc.memory(data)

def rtc_load_delta():                  # loads stored sensor and delta values, or default init
    c=0; p=0; n=0; s=20.0; d=0.1
    data = rtc.memory()
    if not data:
        data = ustruct.pack('3i2f',c,p,n,s,d)
        rtc.memory(data)
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    return d

def rtc_store_sensor(sensor):     # stores last sent sensor and delta values
    data = rtc.memory()
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    data = ustruct.pack('3i2f',c,p,n,sensor,d)
    rtc.memory(data)

def rtc_load_sensor():                  # loads stored sensor and delta values, or default init
    c=0; p=0; n=0; s=20.0; d=0.1
    data = rtc.memory()
    if not data:
        data = ustruct.pack('3i2f',c,p,n,s,d)
        rtc.memory(data)
    c,p,n,s,d = ustruct.unpack('3i2f', data);
    return s
```

5.4 wifi_tools.py

```
# wifi_tools.py
import network
import time

def connect_WiFi(ssid, passwd, timeout=10):
    """
    Connect to the given WiFi network using the specified SSID and password.
    :param ssid: The SSID of the WiFi network.
    :param passwd: The password of the WiFi network.
    :param timeout: Maximum time in seconds to wait for connection.
    :return: True if connected, False otherwise.
    """
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    wlan.config(txpower=5.0)  # or even 5.0
    # If already connected to the same network, return True
    if wlan.isconnected() and wlan.config('essid') == ssid:
        print(wlan.ifconfig())
        return True
    # Connect to the given SSID
    wlan.connect(ssid, passwd)
    # Wait for connection or timeout
    start = time.time()
    while not wlan.isconnected():
        if time.time() - start > timeout:
            return False
        time.sleep(1)
    print(wlan.ifconfig())
    return True

def disconnect_WiFi():
    """
    Disconnect from the currently connected WiFi network.
    """
    wlan = network.WLAN(network.STA_IF)
    if wlan.isconnected():
        wlan.disconnect()

def scan_WiFi():
    """
    Scan for available WiFi networks.
    :return: A list of tuples containing network information:
             (ssid, bssid, channel, RSSI, authmode, hidden)
    """
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    return wlan.scan()
```

5.5 at24_tools.py

```
# 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
```

```
# at24_check.py
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(f"Device at 0x{addr:02X} responded but did not behave like EEPROM. Error:
{e}")

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

5.6 at24_to_nvs.py

```
# at24_to_nvs.py
from nvs_tools import *
from at24_tools import *
from at24_check 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)
```

5.7 lora_init.py

```
#lora_init.py

from machine import Pin, SPI
import time
import sx127x  # SX127x LoRa driver (ensure the library is installed)
import esp32

# --- LoRa Pins and SPI Bus Setup --- HT
lora_pins = {
    'dio_0': 2,        # DIO0 pin for interrupt
    'ss': 4,           # Slave Select (SS)
    'reset': 10,       # Reset pin
    'sck': 6,          # SPI Clock pin
    'miso': 5,         # SPI MISO pin
    'mosi': 7          # SPI MOSI pin
}

# SPI bus configuration for SX1276
lora_spi = SPI(
    baudrate=10000000,    # Set baudrate to 10 MHz
    polarity=0,           # Clock polarity (CPOL)
    phase=0,              # Clock phase (CPHA)
    bits=8,               # 8 bits per transfer
    firstbit=SPI.MSB,     # MSB first
    sck=Pin(lora_pins['sck'], Pin.OUT, Pin.PULL_DOWN),   # SCK (clock)
    mosi=Pin(lora_pins['mosi'], Pin.OUT, Pin.PULL_UP),   # MOSI (Master Out Slave In)
    miso=Pin(lora_pins['miso'], Pin.IN, Pin.PULL_UP),    # MISO (Master In Slave Out)
)

# LoRa configuration with default parameters
lora_default = {
    'frequency': 868E6,             # Frequency for Europe (868 MHz ISM band)
    'tx_power_level': 14,           # Transmission power level (14 dBm)
    'signal_bandwidth': 125E3,      # Signal bandwidth (125 kHz)
    'spreading_factor': 11,          # Spreading factor (7)
    'coding_rate': 8,               # Coding rate (4/5)
    'preamble_length': 8,           # Preamble length (8)
    'implicit_header': False,       # Explicit header mode
    'sync_word': 0x12,              # LoRa sync word
    'enable_crc': True              # Enable CRC for error detection
}

# --- SX1276 LoRa Driver Initialization ---
def lora_init():
    # Reset the SX1276
    reset_pin = Pin(lora_pins['reset'], Pin.OUT)
    reset_pin.value(0)
    time.sleep(0.01)  # Short delay
    reset_pin.value(1)
    # Initialize the SX1276 LoRa driver with default parameters
    lora = sx127x.SX127x(spi=lora_spi, pins=lora_pins, parameters=lora_default)
    # Confirm initialization
    print("LoRa modem initialized with default parameters.")
    return lora

#lora_init()                     # only for test
```