Internet de las Cosas

Práctica 3

Introducción

Esta tercera práctica tratará de proporcionar una aproximación práctica a las comunicaciones usando el estándar LoRa para crear una red P2P. Aprenderemos a configurar el módulo de radio en sus aspectos básicos, cómo crear redes privadas y – finalmente – experimentaremos con la transmisión y recepción de datos.

La placa de desarrollo Arduino MKR WAN 1310 integra un módulo de radio Semtech SX1276 [1] que controlaremos con la ayuda de la librería Arduino LoRa [3], aunque usaremos una versión modificada que se puede descargar desde el Campus Virtual de la asignatura. Podemos acceder a una pequeña descripción de los métodos de la librería consultando este enlace.

Debe notarse que el estándar LoRa es prolijo en detalles y permite múltiples modos de configuración y de comunicación que no es posible describir en toda su extensión en el ámbito temporal de esta práctica. Nosotros nos focalizaremos en describir y experimentar con en los modos configuración y uso más empleados.

Configuración del transceiver de radio Semtech SX1276

La librería Arduino LoRa proporciona una interfaz básica para usar el módulo de radio SX1276 integrado en los MKR WAN 1310 en base a la clase LoRaClass que define internamente el objeto LoRa. En este primer ejemplo nos concentraremos en la configuración del módulo de radio en sus aspectos básicos. Los parámetros básicos a tener en cuenta son los siguientes:

- **Banda ICMN.** Los módulos pueden ser configurados para 433, 868 o 915 MHz. Nosotros usaremos constantemente la banda 868. Esta elección se debe indicar con el parámetro 868E6 al iniciar el módulo con el método LoRa.begin(868E6) de la clase LoRaClass.
- **Ancho de banda.** Con LoRa.setSignalBandwidth(500E3) podemos seleccionar un ancho de banda de 500 kHz, el máximo posible, para la modulación LoRa. Es posible elegir cualquiera de los siguientes valores: 7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3, 41.7E3, 62.5E3, 125E3, 250E3, 500E3. Como norma general, multiplicar por dos el ancho de banda supone reducir a la mitad el tiempo de transmisión. Pero reducir el ancho de banda permite, por contra, reducir el impacto de posibles interferencias y obtener distancias de enlace mayores.
- **Factor de dispersión (spreading factor o Spf).** El factor de dispersión del estándar LoRa puede tomar cualquier valor entero en el intervalo [6, 12], siendo 6 un caso particular que requiere de una configuración especial del SX1276. La clase LoRa proporciona el método setSpreadingFactor(Spf) para determinar el factor de dispersión. Aumentar el Spf permite

hacer que la señal aumente su inmunidad al ruido y puedan lograrse distancias de enlace mayores. Sin embargo, el uso de factores de dispersión mayores incremente de forma muy significativa el tiempo de transmisión de un dato, lo que puede provocar un aumento importante en el consumo. La tabla incluida en la Figura.1 ilustra la influencia de este parámetro para un ejemplo concreto, ancho de banda de 125 kHz y 64 bytes de longitud total del mensaje (13 bytes de cabecera y 53 de carga útil o payload). Nótese cómo la combinación de ancho de banda, factor de dispersión y tamaño del mensaje se combinan para determinar la tasa máxima de envío de paquetes para un duty cycle del 1%. A través de este enlace se puede acceder a una aplicación que nos permite reproducir los resultados de la Figura.1 para otras configuraciones o combinaciones de parámetros.

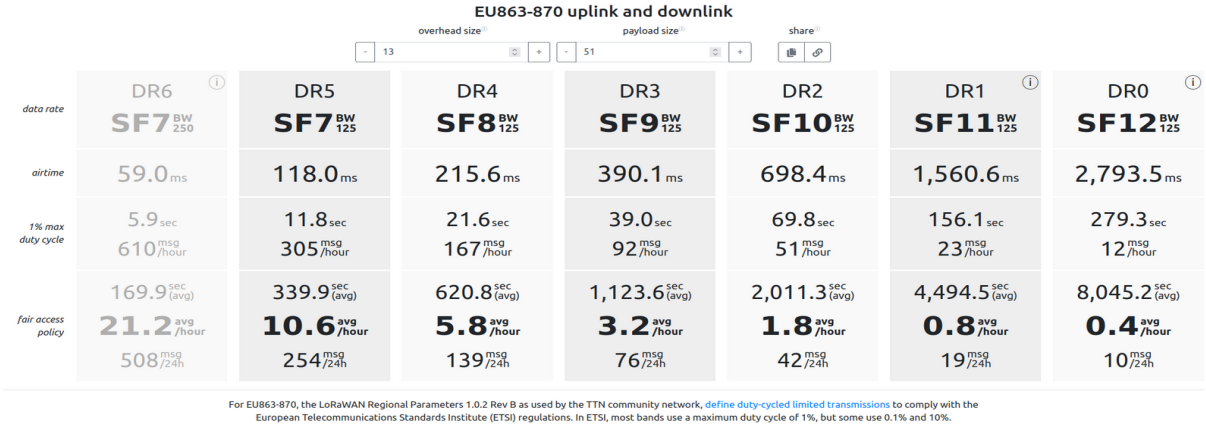

Figura 1: Tiempos de transmisión de un paquete de 64 bytes con ancho de banda de 125 kHz y diferentes factores de dispersión.

