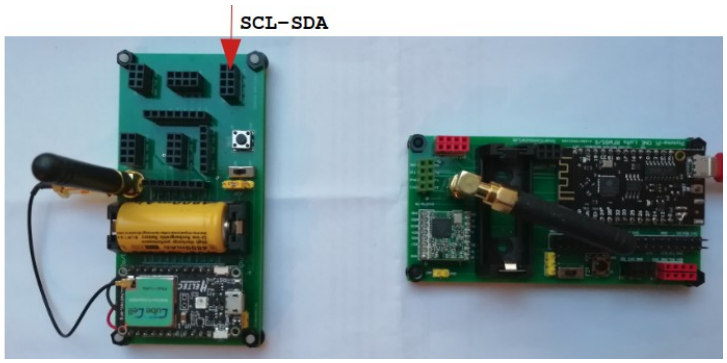
IoT Lab 1

Sending and receiving LoRa packets

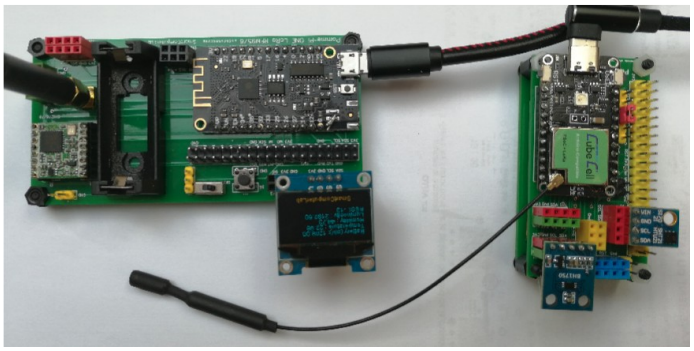
In this exercise we are going to use two different IoT DevKits

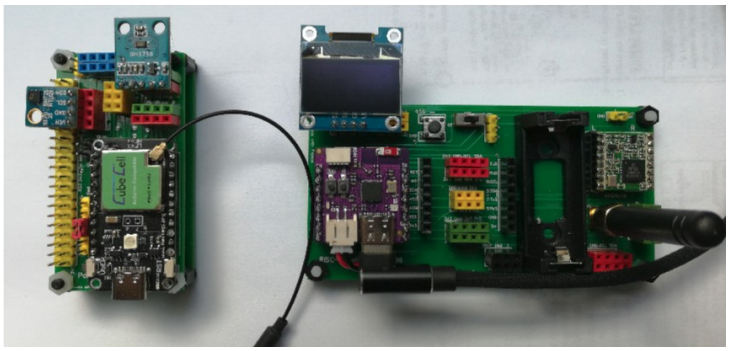
- Pomme-Pi ONE LoRa DevKit or Pomme-Pi RISC-V LoRa Devkit
- CubleCell LoRa/LoRaWAN Devkit or CubeCell Pico LoRa/LoRaWAN DevKit

as shown on the following figure:



The second configurations includes : **Pomme-Pi CubleCell Pico Devkit**





The third configurations includes :
Pomme-Pi CubleCell Pico and Pomme-Pi RISC-V Lora DevKit

The difference between CubeCell ONE and ZERO boards are in the format of the board and in interface configuration. CubeCell ONE integrates pseudo micro-bus interface while CubeCell ZERO has a GPIO interface .
CubeCell based board is designed for low power consumption, it includes a battery and a connector for a small solar panel. Note that CubeCell PSoC incorporates an ARM – CORTEX-M0 processor specifically useful for low power devices.

ESP32 based board integrates external LoRa modem with SX1276/8 chip. With WiFi and BT connectivity is well suited for building LoRa-WiFi gateways to IoT servers and brokers in the Cloud.

ESP32 is a programmable SoC with two high performance RISC processors – LX06 from Extensa. The SoC integrates accelerators for cryptographic functions such as AES.

The use of these components allows us to cover two essential processing elements for IoT architectures.

Content

You should install the libraries (tools) for ESP32 and CubeCell boards. Use preferences and board manager to do it.
Remember that you are using **ESP32** – **Lolin D32** board and **CubeCell** board – **HTCC-AB01** board.
The **MCUs of these boards are completely different**:
ESP32 uses **Extensa-LX06** micro-processor while CubeCell integrates an **ARM-CORTEX-M0** microprocessor.
Pomme-Pi RISC-V Lora DevKit is built with **ESP32C3 SoC** that is based on RISC-V processor.

Both ESP32 (LX06) and CubeCell (ARM-Cortex-M0) **are RISC processors**.

However both are not "**open ISA**" processors as **RISC-V** (similar to MIPS -RISC !)

The aim of these labs is to experiment with a large spectrum of processing units used to "power" modern IoT Architectures.

1.1 Sender side on CubeCell or CubeCell Pico board

The following code implements basic terminal node operations. They include four major stages (explained separately) that are:

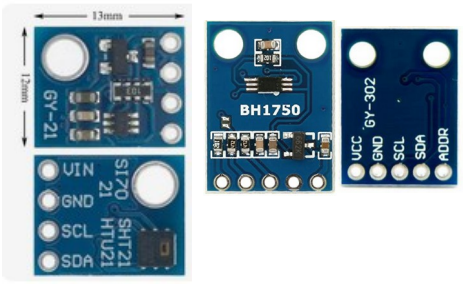
1. **low power stage** exploiting `lowPowerHandler`
2. **sensing stage** exploiting two sensors: **SHT21** (T/H) and **BH1750** (L) – these **may be replaced** by other sensors
3. **sending stage** for LoRa packet
4. **receiving stage** for ACK LoRa packet

Stages 2 to 4 are **high power stages**

The code itself may be segmented according to these operational stages. The initial part of the code (as usual) contains the includes of external codes providing the additional functionalities such as sensor or modem drivers. The drivers communicate with control/data registers integrated with the sensor/modem modules.

1.1. 1 Instantiation of sensor drivers:

```
#include <SHT21.h>
SHT21 sht;


#include <BH1750.h>
BH1750 bh;
```



1.1.2 Instantiation of Lora modem(s) drivers

1.1.2.1 LoRa RFM modules

The LoRa radio communication is implemented via **SX1262** chip from **Semtech**. This is second generation of LoRa modems from Semtech, the first is **SX1278/76** . This first generation is used in our receiver/gateway board with ESP32 SoC.

Fig 1. SX1278 (RFM) versus SX1262 (RFM) module
Both modules are radio-compatible, however the **SX1262** provides a new signal that is **BUSY**. The presence of this signal and associated functionality is relevant for the drivers.
Note that CubeCell board (POM) integrates SX1262, while the **ESP32-Pomme-Pi** board includes **SX1276/78** modem. Consequently we are using **different LoRa drivers/libraries** for these two boards.

1.1.2.2 LoRa RFM radio parameters

LoRa radio parameters define the characteristics of the radio channel modulation. The most important are:

- the **frequency** in ISM band (**434MHz**, **868MHz**, **2.4GHz**) for example – `868500000` or `8685E5 Hz`
- the **spreading factor** (SF) defining the number of modulations per data bit – `9` means **2^9** (**512**)
- the signal bandwidth defining the width of the modulation channel in KHz – **125-500 KHz**

- the coding rate defining the number of symbol bits to carry 4-bit data – **4/5** to **4/8** (min 5, max 8), the redundant bits are used to error detection/correction at the receiving side

Other parameters are more specific, for example **`LORA_IQ_INVERSION_ON`** parameter indicates the way modulation frequencies are used to carry the binary symbols in **UP** or **DOWN** fashion. The separate modulations are carried by separate frequency values that are increasing (**chirp-up**) or decreasing in value (**chirp-down**).

```
#define RF_FREQUENCY                                868500000 // Hz
#define TX_OUTPUT_POWER                             14        // dBm
#define LORA_BANDWIDTH                              0         // [0: 125 kHz,
                                                              //  1: 250 kHz,
                                                              //  2: 500 kHz,
                                                              //  3: Reserved]
#define LORA_SPREADING_FACTOR                       9         // [SF7..SF12]
#define LORA_CODINGRATE                             1         // [1: 4/5,
                                                              //  2: 4/6,
                                                              //  3: 4/7,
                                                              //  4: 4/8]
#define LORA_PREAMBLE_LENGTH                        8         // Same for Tx and Rx
#define LORA_SYMBOL_TIMEOUT                         0         // Symbols
#define LORA_FIX_LENGTH_PAYLOAD_ON                  false
#define LORA_IQ_INVERSION_ON                        false
#define RX_TIMEOUT_VALUE                            1000

static RadioEvents_t RadioEvents;
```

All these parameters are used in the **`setup()`** part of the code to initialize the modem operation. In the code , there are 3 functions called upon the following **events**. The events (signals) are generated by the LoRa modem depending on the state of communication. For example **`TxDone`** event, meaning that the data packet is effectively transmitted, activates **`OnTxDone`** function. In this function the user may put all operations that are associated to this **event**.

```
  RadioEvents.TxDone = OnTxDone;
  RadioEvents.TxTimeout = OnTxTimeout;
  RadioEvents.RxDone = OnRxDone;
  Radio.Init( &RadioEvents );

..
void OnTxDone(void)
{
  Serial.print("TX done!\n");
  turnOnRGB(0,0);
}
```

In the **`setup()`** part the **`program`** uses defined LoRa parameters to **`configure`** effectively the modem operations. Note that we do not provide any details concerning the physical connection on SPI bus between the MCU and the SX1262 modem. This information is provided implicitly via the #include "**`LoRaWan_APP.h`**" library associated to our CubeCell PSOC (*Programmable System on Chip*). This is not the case for our ESP32+LoRa (SX1276/8) board (Pomme-Pi ONE LoRa)

```
  Radio.SetChannel( RF_FREQUENCY );
  Radio.SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                                 LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                                 LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                                 true, 0, 0, LORA_IQ_INVERSION_ON, 3000 );

  Radio.SetRxConfig( MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                                  LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                                  LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                                  0, true, 0, 0, LORA_IQ_INVERSION_ON, true );
```

1.1.2.3 Stage 1 : Low Power operation with CubeCell

One of the most important and interesting features of **CubeCell PSOC** is the possibility to enter into low power mode with **`lowPowerHandler`**.

The essential part of the code is provided by the following line:

```
while (!sleepTimerExpired) lowPowerHandler();
```

In order to control the timing , it is the boolean value of **`sleepTimerExpired`** some additional utilities are required:

```
TimerEvent_t sleepTimer;

bool sleepTimerExpired;  //Records whether our sleep/low power timer expired

static void wakeUp()
{
  sleepTimerExpired=true;
}

static void lowPowerSleep(uint32_t sleeptime)
{
  sleepTimerExpired=false;
  TimerInit(&sleepTimer, &wakeUp );
  TimerSetValue( &sleepTimer, sleeptime );
  TimerStart(&sleepTimer );
  // Low power handler also gets interrupted by other timers
  // So wait until our timer had expired
  while (!sleepTimerExpired) lowPowerHandler();
  TimerStop( &sleepTimer );
}
```

1.1.2.4 Stage 2: Sensing the physical parameters

In our example we use two simple sensors: **SHT21** for temperature (T) and humidity (H), and **BH1750** for luminosity (L).
Note that these sensors may be replaced by other sensors such as AiR Quality (**SGP30**) or GPS modem.
The power to the sensor is programmatically controlled by **`Vext`** signal in order to save the power during the low_power stage.

```
  pinMode(Vext, OUTPUT); delay(100);
  digitalWrite(Vext, LOW);  delay(100);
  Wire.begin();delay(100);

  sdp.data[1]=(float)read_Temp();Serial.println(sdp.data[1]);delay(100);
  sdp.data[2]=(float)read_Humi();Serial.println(sdp.data[2]);delay(100);
  sdp.data[3]=(float)read_Lumi();Serial.println(sdp.data[3]);delay(100);

  Wire.end();  delay(100);
```

CubeCell provides a means to sense the state of the battery that may be required by the user. This operation is done by the following part code - function:

```
uint16_t read_Bat()
{
  uint16_t v;
  pinMode(VBAT_ADC_CTL,OUTPUT);
  digitalWrite(VBAT_ADC_CTL,LOW);
  v=analogRead(ADC)+550; //*2;
  pinMode(VBAT_ADC_CTL, INPUT);
  return v;
}
```

1.1.2.5 Stage 3 - sending LoRa packet

After the acquisition of the sensor data we can build and send LoRa packet. The basic format of the packet is defined as follows. The packet content may be seen as 16-byte frame or 4 floating point values.

```
typedef union pack
{
 uint8_t frame[16]; // byte frame
  float  data[4];   // 4 places for sensor data
} pack_t;

pack_t sdp,rdp;  // emission - send/receive data packet
```

To send a packet we run:

```
Radio.Send((uint8_t *)sdp.frame,16);
delay(500);     // min sending time before the reception
```

There are two events and two associated functions provided to terminate the transmission/sending (Tx) operation:

```
  RadioEvents.TxDone = OnTxDone;
  RadioEvents.TxTimeout = OnTxTimeout;

void OnTxDone(void)
{
  Serial.print("TX done!\n");
  turnOnRGB(0,0);
}

void OnTxTimeout( void )
{
    turnOffRGB();
    Radio.Sleep( );
    Serial.print("TX Timeout......");
```

A correct transmission terminates with **`TxDone` event** and **`OnTxDone` function**, an incorrect one with **`TxTimeout` event** and **`OnTxTimeout` function**.

1.1.2.6 Stage 4 - receiving LoRa packet (ACK)

After the emission of the **data packet** the terminal node awaits on an **ACK packet** to be sent from the gateway node. The **ACK packet** may contain the next **time_to_sleep** for the next low power period. This parameter is useful when several terminals (with addressed packets) need to be scheduled by the gateway.

