A1 – Annexe 1 : Fichiers à insérer dans les applications LoRa-TS et LoRa-MQTT

Dans cet annexe nous incluons les 2 fichiers à insérer dans les codes applicatifs basés sur les protocoles **LoRa-TS** et **LoRa-MQTT**.

A1.1 Couche liaison avec `Lora_Para.h` :

```
#define SS       5      //  NSS
#define RST     15      //  RST
#define DI0     26      //  INTR
#define SCK     18      //  CLK
#define MISO    19      //  MISO
#define MOSI    23      //  MOSI
#define BAND    868E6   //  frequency
#define SF      7       //  spreading factor
#define SBW     125E3   //  signal bandwidth
#define SW      0xF3    //  Sync Word
#define BR      8       //  bit rate (4/5,5/8)

void set_LoRa()  // all default settings
void set_LoRa_Freq(unsigned long freq)  // set frequency, all other default
void set_LoRa_All(int sck,int miso,int mosi,int ss,int rst, int dio0,unsigned long freq,unsigned
sbw, int sf, uint8_t sw)  // all settings
void set_LoRa_Pins(int sck,int miso,int mosi,int ss,int rst, int dio0)  // pins  settings, radio
default
void set_LoRa_Radio(unsigned long freq,unsigned sbw, int sf, uint8_t sw)  // radio settings, pins
default
```

et

```
void LoRa_rxMode()
void LoRa_txMode()
void onTxDone()
```

différents pour les terminaux et les passerelles

```
boolean runEvery(unsigned long interval)
```

Voici le code complet :

```
#define SS       5 // 26     // D0 - to NSS
#define RST     15  //16     // D4  - RST
#define DI0     26      // D8 - INTR
#define SCK     18      // D5 - CLK
#define MISO    19      // D6 - MISO
#define MOSI    23      // D7 - MOSI
#define BAND    868E6   // set frequency
#define SF      7       // set spreading factor
#define SBW     125E3   // set signal bandwidth
#define SW      0xF3    // set Sync Word
#define BR      8       // set bit rate (4/5,5/8)


void set_LoRa()  // all default settings
{
char buff[128];
  SPI.begin(SCK, MISO, MOSI, SS);  // pins and radio default
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  sprintf(buff,"BAND=%f,SF=%d,SBW=%f,SW=%X,BR=%d\n",BAND,SF,SBW,SW,BR);
  Serial.println(buff);
  LoRa.setSpreadingFactor(SF);
  LoRa.setSignalBandwidth(SBW);
  LoRa.setSyncWord(SW);
  }
```

```
void set_LoRa_Freq(unsigned long freq)  // set frequency, all other default settings
{
char buff[128];
  SPI.begin(SCK, MISO, MOSI, SS);  // pins and radio default
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  sprintf(buff,"BAND=%f,SF=%d,SBW=%f,SW=%X,BR=%d\n",freq,SF,SBW,SW,BR);
  Serial.println(buff);
  LoRa.setSpreadingFactor(SF);
  LoRa.setSignalBandwidth(SBW);
  LoRa.setSyncWord(SW);
  }

void set_LoRa_Freq_SF(unsigned long freq, int sf)  // set frequency, all other default settings
{
char buff[128];
  SPI.begin(SCK, MISO, MOSI, SS);  // pins and radio default
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  sprintf(buff,"BAND=%f,SF=%d,SBW=%f,SW=%X,BR=%d\n",freq,sf,SBW,SW,BR);
  Serial.println(buff);
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(SBW);
  LoRa.setSyncWord(SW);
  }

void set_LoRa_All(int sck,int miso,int mosi,int ss,int rst, int dio0,unsigned long freq,unsigned
sbw, int sf, uint8_t sw)  // all settings
{
  SPI.begin(sck, miso, mosi, ss);  // SCK, MISO, MOSI, SS
  LoRa.setPins(ss, rst, dio0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
  }

void set_LoRa_Pins(int sck,int miso,int mosi,int ss,int rst, int dio0)  // pins  settings, radio
default
{
  SPI.begin(sck, miso, mosi, ss);  // SCK, MISO, MOSI, SS
  LoRa.setPins(ss, rst, dio0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(SF);
  LoRa.setSignalBandwidth(SBW);
  LoRa.setSyncWord(SW);
  }

void set_LoRa_Radio(unsigned long freq,unsigned sbw, int sf, uint8_t sw)  // radio settings, pins
default
{
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
```

```
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
  }


// Terminal IQ mode - comment if in MASTER
#ifdef TERMINAL
void LoRa_rxMode(){
  LoRa.enableInvertIQ();                // active invert I and Q signals
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.disableInvertIQ();               // normal mode
}
#endif

// Master IQ mode - comment if in Terminal
#ifdef GATEWAY
void LoRa_rxMode(){
  LoRa.disableInvertIQ();               // normal mode
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.enableInvertIQ();                // active invert I and Q signals
}
#endif

void onTxDone() {
  Serial.println("TxDone");
  LoRa_rxMode();
}



boolean runEvery(unsigned long interval)
{
  static unsigned long previousMillis = 0;
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval)
  {
    previousMillis = currentMillis;
    return true;
  }
  return false;
}
```