To capture the received packet we use **`RxDone` event** and **`OnRxDone` function**. The **`OnRxDone`** function receives the **`payload (16-byte` frame`)`**, the **`size`** of the payload, and **`rssi`** plus **`snr`** values.

```
  RadioEvents.RxDone = OnRxDone;
..
bool lora_idle = true;
..
  while(lora_idle)
    {
    turnOffRGB();
    lora_idle = false;
    Serial.println("RX mode");
    Radio.Rx(0);
    }
}

void OnRxDone( uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr )
{
  pack_t rp;
    memcpy(rp.frame, payload, size );
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.printf("\nrssi=%d ,length=%d, timeout=%d\n",rssi,size,(int)rp.data[0]);
    ttsleep = (int)rp.data[0]*100;delay(100);
    lora_idle = true;turnOffRGB();
}
```

1.1.3 Complete code for CubeCell board

```
#include "LoRaWan_APP.h"
#include "Arduino.h"
#include <SHT21.h>
SHT21 sht;
#include <BH1750.h>
BH1750 bh;
static uint16_t counter=0;

#ifndef LoraWan_RGB
#define LoraWan_RGB 0
#endif
#define RF_FREQUENCY                                868500000 // Hz
#define TX_OUTPUT_POWER                             14        // dBm
#define LORA_BANDWIDTH                              0         // [0: 125 kHz,
                                                              //  1: 250 kHz,
                                                              //  2: 500 kHz,
                                                              //  3: Reserved]
#define LORA_SPREADING_FACTOR                       9         // [SF7..SF12]
#define LORA_CODINGRATE                             1         // [1: 4/5,
                                                              //  2: 4/6,
                                                              //  3: 4/7,
                                                              //  4: 4/8]
#define LORA_PREAMBLE_LENGTH                        8         // Same for Tx and Rx
#define LORA_SYMBOL_TIMEOUT                         0         // Symbols
#define LORA_FIX_LENGTH_PAYLOAD_ON                  false
#define LORA_IQ_INVERSION_ON                        false
#define RX_TIMEOUT_VALUE                            1000
#define BUFFER_SIZE                                 128 // Define the payload size here

static RadioEvents_t RadioEvents;
void OnTxDone( void );
void OnTxTimeout( void );

int16_t rssi,rxSize;
uint8_t sent=0;


TimerEvent_t sleepTimer;  //Some utilities for going into low power mode
bool sleepTimerExpired;

static void wakeUp()
{
  sleepTimerExpired=true;
}

static void lowPowerSleep(uint32_t sleeptime)
{
  sleepTimerExpired=false;
  TimerInit( &sleepTimer, &wakeUp );
  TimerSetValue( &sleepTimer, sleeptime );
  TimerStart( &sleepTimer );
  //Low power handler also gets interrupted by other timers
  //So wait until our timer had expired
  while (!sleepTimerExpired) lowPowerHandler();
  TimerStop( &sleepTimer );
}

typedef union pack
{
 uint8_t frame[16]; // trames avec octets
  float  data[4];   // 4 valeurs en virgule flottante
} pack_t;

pack_t sdp;  // paquet d'émission

uint16_t read_Bat()
{
  uint16_t v;
  pinMode(VBAT_ADC_CTL,OUTPUT);
  digitalWrite(VBAT_ADC_CTL,LOW);
  v=analogRead(ADC)+550; //*2;
  pinMode(VBAT_ADC_CTL, INPUT);
  return v;
}
```

```
float read_Temp()
{
  float t;
  t = sht.getTemperature();  // get temp from SHT
  return t;
}
float read_Lumi()
{
  float l;
  digitalWrite(Vext,LOW); // start power before activating Wire
  delay(200);
  bh.begin();
  delay(200);
  l = bh.readLightLevel();
  return l;
}

float read_Humi()
{
  float h;
  h = sht.getHumidity();  // get temp from SHT
  return h;
}

void setup()
{
  Serial.begin(9600);
  rssi=0;
  RadioEvents.TxDone = OnTxDone;
  RadioEvents.TxTimeout = OnTxTimeout;
  RadioEvents.RxDone = OnRxDone;
  Radio.Init( &RadioEvents );
  Radio.SetChannel( RF_FREQUENCY );
  Radio.SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                                  LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                                  LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                                  true, 0, 0, LORA_IQ_INVERSION_ON, 3000 );

  Radio.SetRxConfig( MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                                   LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                                   LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                                   0, true, 0, 0, LORA_IQ_INVERSION_ON, true );
}

int ttsleep=20000;
bool lora_idle = true;

void loop()
{
  counter++;
  Serial.printf("\ntime_to_sleep=%d ms\n", ttsleep);delay(100);
  lowPowerSleep(ttsleep);
  Serial.printf("\nBack from sleep %d, counter=%d\n", millis(),counter);
  sdp.data[0]=(float)read_Bat();Serial.println(sdp.data[0]);
  pinMode(Vext, OUTPUT); delay(100);
  digitalWrite(Vext, LOW);  delay(100);
  Wire.begin();delay(100);
  sdp.data[1]=(float)read_Temp();Serial.println(sdp.data[1]);delay(100);
  sdp.data[2]=(float)read_Humi();Serial.println(sdp.data[2]);delay(100);
  sdp.data[3]=(float)read_Lumi();Serial.println(sdp.data[3]);delay(100);
  Wire.end();  delay(100);
  digitalWrite(Vext, HIGH); delay(100);
  turnOnRGB(COLOR_SEND,0);
  Radio.Send((uint8_t *)sdp.frame,16);
  delay(500);  // min sending time before the reception
  while(lora_idle)
    {
    turnOffRGB();
    lora_idle = false;
    Serial.println("RX mode");
    Radio.Rx(0);
    }
}
```

```
void OnRxDone( uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr )
{
  pack_t rp;
    memcpy(rp.frame, payload, size );
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.printf("\nrssi=%d ,length=%d, timeout=%d\n",rssi,size,(int)rp.data[0]);
    ttsleep = (int)rp.data[0]*100;delay(100);
    lora_idle = true; turnOffRGB();
}

void OnTxDone( void )
{
  Serial.print("TX done!\n");
  turnOnRGB(0,0);
}

void OnTxTimeout( void )
{
    turnOffRGB();
    Radio.Sleep( );
    Serial.print("TX Timeout......");
}
```

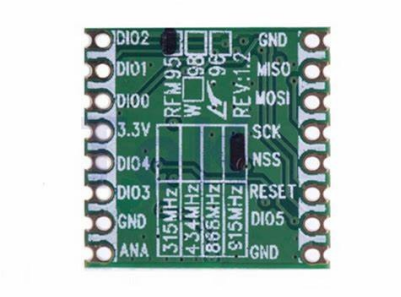
Terminal output:

```
..
Back from sleep 22100108, counter=2934
1212.00
21.95
46.93
177.50
TX done!

time_to_sleep=5500 ms

Back from sleep 22107633, counter=2935
1212.00
21.93
45.84
177.50
TX done!

time_to_sleep=5500 ms

..
```

Note that the **ACK packet** carries **`time_to_sleep`** value that is applied to the next l**`ow_power`** step.

1.2 Receiver side on Pomme-Pi ONE LoRa (`Lolin D32`)

The following is the code for receiver on **Pomme-Pi ONE LoRa** board. This MCU (ESP32) integrates two LX06 RISC cores and WiFi/BT/BLE modems as well as a number of bus interfaces and functional blocks for low power operation and cryptography.
This ESP32 it is not able to operate at the same low_power level as CubeCell PsoC and it consumes **several mA** in the **deep_sleep** mode. However the integrated WiFi modem allow us to send/receive data to/from the Internet. That is why we are using it as LoRa-WiFi gateway.
The external SX1276 (RFM) module provides us with LoRa radio communication. This module is integrated on our Pomme-Pi ONE LoRa board.
Having different Lora modem and different integration way we must define the connection signals for SPI bus and some additional control lines. These lines correspond to the external pins of **Lolin D32** MCU board that is our "main" board.




```
#define SCK     18   // GPIO18 -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    23   // GPIO23 -- SX127x's MOSI
#define SS       5   // GPIO05 -- SX127x's NSS
#define RST     15   // GPIO15 -- SX127x's RESET
#define DI0     25   // GPIO25 -- SX127x's DIO0 - SX127x's IRQ(Interrupt Request)
```

Note that this **SX1276** (RFM) module has no **BUSY** line that is available in **SX1262** module. This also means that the communication operations concerning these modems are slightly different and the drivers/libraries to use these modems are not the same.

The essential LoRa radio parameters are defined as:

```
#define freq    8685E5
#define sf 9
#define sb 125E3
```

Remainder: They correspond to our CubeCell based sender:

```
#define RF_FREQUENCY                                868500000 // Hz
#define TX_OUTPUT_POWER                             14        // dBm
#define LORA_BANDWIDTH                              0         // [0: 125 kHz,
                                                              //  1: 250 kHz,
                                                              //  2: 500 kHz,
                                                              //  3: Reserved]
#define LORA_SPREADING_FACTOR                       9         // [SF7..SF12]
#define LORA_CODINGRATE                             1         // [1: 4/5,
                                                              //  2: 4/6,
                                                              //  3: 4/7,
                                                              //  4: 4/8]
```

These parameters are applied in the setup() function by:

```
if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
LoRa.setCodingRate4(5);   // coding rate 4/5
```

As in the CubeCell code we use **`typedef union`** construct to send/receive LoRa packets in a simple but formatted way.

```
typedef union
{
 uint8_t frame[16];    // byte frame
  float  data[4];      // 4 floating point values for sensor data
} pack_t;

pack_t rdp;            // received data packet
```

We receive tha data packets and send the ACK packets in the main **`loop()`**. This is not the most elegant solution but **it works** !.

```
void loop()
{
int packetLen; char b1[32];
packetLen=LoRa.parsePacket();
if(packetLen==16)
  {
  int i=0;
  while (LoRa.available()) {
    rdp.frame[i]=LoRa.read();i++;
    }
  rssi=LoRa.packetRssi();  // force du signal en réception en dB
  Serial.printf("V:%2.2f,T:%2.2f,H:%2.2f\n",rdp.data[0],rdp.data[1],rdp.data[2]);
  Serial.printf("RSSI=%d\n",rssi);sprintf(b1,"V:=%4.2f",rdp.data[0]);
  disp("received","packet",b1);
  delay(1000);
  LoRa.beginPacket();
  sdp.data[0]=55.5;sdp.data[1]=99.9;
  LoRa.write(sdp.frame,16);
  LoRa.endPacket();
  disp("sent","packet","ACK");
  }
}
```

The sdp.data[0] field in the **ACK frame** is used to send the “next” **deep_sleep** period in **low_power** mode for the CubeCell terminal. Currently it is a constant but it can be modified or “modulated” to organize the reception of the packets provided by several competing terminal nodes.

Note that the content received data packet is displayed on the OLED screen connected to I2C bus.

```
void disp(char *d1,char *d2,char *d3)
{
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);  // ArialMT_Plain_10
  display.drawString(0, 0, d1);
  display.drawString(0, 18, d2);
  display.drawString(0, 36, d3);
  display.setFont(ArialMT_Plain_10);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}
```

1.2.1 Complete code of the receiver on ESP32/D32

```
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);
#define SCK     18   // GPIO18 -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    23   // GPIO23 -- SX127x's MOSI
#define SS       5   // GPIO05 -- SX127x's CS
#define RST     15   // GPIO15 -- SX127x's RESET
#define DI0     25   // GPIO25 (integrated modem)  -- SX127x's IRQ(Interrupt Request)
#define freq    8685E5
#define sf 9
#define sb 125E3

union pack
{
 uint8_t frame[16]; // trames avec octets
  float  data[4];   // 4 valeurs en virgule flottante
} rdp,sdp ;  // paquet d'émission

void disp(char *d1,char *d2,char *d3)
{
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);  // ArialMT_Plain_10
  display.drawString(0, 0, d1);
  display.drawString(0, 18, d2);
  display.drawString(0, 36, d3);
  display.setFont(ArialMT_Plain_10);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}

void setup() {
  Serial.begin(9600);  delay(1000);
  Wire.begin(12,14);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) { Serial.println("Starting LoRa failed!"); while (1);
  }
  Serial.println("Starting LoRa OK!"); delay(1000);
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sb);
  LoRa.setCodingRate4(5);
}

int rssi;

void loop()
{
int packetLen; char b1[32];
packetLen=LoRa.parsePacket();
if(packetLen==16)
  {
  int i=0;
  while (LoRa.available()) { rdp.frame[i]=LoRa.read();i++; }
  rssi=LoRa.packetRssi();  // force du signal en réception en dB
  Serial.printf("V:%2.2f,T:%2.2f,H:%2.2f\n",rdp.data[0],rdp.data[1],rdp.data[2]);
  Serial.printf("RSSI=%d\n",rssi);sprintf(b1,"V:=%4.2f",rdp.data[0]);
  disp("received","packet",b1);
  delay(1000);
  LoRa.beginPacket();
  sdp.data[0]=55.5;sdp.data[1]=99.9;
  LoRa.write(sdp.frame,16);
  LoRa.endPacket();
  disp("sent","packet","ACK");
  }
}
```

1.3 Complete IoT architecture with LoRa-WiFi gateway to TS

Let us complete the previous example with some features such as:

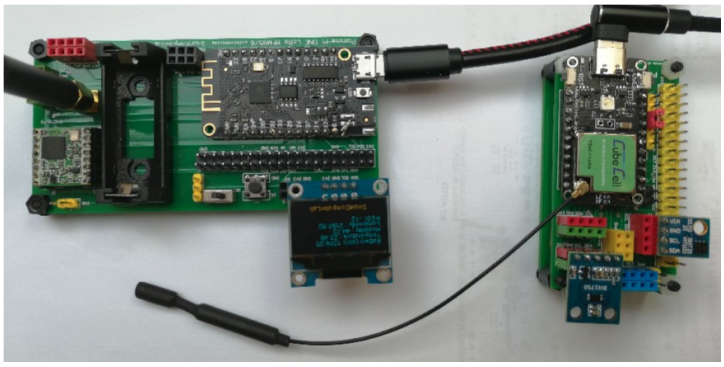
At the **sender side** on CubeCell:

- **reception of ACK packets** after data packets transmission

At the **receiver side** on ESP/D32

- **sending the ACK packets** to the terminal
- **sending the sensor data** to ThingSpeak server

These new features transform our **receiver into a gateway**.



The above figure shows both boards : Pomme-Pi ONE LoRa as receiver/gateway and Pomme-Pi CubeCell Pico as the terminal node.

1.3.1 The sender on CubeCell

The following code implements complete communication cycle with the receiver/gateway that operates in four stages:

1. **`low_power`** stage
2. **sensing** stage – to get the **sensors data**
3. **transmission** stage – to send the **data packet**
4. **reception** stage – to receive **ACK packet** with **the next low_power stage period**

In principle this code is an extension of the previous sender code with the **additional ACK reception** stage. After the transmission stage , the program waits for the incoming ACK packet in **`while(lora_idle)`** loop. In this loop it turns on the **`Radio`** reception with **`Radio.Rx(0);`** and waits for an interruption on **`OnRxDone`** routine.
When the packet arrives it is captured via this routine. To simplify the code, the received ACK packet has the same format as the data packet. **`rp.data[0]`** contains the **new time period** for **`low_power`** stage.

```
// preceding 3 stages of the code
…..
  while(lora_idle)
    {
    turnOffRGB();
    lora_idle = false;
    Serial.println("RX mode");
    Radio.Rx(0);
    }
}

void OnRxDone( uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr )
{
  pack_t rp;
    memcpy(rp.frame, payload, size );
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.printf("\nrssi=%d ,length=%d, timeout=%d\n",rssi,size,(int)rp.data[0]);
    ttsleep = (int)rp.data[0]*200;delay(100);
    lora_idle = true;turnOffRGB();
}
```