A1.2 Couche réseau/application avec `Lora_Packets.h` :

Ce fichier contient les fonctions d'envoi des paquets de contrôle et de données entre les terminaux et les passerelles.
Les paquets de contrôle sont envoyés en «claire», les paquets de données sont cryptés.

Le fichier contient également la fonction ISR de réception des paquets, selon la taille/type de paquets cette fonction utilise différentes files d'attente pour sortir les paquets reçus (signalement par interruption).

```
// this file describes the types of the LoRa frames used to communicate between the LoRa
terminals
// and the gateways : ThingSpeak gateway and MQTT gateway
// the identifiers: source-sid and destination-did are derived from the chip identifiers.
// Only 4 lower bytes are taken into account (uint32_t)
// The control frame contain 32 bytes, the data frames are built with 64 bytes
// the 2 bytes of control field con[2] are used to identify the tape of the frame: the
control frames and the data frames
// The first con[0] bytes indicates the type of the frame and the type of the gateway
(service) to be used.
// The first hexa value of this byte indicates the type of the gateways/service:

//   1 - ThingSpeak(TS) send, 2 -ThingSpeak(TS) receive
//   3 - MQTT - MQP publish, 4 - MQTT - MQS subscribe,
//   5 - TS send and receive, 6 - MQTT publish and subscribe
//   7 - all services

// The second hexa value indicates the type of the packet:

//   0x01 - IDREQ : ID request for any service and ID acknowledge
//   0x11 - IDREQ, 0x12 - IDACK  : ID request and ID acknowledge - TS send - TSS
//   0x21 - IDREQ, 0x22 - IDACK  : ID request and ID acknowledge - TS receive -TSR
//   0x31 - IDREQ, 0x32 - IDACK  : ID request and ID acknowledge - MQTT publish - MQP
//   0x41 - IDREQ, 0x42 - IDACK  : ID request and ID acknowledge - MQTT subscribe - MQS
//   0x51 - IDREQ, 0x52 - IDACK  : ID request and ID acknowledge - TS send and receive -
TSSR
//   0x61 - IDREQ, 0x62 - IDACK  : ID request and ID acknowledge - MQTT publish and
subscribe - MQPS
//   0x71 - IDREQ, 0x72 - IDACK  : ID request and ID acknowledge - all services - TSMQALL
//
//   0x13 - DTSND, 0x14 - DTACK  : DATA send and DATA acknowledge - TS send  - TSS
//   0x23 - DTREQ, 0x24 - DTRCV  : DATA request and DATA receive - TS receive - TSR
//   0x33 - DTPUB, 0x34 - DTPUB ACK  : DATA publish and DATA publish acknowledge - MQTT
publish - MQP
//   0x43 - DTSUB, 0x44 - DTSUB RCV  : DATA subscribe and DATA subscribe receive - MQTT
subscribe - MQS

#define IDREQ_ALL 0x01
#define IDREQ_TSS 0x11
#define IDREQ_TSR 0x21
#define IDREQ_MQP 0x31
#define IDREQ_MQS 0x41

#define IDACK_ALL 0x02
#define IDACK_TSS 0x12
#define IDACK_TSR 0x22
#define IDACK_MQP 0x32
#define IDACK_MQS 0x42

#define DTSND 0x13  // TSS
#define DTREQ 0x23  // TSR
#define DTPUB 0x33  // MQP
#define IDSUB 0x43  // MQS

#define DTACK 0x14  // TSS
#define DTRCV 0x24  // TSR
#define DTPUBACK 0x34  // MQP
#define DTSUBRCV 0x44  // MQS
```

```
// The second byte of the control field con[1] is used to carry the data fields mask
(ThingSpeak)
// Each bit of this field corresponds to one data value of ThingSpeak channel;
// With this mask the fields may be specified when sending or requesting data