1.3.2 Complete code of the sender on CubeCell

```
#include "LoRaWan_APP.h"
#include "Arduino.h"
#include <SHT21.h>
SHT21 sht;
#include <BH1750.h>
BH1750 bh;
static uint16_t counter=0;

#ifndef LoraWan_RGB
#define LoraWan_RGB 0
#endif
#define RF_FREQUENCY                                868500000 // Hz
#define TX_OUTPUT_POWER                             14        // dBm
#define LORA_BANDWIDTH                              0         // [0: 125 kHz,
                                                              //  1: 250 kHz,
                                                              //  2: 500 kHz,
                                                              //  3: Reserved]
#define LORA_SPREADING_FACTOR                       9         // [SF7..SF12]
#define LORA_CODINGRATE                             1         // [1: 4/5,
                                                              //  2: 4/6,
                                                              //  3: 4/7,
                                                              //  4: 4/8]
#define LORA_PREAMBLE_LENGTH                        8         // Same for Tx and Rx
#define LORA_SYMBOL_TIMEOUT                         0         // Symbols
#define LORA_FIX_LENGTH_PAYLOAD_ON                  false
#define LORA_IQ_INVERSION_ON                        false
#define RX_TIMEOUT_VALUE                            1000
#define BUFFER_SIZE                                 128 // Define the payload size here

static RadioEvents_t RadioEvents;
void OnTxDone( void );
void OnTxTimeout( void );
void OnRxTimeout( void );

int16_t rssi,rxSize;
uint8_t sent=0;

//Some utilities for going into low power mode
TimerEvent_t sleepTimer;
//Records whether our sleep/low power timer expired
bool sleepTimerExpired;

static void wakeUp()
{
  sleepTimerExpired=true;
}

static void lowPowerSleep(uint32_t sleeptime)
{
  sleepTimerExpired=false;
  TimerInit( &sleepTimer, &wakeUp );
  TimerSetValue( &sleepTimer, sleeptime );
  TimerStart( &sleepTimer );
  //Low power handler also gets interrupted by other timers
  //So wait until our timer had expired
  while (!sleepTimerExpired) lowPowerHandler();
  TimerStop( &sleepTimer );
}

//////////////////////////////////////////////////

typedef union pack
{
 uint8_t frame[16]; // trames avec octets
  float  data[4];   // 4 valeurs en virgule flottante
} pack_t;

pack_t sdp;  // paquet d'émission

uint16_t read_Bat()
{
  uint16_t v;
  delay(40);
  pinMode(VBAT_ADC_CTL,OUTPUT);
```

```
  digitalWrite(VBAT_ADC_CTL,LOW);
  v=analogRead(ADC)+550; //*2;
  pinMode(VBAT_ADC_CTL, INPUT);
  return v;
}

float read_Temp()
{
  float t;
  t = sht.getTemperature();  // get temp from SHT
  return t;
}
float read_Lumi()
{
  float l;
  digitalWrite(Vext,LOW); // start power before activating Wire
  delay(200);
  bh.begin();
  delay(200);
  l = bh.readLightLevel();
  return l;
}

float read_Humi()
{
  float h;
  h = sht.getHumidity();  // get temp from SHT
  return h;
}

void setup()
{
  Serial.begin(9600);
  rssi=0;
  RadioEvents.TxDone = OnTxDone;
  RadioEvents.TxTimeout = OnTxTimeout;
  RadioEvents.RxDone = OnRxDone;
  Radio.Init( &RadioEvents );
  Radio.SetChannel( RF_FREQUENCY );
  Radio.SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                                 LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                                 LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                                 true, 0, 0, LORA_IQ_INVERSION_ON, 3000 );

  Radio.SetRxConfig( MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                                  LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                                  LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                                  0, true, 0, 0, LORA_IQ_INVERSION_ON, true );

}

int ttsleep=20000;
bool lora_idle = true;

void loop()
{
  counter++;
  Serial.printf("\ntime_to_sleep=%d ms\n", ttsleep);delay(100);
  lowPowerSleep(ttsleep);
  Serial.printf("\nBack from sleep %d, counter=%d\n", millis(),counter);
  sdp.data[0]=(float)read_Bat();Serial.println(sdp.data[0]);
  pinMode(Vext, OUTPUT); delay(100);
  digitalWrite(Vext, LOW);  delay(100);
  Wire.begin();delay(100);

  sdp.data[1]=(float)read_Temp();Serial.println(sdp.data[1]);delay(100);
  sdp.data[2]=(float)read_Humi();Serial.println(sdp.data[2]);delay(100);
  sdp.data[3]=(float)read_Lumi();Serial.println(sdp.data[3]);delay(100);

  Wire.end();  delay(100);
  digitalWrite(Vext, HIGH); delay(100);
  turnOnRGB(COLOR_SEND,0);
  Radio.Send((uint8_t *)sdp.frame,16);
  delay(400);  // min sending time before the reception
```

```
  while(lora_idle)
    {
    turnOffRGB();
    lora_idle = false;
    Serial.println("RX mode");
    Radio.Rx(0);
    }
}

void OnRxDone( uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr )
{
  pack_t rp;
    memcpy(rp.frame, payload, size );
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.printf("\nrssi=%d ,length=%d, timeout=%d\n",rssi,size,(int)rp.data[0]);
    ttsleep = (int)rp.data[0]*200;delay(100);
    lora_idle = true;turnOffRGB();
}

void OnTxDone( void )
{
  Serial.print("TX done!\n");
  turnOnRGB(0,0);
  lora_idle = true;
}

void OnTxTimeout( void )
{
    turnOffRGB();
    Radio.Sleep( );
    Serial.print("TX Timeout......");
}
```

A fragment of printout of the terminal on CubeCell:

```
..
time_to_sleep=12600 ms

rssi=-18 ,length=16, timeout=64

Back from sleep 2198843, counter=139
759.00
23.51
41.56
1526.67
TX done!
RX mode

time_to_sleep=12800 ms

rssi=-17 ,length=16, timeout=78
```

Note that the **`timeout`** value is multiplied by 20 to obtain the next **`low_power`** period (**`time_to_sleep`**).

1.3.3 The receiver/gateway on ESP32/D32

As we have mentioned, the receiver becomes gateway via LoRa-WiFi link to the ThingSpeak server. At the same time it responds to the terminal node with an ACK packet.

These new activities are implemented as FreeRTOS tasks. The usage of task allows us to organize the code into indpendent behavioral blocks nd to provide a means for communication between the receiving routine activated by an **interruption** - **`onReceive(int packetSize)`** , and the sending tasks.

The reception routine sends/stores the received packet into the message queues :

```
QueueHandle_t ack_queue, ts_queue;
```

These queues are feeding the acknowledge and ThingSpeak tasks activated as follows:

```
  xTaskCreatePinnedToCore(
                    ACKTask,   /* Function to implement the task */
                    "ACKTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    2,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore1);  /* Core where the task should run */
  Serial.println("Task ACK created...");
  xTaskCreatePinnedToCore(
                    TSTask,   /* Function to implement the task */
                    "TSTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore0);  /* Core where the task should run */
  Serial.println("Task TS created...")
```

Note that we use 2 different ESP32 cores to run these tasks. Also the priorities of these tasks are not the same; **ACK task has higher priority** in order to respond immediately to the received data packet.

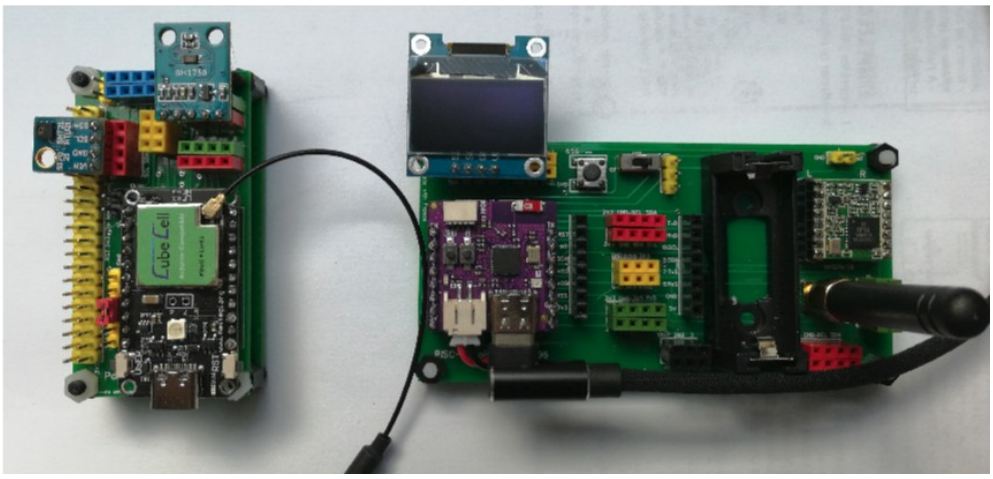
1.3.4 The complete code of the receiver/gateway on ESP32/D32

```
#include <WiFi.h>
#include "ThingSpeak.h"
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);
const char* ssid     = "Livebox-08B0";
const char* password = "G79ji6dtEptVTPWmZP";

#include <SPI.h>
#include <LoRa.h>
#define SCK     18   // GPIO18 -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    23   // GPIO23 -- SX127x's MOSI
#define SS       5   // GPIO05 -- SX127x's CS
#define RST     15   // GPIO15 -- SX127x's RESET
#define DI0     25   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    8685E5
#define sf 9
#define sb 125E3

typedef union
{
 uint8_t frame[16]; // frames with bytes
  float  data[4];   // 4 floating point values
} pack_t ;  // packet type

WiFiClient  client;
unsigned long myChannelNumber = 1697980;
const char * myWriteAPIKey = "4K897XNNHTW7I4NO";
int rssi=0;
QueueHandle_t ack_queue, ts_queue;  // queues for data packets
```

```
void disp(char *d1,char *d2,char *d3,char *d4,char *d5)
{
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);  // ArialMT_Plain_10
  display.drawString(0, 0, d1);
  display.drawString(0, 9, d2);
  display.drawString(0, 18, d3);
  display.drawString(0, 27, d4);
  display.drawString(0, 36, d5);
  display.setFont(ArialMT_Plain_10);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}

void onReceive(int packetSize)
{
pack_t rdp,sdp;
Serial.println("received packet");
  if(packetSize==16)
  {
    int i=0;
    while (LoRa.available())  {  rdp.frame[i]=LoRa.read();i++; }
    rssi=LoRa.packetRssi();
    xQueueReset(ts_queue); // to keep only the last element
    xQueueSend(ts_queue, &rdp, portMAX_DELAY);
    xQueueReset(ack_queue); // to keep only the last element
    xQueueSend(ack_queue, &rdp, portMAX_DELAY);
  }
}

static int taskCore1 = 1;
static int taskCore0 = 0;
void ACKTask( void * pvParameters )
{
  pack_t rdp,sdp;
  while(true)
  {
    xQueueReceive(ack_queue,rdp.frame,portMAX_DELAY);
    delay(1000);
    LoRa.beginPacket();
    sdp.data[0]=(float)random(60,80);sdp.data[1]=(float)random(10,80);
    LoRa.write(sdp.frame,16);
    LoRa.endPacket();
    Serial.printf("Sent ACK, tout=%2.2f\n",sdp.data[0]);
    LoRa.receive();  // put the radio into receive mode
  }
}

void TSTask( void * pvParameters )
{
  pack_t rdp,sdp;
  while(true)
  {
    char d1[32],d2[32],d3[32],d4[32], d5[32];
    xQueueReceive(ts_queue,rdp.frame,portMAX_DELAY); // default:portMAX_DELAY
    Serial.printf("V:%2.2f,T:%2.2f,H:%2.2f,L:%2.2f\
n",rdp.data[0],rdp.data[1],rdp.data[2],rdp.data[3]);
    Serial.printf("RSSI=%d\n",rssi);
    ThingSpeak.setField(1, rdp.data[0]);
    ThingSpeak.setField(2, rdp.data[1]);
    ThingSpeak.setField(3, rdp.data[2]);
    ThingSpeak.setField(4, rdp.data[3]);
    ThingSpeak.setField(5, rssi);
    int x = ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
    if(x == 200){
      Serial.println("Channel update successful.");
    }
    else{
      Serial.println("Problem updating channel. HTTP error code " + String(x));
    }
    sprintf(d1,"Battery (mV): %2.2f",rdp.data[0]);sprintf(d2,"Temperature : %2.2f",rdp.data[1]);
    sprintf(d3,"Humidity : %2.2f",rdp.data[2]);sprintf(d4,"Luminosity : %2.2f",rdp.data[3]);
    sprintf(d5,"RSSI: %d",rssi);
    disp(d1,d2,d3,d4,d5);
```

```
    delay(15000);
  }
}

void setup() {
  Serial.begin(9600);
  Wire.begin(12,14);
  Serial.print("[WiFi] Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);
  while(WiFi.status() != WL_CONNECTED)
    {
      Serial.print(".");
      delay(500);
    }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
  delay(500);
  ThingSpeak.begin(client);  // Initialize ThingSpeak
  delay(1000);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }

  Serial.println("Starting LoRa OK!");
  delay(1000);
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sb);
  LoRa.setCodingRate4(5);
  ack_queue = xQueueCreate(4,16); // queue for 4 data packets
  ts_queue = xQueueCreate(4,16); // queue for 4 data packets
  Serial.print("Starting to create task on core ");

  xTaskCreatePinnedToCore(
                    ACKTask,   /* Function to implement the task */
                    "ACKTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    2,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore1);  /* Core where the task should run */
  Serial.println("Task ACK created...");
  xTaskCreatePinnedToCore(
                    TSTask,   /* Function to implement the task */
                    "TSTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore0);  /* Core where the task should run */
  Serial.println("Task TS created...");
  LoRa.onReceive(onReceive);  // register the receive callback
  LoRa.receive();  // put the radio into receive mode
}

void loop()
{
  Serial.println("in the loop");
  delay(4000);
}
```

1.3.4 The receiver/gateway on ESP32C3 RISC-V



The following IoT architecture implements the same functionalities as the previous one with one major exception – the receiver/gateway runs on **ESP32C3 RISC-V** SoC (mono-core).
The receiving of LoRa packets is done via the callback routine that deposes the received packet in the message queue.
The queue is read by the main **`loop()`** that contains two parts. The first part of the code sends the ACK packets to the sender while the second part controlled by a dynamic delay with **`millis()`** function sends the sensor data to the **ThingSpeak** server.

Note that the WiFi connection is created with use of **`WiFiManager`** that allows us to provide the credential via a simple AP and WEB server created on the board.

1.3.4.1 The receiver/gateway code with callback routine

```
#include <LoRa.h>
#include <Wire.h>

#include <WiFiManager.h>
#include "ThingSpeak.h"

unsigned long myChannelNumber =1626377;
const char *myWriteAPIKey="3IN09682SQX3PT4Z" ;
const char *myReadAPIKey="9JVTP8ZHVTB9G4TT" ;

WiFiClient  client;

#include "SSD1306Wire.h"        // legacy: #include "SSD1306.h"

// Initialize the OLED display using Arduino Wire:
SSD1306Wire display(0x3c, 8, 10);   // ADDRESS, SDA, SCL

// with LoRa modem RFM95 and green RFM board - int and RST to solder

#define SS      5 // 26     // D0 - to NSS
#define RST     3  //16     // D4  - RST
#define DI0     2      // D8 - INTR

#define SCK     1      // D5 - CLK
#define MISO    0      // D6 - MISO
#define MOSI    4      // D7 - MOSI
#define BAND    8685E5
int sf=9;
long sbw=125E3;
typedef union
{
 uint8_t frame[16]; // frames with bytes
  float  sensor[4];   // 4 floating point values
} pack_t ;  // packet type
```

```
QueueHandle_t dqueue;  // queues for data packets

void onReceive(int packetSize)
{
pack_t rdp,sdp;  // receive buffer with pack_t format
int i=0;
  if (packetSize == 0) return;   // if there's no packet, return
  i=0;
  if (packetSize==16)   //
    {
    while (LoRa.available()) {  rdp.frame[i]=LoRa.read();i++;  }
      xQueueReset(dqueue); // to keep only the last element
      xQueueSend(dqueue, &rdp,portMAX_DELAY ); //portMAX_DELAY
    }
}

void disp(char *t1, char *t2, char *t3, char *t4)
{
  display.clear();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0, 0, t1);
  display.drawString(0, 16, t2);
  display.drawString(0, 32, t3);
  display.drawString(0, 48, t4);
  display.display();
}

void setup() {
  Serial.begin(9600);
  WiFi.mode(WIFI_STA);
  WiFiManager wm;
  // wm.resetSettings();
  bool res;
  res = wm.autoConnect("ESP32AP",NULL); // no password
  if(!res) Serial.println("Failed to connect");// ESP.restart();
  else  Serial.println("connected...yeey :)");

  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");

  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);

  delay(1000);
  Serial.println();Serial.println();
  Serial.println("Starting LoRa Receiver");
  display.init();
  display.flipScreenVertically();
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  else
   {
   Serial.println("Starting LoRa ok!");
   LoRa.setSpreadingFactor(sf);
   LoRa.setSignalBandwidth(sbw);
   dqueue = xQueueCreate(4,16); // queue for 4 data packets
   LoRa.onReceive(onReceive);  // pour indiquer la fonction ISR
   LoRa.receive();             // pour activer l'interruption (une fois)
   }
}

uint32_t mindel=15000;  // 15 seconds
long lastsent=0;

void loop()
{
pack_t rp,sdp;    // packet elements to send
  xQueueReceive(dqueue,rp.frame,portMAX_DELAY); // 6s,default:portMAX_DELAY
  delay(1000);
  LoRa.beginPacket();  // sending ACK packet
  sdp.sensor[0]=(float)random(60,80);sdp.sensor[1]=(float)random(10,80);
  LoRa.write(sdp.frame,16);
```

```
  LoRa.endPacket();
  LoRa.receive();
  if(millis()> lastsent+15000)
    {
    ThingSpeak.setField(1,rp.sensor[0]);
    ThingSpeak.setField(2,rp.sensor[1]);
    ThingSpeak.setField(3,rp.sensor[2]);
    ThingSpeak.setField(4,rp.sensor[3]);
    Serial.printf("d1=%2.2f,d2=%2.2f,d3=%2.2f,d4=%2.2f\n",rp.sensor[0],rp.sensor[1],
rp.sensor[2],rp.sensor[3]);
    while (WiFi.status() != WL_CONNECTED) { delay(500);  }
    int x = ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
    if(x == 200){Serial.println("Channel update successful.");}
    else { Serial.println("Problem updating channel. HTTP error code " + String(x));}
    lastsent=millis();
    }
}
```

1.3.4.2 The receiver/gateway code with packet parse task

In the following code we replaced **callback routine** connected to interruption line from the modem by an independent **task** that parses the input buffer of the modem. In this task we are allowed to build and send the ACK packet to the terminal sender. Note that we cannot do it in the callback routine.

```
#include <LoRa.h>
#include <Wire.h>

#include <WiFiManager.h>
#include "ThingSpeak.h"

unsigned long myChannelNumber =1626377;
const char *myWriteAPIKey="3IN09682SQX3PT4Z" ;
const char *myReadAPIKey="9JVTP8ZHVTB9G4TT" ;

WiFiClient  client;

#include "SSD1306Wire.h"        // legacy: #include "SSD1306.h"

// Initialize the OLED display using Arduino Wire:
SSD1306Wire display(0x3c, 8, 10);   // ADDRESS, SDA, SCL

// with LoRa modem RFM95 and green RFM board - int and RST to solder

#define SS      5 // 26     // D0 - to NSS
#define RST     3  //16     // D4  - RST
#define DI0     2      // D8 - INTR

#define SCK     1      // D5 - CLK
#define MISO    0      // D6 - MISO
#define MOSI    4      // D7 - MOSI
#define BAND    8685E5
int sf=9;
long sbw=125E3;
typedef union
{
 uint8_t frame[16]; // frames with bytes
  float  sensor[4];   // 4 floating point values
} pack_t ;  // packet type

QueueHandle_t ts_queue;  // queues for data packets
int rssi=0;

void ParseTask( void * pvParameters )
{
  pack_t rdp,sdp;
  while(true)
  {
    int packetLen; char b1[32];
    packetLen=LoRa.parsePacket();
    if(packetLen==16)
    {
      int i=0;
      while (LoRa.available()) {
      rdp.frame[i]=LoRa.read();i++;
```

```
    }
    rssi=LoRa.packetRssi();
    delay(1000);
    LoRa.beginPacket();  // sending ACK packet
    sdp.sensor[0]=(float)random(60,80);sdp.sensor[1]=(float)random(10,80);
    LoRa.write(sdp.frame,16);
    LoRa.endPacket();

    xQueueReset(ts_queue); // to keep only the last element
    xQueueSend(ts_queue, &rdp, portMAX_DELAY);
    }
  }
}

void disp(char *d1,char *d2,char *d3,char *d4,char *d5)
{
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);  // ArialMT_Plain_10
  display.drawString(0, 0, d1);
  display.drawString(0, 9, d2);
  display.drawString(0, 18, d3);
  display.drawString(0, 27, d4);
  display.drawString(0, 36, d5);
  display.setFont(ArialMT_Plain_10);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}

void setup() {
  Serial.begin(9600);
  WiFi.mode(WIFI_STA);
  WiFiManager wm;
  // wm.resetSettings();
  bool res;
  res = wm.autoConnect("ESP32AP",NULL); // no password
  if(!res) Serial.println("Failed to connect");// ESP.restart();
  else  Serial.println("connected...yeey :)");

  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");

  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);

  delay(1000);
  Serial.println();Serial.println();
  Serial.println("Starting LoRa Receiver");
  display.init();
  display.flipScreenVertically();
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  else
   {
   Serial.println("Starting LoRa ok!");
   LoRa.setSpreadingFactor(sf);
   LoRa.setSignalBandwidth(sbw);
   ts_queue = xQueueCreate(4,16); // queue for 4 data packets
   xTaskCreate(
                    ParseTask,   /* Function to implement the task */
                    "ParseTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL);       /* Task handle. */
  Serial.println("Task Parse created...");
   }
}

uint32_t mindel=15000;  // 15 seconds
long lastsent=0;
```

```
void loop()
{
pack_t rp;
char d1[32],d2[32],d3[32],d4[32], d5[32];

  xQueueReceive(ts_queue,rp.frame,portMAX_DELAY); // 6s,default:portMAX_DELAY

  if(millis()> lastsent+15000)
    {
    ThingSpeak.setField(1,rp.sensor[0]);
    ThingSpeak.setField(2,rp.sensor[1]);
    ThingSpeak.setField(3,rp.sensor[2]);
    ThingSpeak.setField(4,rp.sensor[3]);
    Serial.printf("d1=%2.2f,d2=%2.2f,d3=%2.2f,d4=%2.2f\n",rp.sensor[0],rp.sensor[1],
rp.sensor[2],rp.sensor[3]);
    while (WiFi.status() != WL_CONNECTED) { delay(500);  }
    int x = ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
    if(x == 200){Serial.println("Channel update successful.");}
    else { Serial.println("Problem updating channel. HTTP error code " + String(x));}
    sprintf(d1,"Battery (mV): %2.2f",rp.sensor[0]);sprintf(d2,"Temperature : %2.2f",rp.sensor[1]);
    sprintf(d3,"Humidity : %2.2f",rp.sensor[2]);sprintf(d4,"Luminosity : %2.2f",rp.sensor[3]);
    sprintf(d5,"RSSI: %d",rssi);
    disp(d1,d2,d3,d4,d5);
    lastsent=millis();
    }
}
```

To do:

After testing the presented IoT Architecture examples let us extend it with new features.

1. The IoT Architecture provides a means to operate with many terminals such as CubeCell boards. In this case the terminals must be identified by a **number** (**address**). We have to add it to the packet as a **header**. Note that different terminals may use **separate** ThingSpeak channels, how to do it ?

2. The packets sent by the terminal node(s) contain only the sensor data. To relay the data to ThingSpeak server the receiver/gateway needs to know the channel number and the writeKey on the server.
We also can provide this data directly from the terminal sender. In this case the LoRa packet must carry the corresponding channel number and the writeKey.

3. The terminals should not communicate with other terminals. In order to separate the communication between the terminals and the gateway we may use down-chirp/up-chirp modes. For example the terminal nodes send the “**up-chirp**” packets to the gateway and receive the “**down-chirp**” packets ACK from the gateway.

IoT Lab 2

AES – sending/receiving encrypted LoRa packers

In this lab we are going to build secure LoRa communication using **AES** . AES stands for "**Advanced Encryption Standard**."

- The AES algorithm is the industry-standard encryption protocol that protects sensitive information from traditional brute-force attacks.
- The two most common versions are **256-bit AES** (providing greater security) and **128-bit AES** (providing better performance during the encryption and decryption process).
- With current technology, **AES is uncrackable** through straightforward, brute-force attacks, and it is used in countless applications, from protecting top-secret or classified information in government agencies to keeping your personal data safe when stored on the cloud.
-

Since its adoption in **2000** as the industry standard, AES has become ubiquitous in every part of our digital lives. Unless you've been living in a cave for the last 20 years, you're pretty much guaranteed to have used software that uses it, even if you have no idea what it is.

There are two ways to provide AES functionalities: via software libraries or by hardware accelerators.
In this lab we are going to use both solutions.
The "hardware" solution is available with ESP32 that integrates a number of security related functions accelerators.
With CubeCell integrating a modest ARM-CORTEX-M0 we need to use "software" solution.
Let us start with ESP32.

2.1 AES "hardware" encryption/decryption for ESP32

The following example shows the use of AES encryption mechanism for embedded accelerator in ESP32 SoC. The encrypted/decrypted byte frame is 16-byte long – we are using 128-bit encryption key; in any case it has to be multiple of 16 bytes.

The main parts of the code are the **encrypt** and **decrypt** functions that exploit the "hardware" primitives provided in embedded tools as: **`mbedtls/aes.h`**

2.1.1 Local test of AES encryption/decryption on ESP32

```
#include "mbedtls/aes.h"

void encrypt(unsigned char *plainText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_enc(&aes,(const unsigned char*)key,strlen(key)*8);
  for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_ENCRYPT,
                          (const unsigned char*)(plainText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes);
}

void decrypt(unsigned char *chipherText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_dec( &aes, (const unsigned char*) key, strlen(key) * 8 );
    for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_DECRYPT,
                         (const unsigned char*)(chipherText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes );
}
```

```
void setup() {
  mbedtls_aes_context aes;
  Serial.begin(9600);delay(400);
  Serial.println(); Serial.println();

  char *key = "abcdefghijklmnop";                    // 128-bit encryption key
  unsigned char input[16]= {0xAB,0xCD,0xAC,0xB80x8A,0x77,0xA6,0xA6,0x8B,0xC1,0xD2,0xF3,0xBB,0xF1,
0xF2,0xF3};
  //unsigned char *input = (unsigned char *)"SmartComputerLab";
  unsigned char crypte[16], decrypte[16];

  Serial.println();  Serial.println();delay(400);
  for(int i=0; i<16;i++) Serial.print(input[i],HEX);
  Serial.println();
  encrypt(input, key, crypte, 1);
  for(int i=0; i<16;i++) Serial.print(crypte[i],HEX);
  Serial.println();delay(400);
  Serial.println();
  decrypt(crypte, key, decrypte,1);
  for(int i=0; i<16;i++) Serial.print(decrypte[i],HEX);
  Serial.println();delay(400);
}

void loop() { }
```

Execution result:

```
ABCDACB88A77A6A68BC1D2F3BBF1F2F3
E78C3A1E3D356A116AF49CC77EC773C

ABCDACB88A77A6A68BC1D2F3BBF1F2F3
```

2.2 AES “software” encryption/decryption for CubeCell

The encryption for CubeCell (ARM-CORTEX-M0) must be done via software. There are two necessary files to add (include) to your sketch: **`aes.h`** and **`aes.c`** . They are presented in the **Annex part of this lab**. They can also be found at the following link:

https://github.com/HelTecAutomation/CubeCell-Arduino/tree/master/cores/asr650x/lora/system/crypto

2.1.2 Local test of AES encryption/decryption on CubeCell

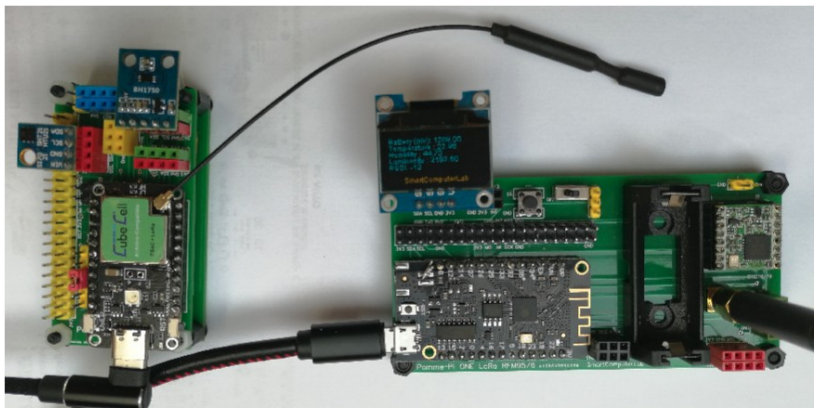
```
#include "aes.h"
aes_context ctx[2];
const uint8_t in[16]={ 0xAB, 0xCD, 0xAC, 0xB8,0x8A, 0x77, 0xA6, 0xA6,0x8B, 0xC1, 0xD2, 0xF3,0xBB,
0xF1, 0xF2, 0xF3};
uint8_t out[16], out1[16];
const uint8_t key[16]={ 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68,0x69, 0x6A, 0x6B, 0x6C,0x6D,
0x6E, 0x6F, 0x70};

void setup(){
  Serial.begin(9600);delay(300);
  Serial.println("startcrypt");delay(300);
  aes_set_key(key,
                16,          // len in bytes - length_type
               ctx );
}

void loop()
{
  Serial.println();Serial.println();
  for(int i=0; i<16;i++) Serial.print(in[i],HEX);
  Serial.println();delay(300);
  aes_encrypt(in,
              out1,
              ctx );
  for(int i=0; i<16;i++) Serial.print(out1[i],HEX);
  Serial.println();delay(300);
  aes_decrypt(out1,
              out,
              ctx );
for(int i=0; i<16;i++) Serial.print(out[i],HEX);
Serial.println();
delay(5000);
}
```

The resulting printout:

```
..
ABCDACB88A77A6A68BC1D2F3BBF1F2F3
E78C3A1E3D356A116AF49CC77EC773C
ABCDACB88A77A6A68BC1D2F3BBF1F2F3


ABCDACB88A77A6A68BC1D2F3BBF1F2F3
E78C3A1E3D356A116AF49CC77EC773C
ABCDACB88A77A6A68BC1D2F3BBF1F2F3


ABCDACB88A77A6A68BC1D2F3BBF1F2F3
E78C3A1E3D356A116AF49CC77EC773C
ABCDACB88A77A6A68BC1D2F3BBF1F2F3
..
```

2.3 Sending and receiving encrypted LoRa packets

Now, when we know how to encrypt and decrypt the chunks of the data we can build a simple IoT architecture with two boards:
Terminal on (CubeCell) and Receiver/Gateway on (ESP32) board.