// The password field pass[16] is used by the control frames IDREQ and IDACK to protect
the access
// to the gateway node; only a frame with correct password (16 bytes) is to be answered
by IDACK frame
// The tout value coded on 16 bits may be used to synchronize the terminals operating in
deep sleep mode.
// The gateway none sends the calculated delay according to its scheduler (agenda) to the
terminal node.

// The data frames are different for ThingSpeak serviecs and for MQTT services.
// The data frame relayed by the gateway node to ThingSpeak server contains the

// The following AES encryp and decrypt function are defined to protect the packets with
a symetric crypt keyword

RTC_DATA_ATTR  uint32_t termID,gwID;
RTC_DATA_ATTR  int stage1_flag=0, cycle_cnt=0;
//in stage1 reception flag for control packets: GW-IDREQ, TERM-IDACK
int stage2_flag=0;

#ifdef TERMINAL
RTC_DATA_ATTR  char CKEY[17];  // add 0 to receive CKEY from Master
#endif

#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 );
}

typedef union
{
  uint8_t frame[32];
  struct
  {
    uint32_t did;       // destination identifier chipID (4 lower bytes)
    uint32_t sid;       // source identifier chipID (4 lower bytes)
    uint8_t  con[2];    // control field: con[0]
    char     pass[16];  // password - 16 characters
    int      tout;      // timeout
    uint8_t  pad[2];    // future use
  } pack;               // control packet
```

```
} conframe_t;              // send control frame , receive control frame

typedef union
{
  uint8_t frame[64];    // TS frame to send/receive data
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    uint8_t  con[2];     // control field: lower byte is used as mask
    int     channel;     // TS channel number
    char    keyword[16];   // write (or read) keyword
    float   sens[8];     // max 8 values - fields
    uint16_t  tout;      // optional timeout
  } pack;                // data packet
} TSframe_t;

typedef union
{
  uint8_t frame[112];    // MQTT frame to publish on the given topic
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    uint8_t  con[2];     // control field
    char     topic[48];  // topic name - e.g. /esp32/Term1/Sens1
    char     mess[48];   // message value
    int      tout;      // optional timeout for publish frame
    uint8_t  pad[2];     // future use
  } pack;                // data packet
} MQTTframe_t;

#ifdef TERMINAL

void send_IDREQ(char *pass)  // TERM: request ID for all modes: MODE, termID - global
{
conframe_t scf,sccf;
  LoRa_txMode();
  LoRa.beginPacket();
  scf.pack.did=(uint32_t)0;
  scf.pack.sid=(uint32_t)termID;
  scf.pack.con[0]=(uint8_t)(MODE*16+1);               // IDREQ_MODE - type 1 control TERM
to GW
  ;scf.pack.con[1]=0x00;
  if(pass!=NULL)
     strncpy(scf.pack.pass,pass,16);
  LoRa.write(scf.frame,32);
  LoRa.endPacket(true);
}

void send_DTSND(uint8_t mask,int chan,char *wkey,float *stab)  // TERM: send data to TS,
gwID, termID - global
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);              // MODE 1 - type 3  data TERM to GW
  sdf.pack.channel= chan;
  strncpy(sdf.pack.keyword,wkey,16);
  sdf.pack.con[1]=mask;
  for(int i=0;i<8;i++) sdf.pack.sens[i]=stab[i];
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