2.3.1 Sending encrypted LoRa packets with CubeCell board

The following code integrates the above presented functions based on software “solution”. That means that we need to include aes.h/aes.cpp files into our project.
The LoRa packets are formed and sent in the same way as in the previous Lab1.
As input value to be encrypted we can choose between a numerical chain or character chain (“smartcomputerlab”) both carried by 16 bytes.

```
#include "LoRaWan_APP.h"
#include "Arduino.h"
#include "aes.h"
aes_context ctx[2];
uint8_t in[16]={ 0xAB, 0xCD, 0xAC, 0xB8,0x8A, 0x77, 0xA6, 0xA6,0x8B, 0xC1, 0xD2, 0xF3,0xBB, 0xF1,
0xF2, 0xF3};
uint8_t out[16], out1[16];
const uint8_t key[16]={ 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68,0x69, 0x6A, 0x6B, 0x6C,0x6D,
0x6E, 0x6F, 0x70};

#ifndef LoraWan_RGB
#define LoraWan_RGB 0
#endif

#define RF_FREQUENCY                                868500000 // Hz

#define TX_OUTPUT_POWER                             14        // dBm

#define LORA_BANDWIDTH                              0         // [0: 125 kHz,
                                                              //  1: 250 kHz,
                                                              //  2: 500 kHz,
                                                              //  3: Reserved]
#define LORA_SPREADING_FACTOR                       9         // [SF7..SF12]
#define LORA_CODINGRATE                             1         // [1: 4/5,
                                                              //  2: 4/6,
                                                              //  3: 4/7,
                                                              //  4: 4/8]
#define LORA_PREAMBLE_LENGTH                        8         // Same for Tx and Rx
#define LORA_SYMBOL_TIMEOUT                         0         // Symbols
#define LORA_FIX_LENGTH_PAYLOAD_ON                  false
#define LORA_IQ_INVERSION_ON                        false


#define RX_TIMEOUT_VALUE                            1000
#define BUFFER_SIZE                                 30 // Define the payload size here

char txpacket[BUFFER_SIZE];
char rxpacket[BUFFER_SIZE];
```

```
static RadioEvents_t RadioEvents;

float txNumber;
bool lora_idle=true;


typedef union
{
 uint8_t frame[16]; // trames avec octets
  float  data[4];   // 4 valeurs en virgule flottante
  char   mess[16];
} pack_t;

pack_t sdp ;  // paquet d'émission

void setup(){
  Serial.begin(9600);delay(300);
    txNumber=0;

    RadioEvents.TxDone = OnTxDone;
    RadioEvents.TxTimeout = OnTxTimeout;
    Radio.Init( &RadioEvents );
    Radio.SetChannel( RF_FREQUENCY );
    Radio.SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                                   LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                                   LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                                   true, 0, 0, LORA_IQ_INVERSION_ON, 3000 );

      aes_set_key(key,
                  16,         // len in bytes - length_type
                 ctx );

}

void loop()
{
  char *mess="smartcomputerlab";
  if(lora_idle == true)
  {
    memcpy(in,(uint8_t *)mess,16); // to replace the numerical values
    Serial.println("Clear packet");delay(300);
    Serial.println(mess);
    for(int i=0; i<16;i++) Serial.print(in[i],HEX);
    Serial.println();delay(300);
    aes_encrypt(in,
                out1,
                ctx );
    Serial.println("Sent crypted packet");
    for(int i=0; i<16;i++) Serial.print(out1[i],HEX);
    Serial.println();
    memcpy(sdp.frame,out1,16);
    Radio.Send( (uint8_t *)sdp.frame, 16 ); //send the package out
    lora_idle = false;
    Serial.println();delay(300);
    delay(3500);
  }
}

void OnTxDone( void )
{
  turnOffRGB();
  Serial.println("TX done......");
  lora_idle = true;
}

void OnTxTimeout( void )
{
  turnOffRGB();
  Radio.Sleep( );
  Serial.println("TX Timeout......");
  lora_idle = true;
}

Sent crypted packet
5A6336D130DF69497D69B6F50CC4AAB
```

Terminal output:

```
TX done......

Clear packet
smartcomputerlab
736D617274636F6D70757465726C6162
Sent crypted packet
5A6336D130DF69497D69B6F50CC4AAB

TX done......

Clear packet
smartcomputerlab
736D617274636F6D70757465726C6162
Sent crypted packet
5A6336D130DF69497D69B6F50CC4AAB
```

2.3.2 Receiving encrypted LoRa packets with ESP32 based board

The following code exploits the “hardware” solution to decrypt the arriving LoRa packets.

```
#include <SPI.h>
#include <LoRa.h>
#include "mbedtls/aes.h"

#define SCK     18   // GPIO18 -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    23   // GPIO23 -- SX127x's MOSI
#define SS       5   // GPIO05 -- SX127x's CS
#define RST     15   // GPIO15 -- SX127x's RESET
#define DI0     25   // GPIO25 (integrated modem)  -- SX127x's IRQ(Interrupt Request)
#define freq    8685E5
#define sf 9
#define sb 125E3

typedef union
{
 uint8_t frame[16];
 float  data[4];
 char     mess[16];
} pack_t;

pack_t rdp;

void encrypt(unsigned char *plainText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_enc(&aes,(const unsigned char*)key,strlen(key)*8);
  for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_ENCRYPT,
                        (const unsigned char*)(plainText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes);
}

void decrypt(unsigned char *chipherText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_dec( &aes, (const unsigned char*) key, strlen(key) * 8 );
    for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_DECRYPT,
                       (const unsigned char*)(chipherText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes );
}
```

```
void setup()
{
  mbedtls_aes_context aes;
  Serial.begin(9600);
  delay(1000);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  Serial.println("Starting LoRa OK!");
  delay(1000);
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sb);
  LoRa.setCodingRate4(5);
}

void loop() {
  //char key[16]= { 0x00, 0x01, 0x02, 0x03,0x00, 0x01, 0x02, 0x03,0x00, 0x01, 0x02, 0x03,0x00, 0x01,
0x02, 0x03};
  char *key = "abcdefghijklmnop";
  unsigned char input[16] = { 0xAB, 0xCD, 0xAC, 0xB8,0x8A, 0x77, 0xA6, 0xA6,0x8B, 0xC1, 0xD2,
0xF3,0xBB, 0xF1, 0xF2, 0xF3};
  //unsigned char *input = (unsigned char *)"SmartComputerLabSmartComputerLab";
  unsigned char crypte[16], decrypte[16];
  int packetLen; char b1[32];
  packetLen=LoRa.parsePacket();
  if(packetLen==16)
  {
  int i=0;
  while (LoRa.available()) {
    rdp.frame[i]=LoRa.read();i++;
    }
  memcpy(crypte,rdp.frame,16);

  Serial.println("\nReceived crypted packet");
  for(int i=0; i<16;i++) Serial.print(crypte[i],HEX);
  Serial.println();delay(400);
  decrypt(crypte, key, decrypte,1);
  Serial.println("Decrypted packet");
  for(int i=0; i<16;i++) Serial.print(decrypte[i],HEX);
  Serial.println();delay(400);
  Serial.printf("%16.16s\n",decrypte);
  }
}
```

The terminal output (for the received packets):

```
Received crypted packet
5A6336D130DF69497D69B6F50CC4AAB
Decrypted packet
736D617274636F6D70757465726C6162
smartcomputerlab

Received crypted packet
5A6336D130DF69497D69B6F50CC4AAB
Decrypted packet
736D617274636F6D70757465726C6162
smartcomputerlab
```

To do

1. As we know LoRa packets are not protected; so we have to add encryption to hide the payload. We can do it with **AES encryption** available in **software for CubeCell**, and integrated in **hardware with ESP32**. Complete the IoT architecture studied in Lab1 with **AES encryption for data packets**.

Annex

A.1 aes.h

```
/*
 ---------------------------------------------------------------------------
 Copyright (c) 1998-2008, Brian Gladman, Worcester, UK. All rights reserved.

 LICENSE TERMS

 The redistribution and use of this software (with or without changes)
 is allowed without the payment of fees or royalties provided that:

  1. source code distributions include the above copyright notice, this
     list of conditions and the following disclaimer;

  2. binary distributions include the above copyright notice, this list
     of conditions and the following disclaimer in their documentation;

  3. the name of the copyright holder is not used to endorse products
     built using this software without specific written permission.

 DISCLAIMER

 This software is provided 'as is' with no explicit or implied warranties
 in respect of its properties, including, but not limited to, correctness
 and/or fitness for purpose.
 ---------------------------------------------------------------------------
 Issue 09/09/2006

 This is an AES implementation that uses only 8-bit byte operations on the
 cipher state.
 */

#ifndef AES_H
#define AES_H

#if 1
#  define AES_ENC_PREKEYED  /* AES encryption with a precomputed key schedule  */
#endif
#if 1
#  define AES_DEC_PREKEYED  /* AES decryption with a precomputed key schedule  */
#endif
#if 0
#  define AES_ENC_128_OTFK  /* AES encryption with 'on the fly' 128 bit keying */
#endif
#if 0
#  define AES_DEC_128_OTFK  /* AES decryption with 'on the fly' 128 bit keying */
#endif
#if 0
#  define AES_ENC_256_OTFK  /* AES encryption with 'on the fly' 256 bit keying */
#endif
#if 0
#  define AES_DEC_256_OTFK  /* AES decryption with 'on the fly' 256 bit keying */
#endif

#define N_ROW                   4
#define N_COL                   4
#define N_BLOCK   (N_ROW * N_COL)
#define N_MAX_ROUNDS           14

typedef uint8_t return_type;

/*  Warning: The key length for 256 bit keys overflows a byte
    (see comment below)
*/

typedef uint8_t length_type;

typedef struct
{   uint8_t ksch[(N_MAX_ROUNDS + 1) * N_BLOCK];
    uint8_t rnd;
} aes_context;

/*  The following calls are for a precomputed key schedule
```

```
    NOTE: If the length_type used for the key length is an
    unsigned 8-bit character, a key length of 256 bits must
    be entered as a length in bytes (valid inputs are hence
    128, 192, 16, 24 and 32).
*/

#if defined( AES_ENC_PREKEYED ) || defined( AES_DEC_PREKEYED )

return_type aes_set_key( const uint8_t key[],

                         length_type keylen,
                         aes_context ctx[1] );
#endif

#if defined( AES_ENC_PREKEYED )

return_type aes_encrypt( const uint8_t in[N_BLOCK],
                         uint8_t out[N_BLOCK],
                         const aes_context ctx[1] );

return_type aes_cbc_encrypt( const uint8_t *in,
                         uint8_t *out,
                         int32_t n_block,
                         uint8_t iv[N_BLOCK],
                         const aes_context ctx[1] );
#endif

#if defined( AES_DEC_PREKEYED )

return_type aes_decrypt( const uint8_t in[N_BLOCK],
                         uint8_t out[N_BLOCK],
                         const aes_context ctx[1] );

return_type aes_cbc_decrypt( const uint8_t *in,
                         uint8_t *out,
                         int32_t n_block,
                         uint8_t iv[N_BLOCK],
                         const aes_context ctx[1] );
#endif

/*  The following calls are for 'on the fly' keying.  In this case the
    encryption and decryption keys are different.

    The encryption subroutines take a key in an array of bytes in
    key[L] where L is 16, 24 or 32 bytes for key lengths of 128,
    192, and 256 bits respectively.  They then encrypts the input
    data, in[] with this key and put the reult in the output array
    out[].  In addition, the second key array, o_key[L], is used
    to output the key that is needed by the decryption subroutine
    to reverse the encryption operation.  The two key arrays can
    be the same array but in this case the original key will be
    overwritten.

    In the same way, the decryption subroutines output keys that
    can be used to reverse their effect when used for encryption.

    Only 128 and 256 bit keys are supported in these 'on the fly'
    modes.
*/

#if defined( AES_ENC_128_OTFK )
void aes_encrypt_128( const uint8_t in[N_BLOCK],
                      uint8_t out[N_BLOCK],
                      const uint8_t key[N_BLOCK],
                      uint8_t o_key[N_BLOCK] );
#endif

#if defined( AES_DEC_128_OTFK )
void aes_decrypt_128( const uint8_t in[N_BLOCK],
                      uint8_t out[N_BLOCK],
                      const uint8_t key[N_BLOCK],
                      uint8_t o_key[N_BLOCK] );
#endif

#if defined( AES_ENC_256_OTFK )
```

```
void aes_encrypt_256( const uint8_t in[N_BLOCK],
                      uint8_t out[N_BLOCK],
                      const uint8_t key[2 * N_BLOCK],
                      uint8_t o_key[2 * N_BLOCK] );
#endif

#if defined( AES_DEC_256_OTFK )
void aes_decrypt_256( const uint8_t in[N_BLOCK],
                      uint8_t out[N_BLOCK],
                      const uint8_t key[2 * N_BLOCK],
                      uint8_t o_key[2 * N_BLOCK] );
#endif

#endif
```

A.2 aes.c

```
/*
 ---------------------------------------------------------------------------
 Copyright (c) 1998-2008, Brian Gladman, Worcester, UK. All rights reserved.

 LICENSE TERMS

 The redistribution and use of this software (with or without changes)
 is allowed without the payment of fees or royalties provided that:

  1. source code distributions include the above copyright notice, this
     list of conditions and the following disclaimer;

  2. binary distributions include the above copyright notice, this list
     of conditions and the following disclaimer in their documentation;

  3. the name of the copyright holder is not used to endorse products
     built using this software without specific written permission.

 DISCLAIMER

 This software is provided 'as is' with no explicit or implied warranties
 in respect of its properties, including, but not limited to, correctness
 and/or fitness for purpose.
 ---------------------------------------------------------------------------
 Issue 09/09/2006

 This is an AES implementation that uses only 8-bit byte operations on the
 cipher state (there are options to use 32-bit types if available).

 The combination of mix columns and byte substitution used here is based on
 that developed by Karl Malbrain. His contribution is acknowledged.
 */

/* define if you have a fast memcpy function on your system */
#if 0
#  define HAVE_MEMCPY
#  include <string.h>
#  if defined( _MSC_VER )
#    include <intrin.h>
#    pragma intrinsic( memcpy )
#  endif
#endif


#include <stdlib.h>
#include <stdint.h>

/* define if you have fast 32-bit types on your system */
#if 0
#  define HAVE_UINT_32T
#endif

/* define if you don't want any tables */
#if 1
#  define USE_TABLES
#endif
```

```
/*  On Intel Core 2 duo VERSION_1 is faster */

/* alternative versions (test for performance on your system) */
#if 1
#  define VERSION_1
#endif

#include "aes.h"

//#if defined( HAVE_UINT_32T )
//  typedef unsigned long uint32_t;
//#endif

/* functions for finite field multiplication in the AES Galois field    */

#define WPOLY   0x011b
#define BPOLY     0x1b
#define DPOLY   0x008d

#define f1(x)   (x)
#define f2(x)   ((x << 1) ^ (((x >> 7) & 1) * WPOLY))
#define f4(x)   ((x << 2) ^ (((x >> 6) & 1) * WPOLY) ^ (((x >> 6) & 2) * WPOLY))
#define f8(x)   ((x << 3) ^ (((x >> 5) & 1) * WPOLY) ^ (((x >> 5) & 2) * WPOLY) \
                          ^ (((x >> 5) & 4) * WPOLY))
#define d2(x)   (((x) >> 1) ^ ((x) & 1 ? DPOLY : 0))

#define f3(x)   (f2(x) ^ x)
#define f9(x)   (f8(x) ^ x)
#define fb(x)   (f8(x) ^ f2(x) ^ x)
#define fd(x)   (f8(x) ^ f4(x) ^ x)
#define fe(x)   (f8(x) ^ f4(x) ^ f2(x))

#if defined( USE_TABLES )

#define sb_data(w) {    /* S Box data values */                            \
    w(0x63), w(0x7c), w(0x77), w(0x7b), w(0xf2), w(0x6b), w(0x6f), w(0xc5),\
    w(0x30), w(0x01), w(0x67), w(0x2b), w(0xfe), w(0xd7), w(0xab), w(0x76),\
    w(0xca), w(0x82), w(0xc9), w(0x7d), w(0xfa), w(0x59), w(0x47), w(0xf0),\
    w(0xad), w(0xd4), w(0xa2), w(0xaf), w(0x9c), w(0xa4), w(0x72), w(0xc0),\
    w(0xb7), w(0xfd), w(0x93), w(0x26), w(0x36), w(0x3f), w(0xf7), w(0xcc),\
    w(0x34), w(0xa5), w(0xe5), w(0xf1), w(0x71), w(0xd8), w(0x31), w(0x15),\
    w(0x04), w(0xc7), w(0x23), w(0xc3), w(0x18), w(0x96), w(0x05), w(0x9a),\
    w(0x07), w(0x12), w(0x80), w(0xe2), w(0xeb), w(0x27), w(0xb2), w(0x75),\
    w(0x09), w(0x83), w(0x2c), w(0x1a), w(0x1b), w(0x6e), w(0x5a), w(0xa0),\
    w(0x52), w(0x3b), w(0xd6), w(0xb3), w(0x29), w(0xe3), w(0x2f), w(0x84),\
    w(0x53), w(0xd1), w(0x00), w(0xed), w(0x20), w(0xfc), w(0xb1), w(0x5b),\
    w(0x6a), w(0xcb), w(0xbe), w(0x39), w(0x4a), w(0x4c), w(0x58), w(0xcf),\
    w(0xd0), w(0xef), w(0xaa), w(0xfb), w(0x43), w(0x4d), w(0x33), w(0x85),\
    w(0x45), w(0xf9), w(0x02), w(0x7f), w(0x50), w(0x3c), w(0x9f), w(0xa8),\
    w(0x51), w(0xa3), w(0x40), w(0x8f), w(0x92), w(0x9d), w(0x38), w(0xf5),\
    w(0xbc), w(0xb6), w(0xda), w(0x21), w(0x10), w(0xff), w(0xf3), w(0xd2),\
    w(0xcd), w(0x0c), w(0x13), w(0xec), w(0x5f), w(0x97), w(0x44), w(0x17),\
    w(0xc4), w(0xa7), w(0x7e), w(0x3d), w(0x64), w(0x5d), w(0x19), w(0x73),\
    w(0x60), w(0x81), w(0x4f), w(0xdc), w(0x22), w(0x2a), w(0x90), w(0x88),\
    w(0x46), w(0xee), w(0xb8), w(0x14), w(0xde), w(0x5e), w(0x0b), w(0xdb),\
    w(0xe0), w(0x32), w(0x3a), w(0x0a), w(0x49), w(0x06), w(0x24), w(0x5c),\
    w(0xc2), w(0xd3), w(0xac), w(0x62), w(0x91), w(0x95), w(0xe4), w(0x79),\
    w(0xe7), w(0xc8), w(0x37), w(0x6d), w(0x8d), w(0xd5), w(0x4e), w(0xa9),\
    w(0x6c), w(0x56), w(0xf4), w(0xea), w(0x65), w(0x7a), w(0xae), w(0x08),\
    w(0xba), w(0x78), w(0x25), w(0x2e), w(0x1c), w(0xa6), w(0xb4), w(0xc6),\
    w(0xe8), w(0xdd), w(0x74), w(0x1f), w(0x4b), w(0xbd), w(0x8b), w(0x8a),\
    w(0x70), w(0x3e), w(0xb5), w(0x66), w(0x48), w(0x03), w(0xf6), w(0x0e),\
    w(0x61), w(0x35), w(0x57), w(0xb9), w(0x86), w(0xc1), w(0x1d), w(0x9e),\
    w(0xe1), w(0xf8), w(0x98), w(0x11), w(0x69), w(0xd9), w(0x8e), w(0x94),\
    w(0x9b), w(0x1e), w(0x87), w(0xe9), w(0xce), w(0x55), w(0x28), w(0xdf),\
    w(0x8c), w(0xa1), w(0x89), w(0x0d), w(0xbf), w(0xe6), w(0x42), w(0x68),\
    w(0x41), w(0x99), w(0x2d), w(0x0f), w(0xb0), w(0x54), w(0xbb), w(0x16) }

#define isb_data(w) {   /* inverse S Box data values */                    \
    w(0x52), w(0x09), w(0x6a), w(0xd5), w(0x30), w(0x36), w(0xa5), w(0x38),\
    w(0xbf), w(0x40), w(0xa3), w(0x9e), w(0x81), w(0xf3), w(0xd7), w(0xfb),\
    w(0x7c), w(0xe3), w(0x39), w(0x82), w(0x9b), w(0x2f), w(0xff), w(0x87),\
    w(0x34), w(0x8e), w(0x43), w(0x44), w(0xc4), w(0xde), w(0xe9), w(0xcb),\
    w(0x54), w(0x7b), w(0x94), w(0x32), w(0xa6), w(0xc2), w(0x23), w(0x3d),\
    w(0xee), w(0x4c), w(0x95), w(0x0b), w(0x42), w(0xfa), w(0xc3), w(0x4e),\
    w(0x08), w(0x2e), w(0xa1), w(0x66), w(0x28), w(0xd9), w(0x24), w(0xb2),\
```

```
    w(0x76), w(0x5b), w(0xa2), w(0x49), w(0x6d), w(0x8b), w(0xd1), w(0x25),\
    w(0x72), w(0xf8), w(0xf6), w(0x64), w(0x86), w(0x68), w(0x98), w(0x16),\
    w(0xd4), w(0xa4), w(0x5c), w(0xcc), w(0x5d), w(0x65), w(0xb6), w(0x92),\
    w(0x6c), w(0x70), w(0x48), w(0x50), w(0xfd), w(0xed), w(0xb9), w(0xda),\
    w(0x5e), w(0x15), w(0x46), w(0x57), w(0xa7), w(0x8d), w(0x9d), w(0x84),\
    w(0x90), w(0xd8), w(0xab), w(0x00), w(0x8c), w(0xbc), w(0xd3), w(0x0a),\
    w(0xf7), w(0xe4), w(0x58), w(0x05), w(0xb8), w(0xb3), w(0x45), w(0x06),\
    w(0xd0), w(0x2c), w(0x1e), w(0x8f), w(0xca), w(0x3f), w(0x0f), w(0x02),\
    w(0xc1), w(0xaf), w(0xbd), w(0x03), w(0x01), w(0x13), w(0x8a), w(0x6b),\
    w(0x3a), w(0x91), w(0x11), w(0x41), w(0x4f), w(0x67), w(0xdc), w(0xea),\
    w(0x97), w(0xf2), w(0xcf), w(0xce), w(0xf0), w(0xb4), w(0xe6), w(0x73),\
    w(0x96), w(0xac), w(0x74), w(0x22), w(0xe7), w(0xad), w(0x35), w(0x85),\
    w(0xe2), w(0xf9), w(0x37), w(0xe8), w(0x1c), w(0x75), w(0xdf), w(0x6e),\
    w(0x47), w(0xf1), w(0x1a), w(0x71), w(0x1d), w(0x29), w(0xc5), w(0x89),\
    w(0x6f), w(0xb7), w(0x62), w(0x0e), w(0xaa), w(0x18), w(0xbe), w(0x1b),\
    w(0xfc), w(0x56), w(0x3e), w(0x4b), w(0xc6), w(0xd2), w(0x79), w(0x20),\
    w(0x9a), w(0xdb), w(0xc0), w(0xfe), w(0x78), w(0xcd), w(0x5a), w(0xf4),\
    w(0x1f), w(0xdd), w(0xa8), w(0x33), w(0x88), w(0x07), w(0xc7), w(0x31),\
    w(0xb1), w(0x12), w(0x10), w(0x59), w(0x27), w(0x80), w(0xec), w(0x5f),\
    w(0x60), w(0x51), w(0x7f), w(0xa9), w(0x19), w(0xb5), w(0x4a), w(0x0d),\
    w(0x2d), w(0xe5), w(0x7a), w(0x9f), w(0x93), w(0xc9), w(0x9c), w(0xef),\
    w(0xa0), w(0xe0), w(0x3b), w(0x4d), w(0xae), w(0x2a), w(0xf5), w(0xb0),\
    w(0xc8), w(0xeb), w(0xbb), w(0x3c), w(0x83), w(0x53), w(0x99), w(0x61),\
    w(0x17), w(0x2b), w(0x04), w(0x7e), w(0xba), w(0x77), w(0xd6), w(0x26),\
    w(0xe1), w(0x69), w(0x14), w(0x63), w(0x55), w(0x21), w(0x0c), w(0x7d) }

#define mm_data(w) {    /* basic data for forming finite field tables */   \
    w(0x00), w(0x01), w(0x02), w(0x03), w(0x04), w(0x05), w(0x06), w(0x07),\
    w(0x08), w(0x09), w(0x0a), w(0x0b), w(0x0c), w(0x0d), w(0x0e), w(0x0f),\
    w(0x10), w(0x11), w(0x12), w(0x13), w(0x14), w(0x15), w(0x16), w(0x17),\
    w(0x18), w(0x19), w(0x1a), w(0x1b), w(0x1c), w(0x1d), w(0x1e), w(0x1f),\
    w(0x20), w(0x21), w(0x22), w(0x23), w(0x24), w(0x25), w(0x26), w(0x27),\
    w(0x28), w(0x29), w(0x2a), w(0x2b), w(0x2c), w(0x2d), w(0x2e), w(0x2f),\
    w(0x30), w(0x31), w(0x32), w(0x33), w(0x34), w(0x35), w(0x36), w(0x37),\
    w(0x38), w(0x39), w(0x3a), w(0x3b), w(0x3c), w(0x3d), w(0x3e), w(0x3f),\
    w(0x40), w(0x41), w(0x42), w(0x43), w(0x44), w(0x45), w(0x46), w(0x47),\
    w(0x48), w(0x49), w(0x4a), w(0x4b), w(0x4c), w(0x4d), w(0x4e), w(0x4f),\
    w(0x50), w(0x51), w(0x52), w(0x53), w(0x54), w(0x55), w(0x56), w(0x57),\
    w(0x58), w(0x59), w(0x5a), w(0x5b), w(0x5c), w(0x5d), w(0x5e), w(0x5f),\
    w(0x60), w(0x61), w(0x62), w(0x63), w(0x64), w(0x65), w(0x66), w(0x67),\
    w(0x68), w(0x69), w(0x6a), w(0x6b), w(0x6c), w(0x6d), w(0x6e), w(0x6f),\
    w(0x70), w(0x71), w(0x72), w(0x73), w(0x74), w(0x75), w(0x76), w(0x77),\
    w(0x78), w(0x79), w(0x7a), w(0x7b), w(0x7c), w(0x7d), w(0x7e), w(0x7f),\
    w(0x80), w(0x81), w(0x82), w(0x83), w(0x84), w(0x85), w(0x86), w(0x87),\
    w(0x88), w(0x89), w(0x8a), w(0x8b), w(0x8c), w(0x8d), w(0x8e), w(0x8f),\
    w(0x90), w(0x91), w(0x92), w(0x93), w(0x94), w(0x95), w(0x96), w(0x97),\
    w(0x98), w(0x99), w(0x9a), w(0x9b), w(0x9c), w(0x9d), w(0x9e), w(0x9f),\
    w(0xa0), w(0xa1), w(0xa2), w(0xa3), w(0xa4), w(0xa5), w(0xa6), w(0xa7),\
    w(0xa8), w(0xa9), w(0xaa), w(0xab), w(0xac), w(0xad), w(0xae), w(0xaf),\
    w(0xb0), w(0xb1), w(0xb2), w(0xb3), w(0xb4), w(0xb5), w(0xb6), w(0xb7),\
    w(0xb8), w(0xb9), w(0xba), w(0xbb), w(0xbc), w(0xbd), w(0xbe), w(0xbf),\
    w(0xc0), w(0xc1), w(0xc2), w(0xc3), w(0xc4), w(0xc5), w(0xc6), w(0xc7),\
    w(0xc8), w(0xc9), w(0xca), w(0xcb), w(0xcc), w(0xcd), w(0xce), w(0xcf),\
    w(0xd0), w(0xd1), w(0xd2), w(0xd3), w(0xd4), w(0xd5), w(0xd6), w(0xd7),\
    w(0xd8), w(0xd9), w(0xda), w(0xdb), w(0xdc), w(0xdd), w(0xde), w(0xdf),\
    w(0xe0), w(0xe1), w(0xe2), w(0xe3), w(0xe4), w(0xe5), w(0xe6), w(0xe7),\
    w(0xe8), w(0xe9), w(0xea), w(0xeb), w(0xec), w(0xed), w(0xee), w(0xef),\
    w(0xf0), w(0xf1), w(0xf2), w(0xf3), w(0xf4), w(0xf5), w(0xf6), w(0xf7),\
    w(0xf8), w(0xf9), w(0xfa), w(0xfb), w(0xfc), w(0xfd), w(0xfe), w(0xff) }

static const uint8_t sbox[256]  =  sb_data(f1);

#if defined( AES_DEC_PREKEYED )
static const uint8_t isbox[256] = isb_data(f1);
#endif

static const uint8_t gfm2_sbox[256] = sb_data(f2);
static const uint8_t gfm3_sbox[256] = sb_data(f3);

#if defined( AES_DEC_PREKEYED )
static const uint8_t gfmul_9[256] = mm_data(f9);
static const uint8_t gfmul_b[256] = mm_data(fb);