```
void send_DTREQ(uint8_t mask,int chan,char *rkey)  // TERM: send data request to TS -
gwID, termID - global
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);          // MODE  2 - type 3  data TERM to GW
  sdf.pack.con[1]=mask;
  sdf.pack.channel= chan;
  strncpy(sdf.pack.keyword,rkey,16);
  for(int i=0;i<8;i++) sdf.pack.sens[i]=0.0;
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}

void send_DTPUB(char *topic, char *mess)  // TERM: send DTPUB packet
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);         // MODE  3 - type 3  data TERM to GW
  sdf.pack.con[1]=0x00;
  strcpy(sdf.pack.topic,topic);
  strcpy(sdf.pack.mess,mess);
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}

void send_DTSUB(char *topic)  // TERM:send DTSUB frame - subscribe topic
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);       // MODE  4 - type 3  data TERM to GW
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  memset(sdf.pack.mess,0x00,48);
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
#endif
#ifdef GATEWAY

void send_IDACK(char *aes, uint16_t tout)  // GW: ID ACK for all modes: code
{
conframe_t scf,sccf;
  LoRa_txMode();
  LoRa.beginPacket();
  scf.pack.did=(uint32_t)termID;
  scf.pack.sid=(uint32_t)gwID;
  scf.pack.con[0]=(uint8_t)16*MODE+2;       // MODE 2 - control GW to TERM
  scf.pack.con[1]=0x00;
  strncpy(scf.pack.pass,aes,16);
  scf.pack.tout=tout;                  // timeout for the next packet
  //encrypt(scf.frame,CKEY,sccf.frame,2);
  LoRa.write(scf.frame,32);
  LoRa.endPacket(true);
}
```

```
void send_DTACK(uint8_t mask,int tout,char *wkey)  // GW: ACK data to TERM
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;
  sdf.pack.con[1]=mask;
              // calculate timeout for the next data frame
  sdf.pack.channel= tout;
  if (wkey!=NULL)
    strncpy(sdf.pack.keyword,wkey,16);           // MODE 1 - type 4: GW to TERM
  for(int i=0;i<8;i++) sdf.pack.sens[i]=0.0;
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}

void send_DTRCV(uint8_t mask,int tout,char *mess,float stab[])  // GW: send received data
in stab[]
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;                       // MODE 2 - type 4: GW to
TERM
  sdf.pack.con[1]=mask;
              // calculated timeout for the next data frame
  sdf.pack.channel= tout;
  strncpy(sdf.pack.keyword,mess,16);
  for(int i=0;i<8;i++) sdf.pack.sens[i]=stab[i];
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}


void send_DTPUBACK(char *topic,char *mess, int tout)  // GW: send ACK for PUB message
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;    // MODE 3 - type 4: GW to TERM
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  strncpy(sdf.pack.mess,mess,48);
  sdf.pack.tout=tout;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}


void send_DTSUBRCV(char *topic,char *mess,int tout)  // GW:send received message
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;   // MODE 4 - type 4: GW to TERM
  sdf.pack.con[1]=0x00;
```

```
  strncpy(sdf.pack.topic,topic,48);
  strncpy(sdf.pack.mess,mess,48);
  sdf.pack.tout=tout;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}

#endif

// this file contains onReceive() ISR that receives and decrypts the received data
packets
// the analysis of the paquets is done in the main task after their reception in the
corresponding queues

QueueHandle_t tsrcv_queue, mqrcv_queue, con_queue;   // receive queue to get out the data
or control packets form onReceive() ISR
int queueSize = 32;

void onReceive(int packetSize)
{
if(packetSize==32)
  {
  conframe_t rcf; int i=0;
  while (LoRa.available()) { rcf.frame[i] = LoRa.read();i++;}
  xQueueReset(con_queue);  // reset queue to keep only the last packet
  xQueueSendFromISR(con_queue, rcf.frame, NULL); Serial.println("Received IDREQ_MODE");
  }

if(packetSize==64)
  {
  TSframe_t rdf,rdcf; int i=0;
  char ckey[17];
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  decrypt(rdcf.frame,ckey,rdf.frame,4);
  xQueueReset(tsrcv_queue);  // reset queue to keep only the last packet
  xQueueSend(tsrcv_queue, rdf.frame, portMAX_DELAY);
  }

if(packetSize==112)
  {
  MQTTframe_t rdf,rdcf; int i=0;
  char ckey[17];
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  decrypt(rdcf.frame,ckey,rdf.frame,7);
  xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
  xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
  }
}
```