static const uint8_t gfmul_d[256] = mm_data(fd);
static const uint8_t gfmul_e[256] = mm_data(fe);
#endif
```

```
#define s_box(x)     sbox[(x)]
#if defined( AES_DEC_PREKEYED )
#define is_box(x)    isbox[(x)]
#endif
#define gfm2_sb(x)   gfm2_sbox[(x)]
#define gfm3_sb(x)   gfm3_sbox[(x)]
#if defined( AES_DEC_PREKEYED )
#define gfm_9(x)     gfmul_9[(x)]
#define gfm_b(x)     gfmul_b[(x)]
#define gfm_d(x)     gfmul_d[(x)]
#define gfm_e(x)     gfmul_e[(x)]
#endif
#else

/* this is the high bit of x right shifted by 1 */
/* position. Since the starting polynomial has  */
/* 9 bits (0x11b), this right shift keeps the   */
/* values of all top bits within a byte         */

static uint8_t hibit(const uint8_t x)
{   uint8_t r = (uint8_t)((x >> 1) | (x >> 2));

    r |= (r >> 2);
    r |= (r >> 4);
    return (r + 1) >> 1;
}

/* return the inverse of the finite field element x */

static uint8_t gf_inv(const uint8_t x)
{   uint8_t p1 = x, p2 = BPOLY, n1 = hibit(x), n2 = 0x80, v1 = 1, v2 = 0;

    if(x < 2)
        return x;

    for( ; ; )
    {
        if(n1)
            while(n2 >= n1)             /* divide polynomial p2 by p1    */
            {
                n2 /= n1;               /* shift smaller polynomial left */
                p2 ^= (p1 * n2) & 0xff; /* and remove from larger one    */
                v2 ^= (v1 * n2);        /* shift accumulated value and   */
                n2 = hibit(p2);         /* add into result               */
            }
        else
            return v1;

        if(n2)                          /* repeat with values swapped    */
            while(n1 >= n2)
            {
                n1 /= n2;
                p1 ^= p2 * n1;
                v1 ^= v2 * n1;
                n1 = hibit(p1);
            }
        else
            return v2;
    }
}

/* The forward and inverse affine transformations used in the S-box */
uint8_t fwd_affine(const uint8_t x)
{
#if defined( HAVE_UINT_32T )
    uint32_t w = x;
    w ^= (w << 1) ^ (w << 2) ^ (w << 3) ^ (w << 4);
    return 0x63 ^ ((w ^ (w >> 8)) & 0xff);
#else
    return 0x63 ^ x ^ (x << 1) ^ (x << 2) ^ (x << 3) ^ (x << 4)
                    ^ (x >> 7) ^ (x >> 6) ^ (x >> 5) ^ (x >> 4);
#endif
}

uint8_t inv_affine(const uint8_t x)
{
```

```
#if defined( HAVE_UINT_32T )
    uint32_t w = x;
    w = (w << 1) ^ (w << 3) ^ (w << 6);
    return 0x05 ^ ((w ^ (w >> 8)) & 0xff);
#else
    return 0x05 ^ (x << 1) ^ (x << 3) ^ (x << 6)
                ^ (x >> 7) ^ (x >> 5) ^ (x >> 2);
#endif
}

#define s_box(x)   fwd_affine(gf_inv(x))
#define is_box(x)  gf_inv(inv_affine(x))
#define gfm2_sb(x) f2(s_box(x))
#define gfm3_sb(x) f3(s_box(x))
#define gfm_9(x)   f9(x)
#define gfm_b(x)   fb(x)
#define gfm_d(x)   fd(x)
#define gfm_e(x)   fe(x)

#endif

#if defined( HAVE_MEMCPY )
#  define block_copy_nn(d, s, l)    memcpy(d, s, l)
#  define block_copy(d, s)          memcpy(d, s, N_BLOCK)
#else
#  define block_copy_nn(d, s, l)    copy_block_nn(d, s, l)
#  define block_copy(d, s)          copy_block(d, s)
#endif

static void copy_block( void *d, const void *s )
{
#if defined( HAVE_UINT_32T )
    ((uint32_t*)d)[ 0] = ((uint32_t*)s)[ 0];
    ((uint32_t*)d)[ 1] = ((uint32_t*)s)[ 1];
    ((uint32_t*)d)[ 2] = ((uint32_t*)s)[ 2];
    ((uint32_t*)d)[ 3] = ((uint32_t*)s)[ 3];
#else
    ((uint8_t*)d)[ 0] = ((uint8_t*)s)[ 0];
    ((uint8_t*)d)[ 1] = ((uint8_t*)s)[ 1];
    ((uint8_t*)d)[ 2] = ((uint8_t*)s)[ 2];
    ((uint8_t*)d)[ 3] = ((uint8_t*)s)[ 3];
    ((uint8_t*)d)[ 4] = ((uint8_t*)s)[ 4];
    ((uint8_t*)d)[ 5] = ((uint8_t*)s)[ 5];
    ((uint8_t*)d)[ 6] = ((uint8_t*)s)[ 6];
    ((uint8_t*)d)[ 7] = ((uint8_t*)s)[ 7];
    ((uint8_t*)d)[ 8] = ((uint8_t*)s)[ 8];
    ((uint8_t*)d)[ 9] = ((uint8_t*)s)[ 9];
    ((uint8_t*)d)[10] = ((uint8_t*)s)[10];
    ((uint8_t*)d)[11] = ((uint8_t*)s)[11];
    ((uint8_t*)d)[12] = ((uint8_t*)s)[12];
    ((uint8_t*)d)[13] = ((uint8_t*)s)[13];
    ((uint8_t*)d)[14] = ((uint8_t*)s)[14];
    ((uint8_t*)d)[15] = ((uint8_t*)s)[15];
#endif
}

static void copy_block_nn( uint8_t * d, const uint8_t *s, uint8_t nn )
{
    while( nn-- )
        //*((uint8_t*)d)++ = *((uint8_t*)s)++;
        *d++ = *s++;
}

static void xor_block( void *d, const void *s )
{
#if defined( HAVE_UINT_32T )
    ((uint32_t*)d)[ 0] ^= ((uint32_t*)s)[ 0];
    ((uint32_t*)d)[ 1] ^= ((uint32_t*)s)[ 1];
    ((uint32_t*)d)[ 2] ^= ((uint32_t*)s)[ 2];
    ((uint32_t*)d)[ 3] ^= ((uint32_t*)s)[ 3];
#else
    ((uint8_t*)d)[ 0] ^= ((uint8_t*)s)[ 0];
    ((uint8_t*)d)[ 1] ^= ((uint8_t*)s)[ 1];
    ((uint8_t*)d)[ 2] ^= ((uint8_t*)s)[ 2];
    ((uint8_t*)d)[ 3] ^= ((uint8_t*)s)[ 3];
    ((uint8_t*)d)[ 4] ^= ((uint8_t*)s)[ 4];
```

```
    ((uint8_t*)d)[ 5] ^= ((uint8_t*)s)[ 5];
    ((uint8_t*)d)[ 6] ^= ((uint8_t*)s)[ 6];
    ((uint8_t*)d)[ 7] ^= ((uint8_t*)s)[ 7];
    ((uint8_t*)d)[ 8] ^= ((uint8_t*)s)[ 8];
    ((uint8_t*)d)[ 9] ^= ((uint8_t*)s)[ 9];
    ((uint8_t*)d)[10] ^= ((uint8_t*)s)[10];
    ((uint8_t*)d)[11] ^= ((uint8_t*)s)[11];
    ((uint8_t*)d)[12] ^= ((uint8_t*)s)[12];
    ((uint8_t*)d)[13] ^= ((uint8_t*)s)[13];
    ((uint8_t*)d)[14] ^= ((uint8_t*)s)[14];
    ((uint8_t*)d)[15] ^= ((uint8_t*)s)[15];
#endif
}

static void copy_and_key( void *d, const void *s, const void *k )
{
#if defined( HAVE_UINT_32T )
    ((uint32_t*)d)[ 0] = ((uint32_t*)s)[ 0] ^ ((uint32_t*)k)[ 0];
    ((uint32_t*)d)[ 1] = ((uint32_t*)s)[ 1] ^ ((uint32_t*)k)[ 1];
    ((uint32_t*)d)[ 2] = ((uint32_t*)s)[ 2] ^ ((uint32_t*)k)[ 2];
    ((uint32_t*)d)[ 3] = ((uint32_t*)s)[ 3] ^ ((uint32_t*)k)[ 3];
#elif 1
    ((uint8_t*)d)[ 0] = ((uint8_t*)s)[ 0] ^ ((uint8_t*)k)[ 0];
    ((uint8_t*)d)[ 1] = ((uint8_t*)s)[ 1] ^ ((uint8_t*)k)[ 1];
    ((uint8_t*)d)[ 2] = ((uint8_t*)s)[ 2] ^ ((uint8_t*)k)[ 2];
    ((uint8_t*)d)[ 3] = ((uint8_t*)s)[ 3] ^ ((uint8_t*)k)[ 3];
    ((uint8_t*)d)[ 4] = ((uint8_t*)s)[ 4] ^ ((uint8_t*)k)[ 4];
    ((uint8_t*)d)[ 5] = ((uint8_t*)s)[ 5] ^ ((uint8_t*)k)[ 5];
    ((uint8_t*)d)[ 6] = ((uint8_t*)s)[ 6] ^ ((uint8_t*)k)[ 6];
    ((uint8_t*)d)[ 7] = ((uint8_t*)s)[ 7] ^ ((uint8_t*)k)[ 7];
    ((uint8_t*)d)[ 8] = ((uint8_t*)s)[ 8] ^ ((uint8_t*)k)[ 8];
    ((uint8_t*)d)[ 9] = ((uint8_t*)s)[ 9] ^ ((uint8_t*)k)[ 9];
    ((uint8_t*)d)[10] = ((uint8_t*)s)[10] ^ ((uint8_t*)k)[10];
    ((uint8_t*)d)[11] = ((uint8_t*)s)[11] ^ ((uint8_t*)k)[11];
    ((uint8_t*)d)[12] = ((uint8_t*)s)[12] ^ ((uint8_t*)k)[12];
    ((uint8_t*)d)[13] = ((uint8_t*)s)[13] ^ ((uint8_t*)k)[13];
    ((uint8_t*)d)[14] = ((uint8_t*)s)[14] ^ ((uint8_t*)k)[14];
    ((uint8_t*)d)[15] = ((uint8_t*)s)[15] ^ ((uint8_t*)k)[15];
#else
    block_copy(d, s);
    xor_block(d, k);
#endif
}

static void add_round_key( uint8_t d[N_BLOCK], const uint8_t k[N_BLOCK] )
{
    xor_block(d, k);
}

static void shift_sub_rows( uint8_t st[N_BLOCK] )
{   uint8_t tt;

    st[ 0] = s_box(st[ 0]); st[ 4] = s_box(st[ 4]);
    st[ 8] = s_box(st[ 8]); st[12] = s_box(st[12]);

    tt = st[1]; st[ 1] = s_box(st[ 5]); st[ 5] = s_box(st[ 9]);
    st[ 9] = s_box(st[13]); st[13] = s_box( tt );

    tt = st[2]; st[ 2] = s_box(st[10]); st[10] = s_box( tt );
    tt = st[6]; st[ 6] = s_box(st[14]); st[14] = s_box( tt );

    tt = st[15]; st[15] = s_box(st[11]); st[11] = s_box(st[ 7]);
    st[ 7] = s_box(st[ 3]); st[ 3] = s_box( tt );
}

#if defined( AES_DEC_PREKEYED )

static void inv_shift_sub_rows( uint8_t st[N_BLOCK] )
{   uint8_t tt;

    st[ 0] = is_box(st[ 0]); st[ 4] = is_box(st[ 4]);
    st[ 8] = is_box(st[ 8]); st[12] = is_box(st[12]);

    tt = st[13]; st[13] = is_box(st[9]); st[ 9] = is_box(st[5]);
    st[ 5] = is_box(st[1]); st[ 1] = is_box( tt );
```

```
    tt = st[2]; st[ 2] = is_box(st[10]); st[10] = is_box( tt );
    tt = st[6]; st[ 6] = is_box(st[14]); st[14] = is_box( tt );

    tt = st[3]; st[ 3] = is_box(st[ 7]); st[ 7] = is_box(st[11]);
    st[11] = is_box(st[15]); st[15] = is_box( tt );
}

#endif

#if defined( VERSION_1 )
  static void mix_sub_columns( uint8_t dt[N_BLOCK] )
  { uint8_t st[N_BLOCK];
    block_copy(st, dt);
#else
  static void mix_sub_columns( uint8_t dt[N_BLOCK], uint8_t st[N_BLOCK] )
  {
#endif
    dt[ 0] = gfm2_sb(st[0]) ^ gfm3_sb(st[5]) ^ s_box(st[10]) ^ s_box(st[15]);
    dt[ 1] = s_box(st[0]) ^ gfm2_sb(st[5]) ^ gfm3_sb(st[10]) ^ s_box(st[15]);
    dt[ 2] = s_box(st[0]) ^ s_box(st[5]) ^ gfm2_sb(st[10]) ^ gfm3_sb(st[15]);
    dt[ 3] = gfm3_sb(st[0]) ^ s_box(st[5]) ^ s_box(st[10]) ^ gfm2_sb(st[15]);

    dt[ 4] = gfm2_sb(st[4]) ^ gfm3_sb(st[9]) ^ s_box(st[14]) ^ s_box(st[3]);
    dt[ 5] = s_box(st[4]) ^ gfm2_sb(st[9]) ^ gfm3_sb(st[14]) ^ s_box(st[3]);
    dt[ 6] = s_box(st[4]) ^ s_box(st[9]) ^ gfm2_sb(st[14]) ^ gfm3_sb(st[3]);
    dt[ 7] = gfm3_sb(st[4]) ^ s_box(st[9]) ^ s_box(st[14]) ^ gfm2_sb(st[3]);

    dt[ 8] = gfm2_sb(st[8]) ^ gfm3_sb(st[13]) ^ s_box(st[2]) ^ s_box(st[7]);
    dt[ 9] = s_box(st[8]) ^ gfm2_sb(st[13]) ^ gfm3_sb(st[2]) ^ s_box(st[7]);
    dt[10] = s_box(st[8]) ^ s_box(st[13]) ^ gfm2_sb(st[2]) ^ gfm3_sb(st[7]);
    dt[11] = gfm3_sb(st[8]) ^ s_box(st[13]) ^ s_box(st[2]) ^ gfm2_sb(st[7]);

    dt[12] = gfm2_sb(st[12]) ^ gfm3_sb(st[1]) ^ s_box(st[6]) ^ s_box(st[11]);
    dt[13] = s_box(st[12]) ^ gfm2_sb(st[1]) ^ gfm3_sb(st[6]) ^ s_box(st[11]);
    dt[14] = s_box(st[12]) ^ s_box(st[1]) ^ gfm2_sb(st[6]) ^ gfm3_sb(st[11]);
    dt[15] = gfm3_sb(st[12]) ^ s_box(st[1]) ^ s_box(st[6]) ^ gfm2_sb(st[11]);
  }

#if defined( AES_DEC_PREKEYED )

#if defined( VERSION_1 )
  static void inv_mix_sub_columns( uint8_t dt[N_BLOCK] )
  { uint8_t st[N_BLOCK];
    block_copy(st, dt);
#else
  static void inv_mix_sub_columns( uint8_t dt[N_BLOCK], uint8_t st[N_BLOCK] )
  {
#endif
    dt[ 0] = is_box(gfm_e(st[ 0]) ^ gfm_b(st[ 1]) ^ gfm_d(st[ 2]) ^ gfm_9(st[ 3]));
    dt[ 5] = is_box(gfm_9(st[ 0]) ^ gfm_e(st[ 1]) ^ gfm_b(st[ 2]) ^ gfm_d(st[ 3]));
    dt[10] = is_box(gfm_d(st[ 0]) ^ gfm_9(st[ 1]) ^ gfm_e(st[ 2]) ^ gfm_b(st[ 3]));
    dt[15] = is_box(gfm_b(st[ 0]) ^ gfm_d(st[ 1]) ^ gfm_9(st[ 2]) ^ gfm_e(st[ 3]));

    dt[ 4] = is_box(gfm_e(st[ 4]) ^ gfm_b(st[ 5]) ^ gfm_d(st[ 6]) ^ gfm_9(st[ 7]));
    dt[ 9] = is_box(gfm_9(st[ 4]) ^ gfm_e(st[ 5]) ^ gfm_b(st[ 6]) ^ gfm_d(st[ 7]));
    dt[14] = is_box(gfm_d(st[ 4]) ^ gfm_9(st[ 5]) ^ gfm_e(st[ 6]) ^ gfm_b(st[ 7]));
    dt[ 3] = is_box(gfm_b(st[ 4]) ^ gfm_d(st[ 5]) ^ gfm_9(st[ 6]) ^ gfm_e(st[ 7]));

    dt[ 8] = is_box(gfm_e(st[ 8]) ^ gfm_b(st[ 9]) ^ gfm_d(st[10]) ^ gfm_9(st[11]));
    dt[13] = is_box(gfm_9(st[ 8]) ^ gfm_e(st[ 9]) ^ gfm_b(st[10]) ^ gfm_d(st[11]));
    dt[ 2] = is_box(gfm_d(st[ 8]) ^ gfm_9(st[ 9]) ^ gfm_e(st[10]) ^ gfm_b(st[11]));
    dt[ 7] = is_box(gfm_b(st[ 8]) ^ gfm_d(st[ 9]) ^ gfm_9(st[10]) ^ gfm_e(st[11]));

    dt[12] = is_box(gfm_e(st[12]) ^ gfm_b(st[13]) ^ gfm_d(st[14]) ^ gfm_9(st[15]));
    dt[ 1] = is_box(gfm_9(st[12]) ^ gfm_e(st[13]) ^ gfm_b(st[14]) ^ gfm_d(st[15]));
    dt[ 6] = is_box(gfm_d(st[12]) ^ gfm_9(st[13]) ^ gfm_e(st[14]) ^ gfm_b(st[15]));
    dt[11] = is_box(gfm_b(st[12]) ^ gfm_d(st[13]) ^ gfm_9(st[14]) ^ gfm_e(st[15]));
  }

#endif

#if defined( AES_ENC_PREKEYED ) || defined( AES_DEC_PREKEYED )

/*  Set the cipher key for the pre-keyed version */

return_type aes_set_key( const uint8_t key[], length_type keylen, aes_context ctx[1] )
```

```
{
    uint8_t cc, rc, hi;

    switch( keylen )
    {
    case 16:
    case 24:
    case 32:
        break;
    default:
        ctx->rnd = 0;
        return ( uint8_t )-1;
    }
    block_copy_nn(ctx->ksch, key, keylen);
    hi = (keylen + 28) << 2;
    ctx->rnd = (hi >> 4) - 1;
    for( cc = keylen, rc = 1; cc < hi; cc += 4 )
    {   uint8_t tt, t0, t1, t2, t3;

        t0 = ctx->ksch[cc - 4];
        t1 = ctx->ksch[cc - 3];
        t2 = ctx->ksch[cc - 2];
        t3 = ctx->ksch[cc - 1];
        if( cc % keylen == 0 )
        {
            tt = t0;
            t0 = s_box(t1) ^ rc;
            t1 = s_box(t2);
            t2 = s_box(t3);
            t3 = s_box(tt);
            rc = f2(rc);
        }
        else if( keylen > 24 && cc % keylen == 16 )
        {
            t0 = s_box(t0);
            t1 = s_box(t1);
            t2 = s_box(t2);
            t3 = s_box(t3);
        }
        tt = cc - keylen;
        ctx->ksch[cc + 0] = ctx->ksch[tt + 0] ^ t0;
        ctx->ksch[cc + 1] = ctx->ksch[tt + 1] ^ t1;
        ctx->ksch[cc + 2] = ctx->ksch[tt + 2] ^ t2;
        ctx->ksch[cc + 3] = ctx->ksch[tt + 3] ^ t3;
    }
    return 0;
}

#endif

#if defined( AES_ENC_PREKEYED )

/*  Encrypt a single block of 16 bytes */

return_type aes_encrypt( const uint8_t in[N_BLOCK], uint8_t  out[N_BLOCK], const aes_context
ctx[1] )
{
    if( ctx->rnd )
    {
        uint8_t s1[N_BLOCK], r;
        copy_and_key( s1, in, ctx->ksch );

        for( r = 1 ; r < ctx->rnd ; ++r )
#if defined( VERSION_1 )
        {
            mix_sub_columns( s1 );
            add_round_key( s1, ctx->ksch + r * N_BLOCK);
        }
#else
        {   uint8_t s2[N_BLOCK];
            mix_sub_columns( s2, s1 );
            copy_and_key( s1, s2, ctx->ksch + r * N_BLOCK);
        }
#endif
        shift_sub_rows( s1 );
        copy_and_key( out, s1, ctx->ksch + r * N_BLOCK );
```

```
    }
    else
        return ( uint8_t )-1;
    return 0;
}

/* CBC encrypt a number of blocks (input and return an IV) */

return_type aes_cbc_encrypt( const uint8_t *in, uint8_t *out,
                        int32_t n_block, uint8_t iv[N_BLOCK], const aes_context ctx[1] )
{

    while(n_block--)
    {
        xor_block(iv, in);
        if(aes_encrypt(iv, iv, ctx) != EXIT_SUCCESS)
            return EXIT_FAILURE;
        //memcpy(out, iv, N_BLOCK);
        block_copy(out, iv);
        in += N_BLOCK;
        out += N_BLOCK;
    }
    return EXIT_SUCCESS;
}

#endif

#if defined( AES_DEC_PREKEYED )

/*  Decrypt a single block of 16 bytes */

return_type aes_decrypt( const uint8_t in[N_BLOCK], uint8_t out[N_BLOCK], const aes_context ctx[1] )
{
    if( ctx->rnd )
    {
        uint8_t s1[N_BLOCK], r;
        copy_and_key( s1, in, ctx->ksch + ctx->rnd * N_BLOCK );
        inv_shift_sub_rows( s1 );

        for( r = ctx->rnd ; --r ; )
#if defined( VERSION_1 )
        {
            add_round_key( s1, ctx->ksch + r * N_BLOCK );
            inv_mix_sub_columns( s1 );
        }
#else
        {   uint8_t s2[N_BLOCK];
            copy_and_key( s2, s1, ctx->ksch + r * N_BLOCK );
            inv_mix_sub_columns( s1, s2 );
        }
#endif
        copy_and_key( out, s1, ctx->ksch );
    }
    else
        return -1;
    return 0;
}

/* CBC decrypt a number of blocks (input and return an IV) */

return_type aes_cbc_decrypt( const uint8_t *in, uint8_t *out,
                        int32_t n_block, uint8_t iv[N_BLOCK], const aes_context ctx[1] )
{
    while(n_block--)
    {   uint8_t tmp[N_BLOCK];

        //memcpy(tmp, in, N_BLOCK);
        block_copy(tmp, in);
        if(aes_decrypt(in, out, ctx) != EXIT_SUCCESS)
            return EXIT_FAILURE;
        xor_block(out, iv);
        //memcpy(iv, tmp, N_BLOCK);
        block_copy(iv, tmp);
        in += N_BLOCK;
        out += N_BLOCK;
    }
```

```
    return EXIT_SUCCESS;
}

#endif

#if defined( AES_ENC_128_OTFK )

/*  The 'on the fly' encryption key update for for 128 bit keys */

static void update_encrypt_key_128( uint8_t k[N_BLOCK], uint8_t *rc )
{   uint8_t cc;

    k[0] ^= s_box(k[13]) ^ *rc;
    k[1] ^= s_box(k[14]);
    k[2] ^= s_box(k[15]);
    k[3] ^= s_box(k[12]);
    *rc = f2( *rc );

    for(cc = 4; cc < 16; cc += 4 )
    {
        k[cc + 0] ^= k[cc - 4];
        k[cc + 1] ^= k[cc - 3];
        k[cc + 2] ^= k[cc - 2];
        k[cc + 3] ^= k[cc - 1];
    }
}

/*  Encrypt a single block of 16 bytes with 'on the fly' 128 bit keying */

void aes_encrypt_128( const uint8_t in[N_BLOCK], uint8_t out[N_BLOCK],
                      const uint8_t key[N_BLOCK], uint8_t o_key[N_BLOCK] )
{   uint8_t s1[N_BLOCK], r, rc = 1;

    if(o_key != key)
        block_copy( o_key, key );
    copy_and_key( s1, in, o_key );

    for( r = 1 ; r < 10 ; ++r )
#if defined( VERSION_1 )
    {
        mix_sub_columns( s1 );
        update_encrypt_key_128( o_key, &rc );
        add_round_key( s1, o_key );
    }
#else
    {   uint8_t s2[N_BLOCK];
        mix_sub_columns( s2, s1 );
        update_encrypt_key_128( o_key, &rc );
        copy_and_key( s1, s2, o_key );
    }
#endif

    shift_sub_rows( s1 );
    update_encrypt_key_128( o_key, &rc );
    copy_and_key( out, s1, o_key );
}

#endif

#if defined( AES_DEC_128_OTFK )

/*  The 'on the fly' decryption key update for for 128 bit keys */

static void update_decrypt_key_128( uint8_t k[N_BLOCK], uint8_t *rc )
{   uint8_t cc;

    for( cc = 12; cc > 0; cc -= 4 )
    {
        k[cc + 0] ^= k[cc - 4];
        k[cc + 1] ^= k[cc - 3];
        k[cc + 2] ^= k[cc - 2];
        k[cc + 3] ^= k[cc - 1];
    }
    *rc = d2(*rc);
    k[0] ^= s_box(k[13]) ^ *rc;
    k[1] ^= s_box(k[14]);
```

```
    k[2] ^= s_box(k[15]);
    k[3] ^= s_box(k[12]);
}

/*  Decrypt a single block of 16 bytes with 'on the fly' 128 bit keying */

void aes_decrypt_128( const uint8_t in[N_BLOCK], uint8_t out[N_BLOCK],
                      const uint8_t key[N_BLOCK], uint8_t o_key[N_BLOCK] )
{
    uint8_t s1[N_BLOCK], r, rc = 0x6c;
    if(o_key != key)
        block_copy( o_key, key );

    copy_and_key( s1, in, o_key );
    inv_shift_sub_rows( s1 );

    for( r = 10 ; --r ; )
#if defined( VERSION_1 )
    {
        update_decrypt_key_128( o_key, &rc );
        add_round_key( s1, o_key );
        inv_mix_sub_columns( s1 );
    }
#else
    {   uint8_t s2[N_BLOCK];
        update_decrypt_key_128( o_key, &rc );
        copy_and_key( s2, s1, o_key );
        inv_mix_sub_columns( s1, s2 );
    }
#endif
    update_decrypt_key_128( o_key, &rc );
    copy_and_key( out, s1, o_key );
}

#endif

#if defined( AES_ENC_256_OTFK )

/*  The 'on the fly' encryption key update for for 256 bit keys */

static void update_encrypt_key_256( uint8_t k[2 * N_BLOCK], uint8_t *rc )
{   uint8_t cc;

    k[0] ^= s_box(k[29]) ^ *rc;
    k[1] ^= s_box(k[30]);
    k[2] ^= s_box(k[31]);
    k[3] ^= s_box(k[28]);
    *rc = f2( *rc );

    for(cc = 4; cc < 16; cc += 4)
    {
        k[cc + 0] ^= k[cc - 4];
        k[cc + 1] ^= k[cc - 3];
        k[cc + 2] ^= k[cc - 2];
        k[cc + 3] ^= k[cc - 1];
    }

    k[16] ^= s_box(k[12]);
    k[17] ^= s_box(k[13]);
    k[18] ^= s_box(k[14]);
    k[19] ^= s_box(k[15]);

    for( cc = 20; cc < 32; cc += 4 )
    {
        k[cc + 0] ^= k[cc - 4];
        k[cc + 1] ^= k[cc - 3];
        k[cc + 2] ^= k[cc - 2];
        k[cc + 3] ^= k[cc - 1];
    }
}

/*  Encrypt a single block of 16 bytes with 'on the fly' 256 bit keying */

void aes_encrypt_256( const uint8_t in[N_BLOCK], uint8_t out[N_BLOCK],
                      const uint8_t key[2 * N_BLOCK], uint8_t o_key[2 * N_BLOCK] )
{
```

```
    uint8_t s1[N_BLOCK], r, rc = 1;
    if(o_key != key)
    {
        block_copy( o_key, key );
        block_copy( o_key + 16, key + 16 );
    }
    copy_and_key( s1, in, o_key );

    for( r = 1 ; r < 14 ; ++r )
#if defined( VERSION_1 )
    {
        mix_sub_columns(s1);
        if( r & 1 )
            add_round_key( s1, o_key + 16 );
        else
        {
            update_encrypt_key_256( o_key, &rc );
            add_round_key( s1, o_key );
        }
    }
#else
    {   uint8_t s2[N_BLOCK];
        mix_sub_columns( s2, s1 );
        if( r & 1 )
            copy_and_key( s1, s2, o_key + 16 );
        else
        {
            update_encrypt_key_256( o_key, &rc );
            copy_and_key( s1, s2, o_key );
        }
    }
#endif

    shift_sub_rows( s1 );
    update_encrypt_key_256( o_key, &rc );
    copy_and_key( out, s1, o_key );
}

#endif

#if defined( AES_DEC_256_OTFK )

/*  The 'on the fly' encryption key update for for 256 bit keys */

static void update_decrypt_key_256( uint8_t k[2 * N_BLOCK], uint8_t *rc )
{   uint8_t cc;

    for(cc = 28; cc > 16; cc -= 4)
    {
        k[cc + 0] ^= k[cc - 4];
        k[cc + 1] ^= k[cc - 3];
        k[cc + 2] ^= k[cc - 2];
        k[cc + 3] ^= k[cc - 1];
    }

    k[16] ^= s_box(k[12]);
    k[17] ^= s_box(k[13]);
    k[18] ^= s_box(k[14]);
    k[19] ^= s_box(k[15]);

    for(cc = 12; cc > 0; cc -= 4)
    {
        k[cc + 0] ^= k[cc - 4];
        k[cc + 1] ^= k[cc - 3];
        k[cc + 2] ^= k[cc - 2];
        k[cc + 3] ^= k[cc - 1];
    }

    *rc = d2(*rc);
    k[0] ^= s_box(k[29]) ^ *rc;
    k[1] ^= s_box(k[30]);
    k[2] ^= s_box(k[31]);
    k[3] ^= s_box(k[28]);
}

/*  Decrypt a single block of 16 bytes with 'on the fly'
```

```
    256 bit keying
*/
void aes_decrypt_256( const uint8_t in[N_BLOCK], uint8_t out[N_BLOCK],
                      const uint8_t key[2 * N_BLOCK], uint8_t o_key[2 * N_BLOCK] )
{
    uint8_t s1[N_BLOCK], r, rc = 0x80;

    if(o_key != key)
    {
        block_copy( o_key, key );
        block_copy( o_key + 16, key + 16 );
    }

    copy_and_key( s1, in, o_key );
    inv_shift_sub_rows( s1 );

    for( r = 14 ; --r ; )
#if defined( VERSION_1 )
    {
        if( ( r & 1 ) )
        {
            update_decrypt_key_256( o_key, &rc );
            add_round_key( s1, o_key + 16 );
        }
        else
            add_round_key( s1, o_key );
        inv_mix_sub_columns( s1 );
    }
#else
    {   uint8_t s2[N_BLOCK];
        if( ( r & 1 ) )
        {
            update_decrypt_key_256( o_key, &rc );
            copy_and_key( s2, s1, o_key + 16 );
        }
        else
            copy_and_key( s2, s1, o_key );
        inv_mix_sub_columns( s1, s2 );
    }
#endif
    copy_and_key( out, s1, o_key );
}

#endif
```

Table of Contents