- **Tasa de codificación o coding rate.** El estándar LoRa permite incluir bits adicionales a la hora de codificar y transmitir un byte de datos. Esta estrategia permite detectar y corregir, hasta cierto punto, los errores que se produzcan en la recepción. Como contrapartida, aumentar la tasa de codificación incrementa el número de símbolos a transmitir y, por tanto, el tiempo de transmisión. El método setCodingRate4(CR) del objeto LoRa posibilita el ajuste de la tasa de codificación. CR representa el numerador en el cociente 4/CR y puede tomar cualquier valor entero en el intervalo [5, 8]. Como hemos comentado, el tiempo de transmisión se incrementa a medida que aumentamos el valor de CR.
- **Potencia de transmisión.** Mediante el método de la clase LoRaClass setTxPower(pwdBm, PA_OUTPUT_PA_BOOST_PIN) es posible seleccionar la potencia de las transmisiones. El parámetro pwdBm indicaría la potencia deseada en dBm y puede tomar valores enteros en el intervalo [2, 20]. En general, ajustaremos la potencia de emisión en función de las condiciones de enlace para lograr un enlace fiable con un consumo de energía mínimo. Para las pruebas en interior es aconsejable poner este valor al mínimo y evitar así saturar a los receptores. Si esto se produce, la transmisión fallará sin causa aparente.
- **Palabra de sincronización (SyncWord).** La palabra de sincronización es un número de hasta 8 octetos que sirve para identificar la identidad de la red. Los valores por defecto son 0x12 para una red privada y 0x34 para una red pública (LoRaWAN, por ejemplo). Nosotros usaremos una palabra de sincronización de 1 byte para identificar la red privada de cada grupo de prácticas, de manera que cada radio solo reciba los mensajes que generen otras radios que compartan la misma palabra de sincronización. Esto exigirá que los diferentes grupos se pongan de acuerdo para seleccionar diferentes palabras de sincronización, por

ejemplo, 0x12, 0x22, 0x32, … 0xF2. La librería Arduino LoRa proporciona el método setSyncWord(syncWord) para establecer una palabra de sincronización de un único byte. Ninguno de los octetos de la palabra de sincronización puede ser nulo (0x00).

- **Preámbulo.** El preámbulo forma parte de la parte del mensaje que se usa para sincronizar el receptor con el mensaje que se transmite. Por defecto la longitud del preámbulo es de 8 símbolos, pero puede tomar cualquier valor entero en el intervalo [6, 65535]. El receptor debe configurarse con un preámbulo con la misma longitud que la usada con el transmisor. Valores mayores inciden positivamente en la robustez de la transmisión, pero en situaciones donde las comunicaciones sean frecuentes puede ser necesario reducir el preámbulo para reducir los tiempos de transmisión y respetar el límite de duty cycle. El método setPreambleLength(ns) de la clase LoRaClass permite fijar el número de símbolos, ns, que se usarán como preámbulo.

El resultado de seleccionar valores para estos parámetros se reflejará en el contenido de los registros de configuración del módulo de radio (consultar la sección 4 del manual del módulo SX1276 [2]). La librería LoRa dispone de un método público, dumpRegisters(SerialPort), para proyectar por el puerto serie indicado (SerialPort) el contenido de los registros de configuración del transceiver SX1276. Este método ha sido modificado para incluir un resumen de los parámetros que se han descrito más arriba que resulte más fácil de analizar.

El ejemplo siguiente usa dumpRegisters() para mostrar primero la configuración por defecto, tras inicializar el módulo de radio con el método begin() de la clase LoRaClass, y después, tras modificar algunos parámetros de la configuración.

Invocando el método sleep() podemos poner el módulo de radio en un estado de bajo consumo de energía.

```
/* ----------------------------------------------------------------------
 *  Ejemplo MKR1310_LoRaConfig_DumpRegisters
 *  Práctica 3
 *  Asignatura (GII-IoT)
 *
 *  Este ejemplo requiere de una versión modificada
 *  de la librería Arduino LoRa (descargable desde
 *  CV de la asignatura
 *
 *  También usa la librería Arduino_BQ24195
 *  https://github.com/arduino-libraries/Arduino_BQ24195
 * ----------------------------------------------------------------------
 */
#include <SPI.h>
#include <LoRa.h>
#include <Arduino_PMIC.h>

void setup()
{
  SerialUSB.begin(9600);
  while (!SerialUSB);
```

```
  if (!init_PMIC()) {
    SerialUSB.println("Initilization of BQ24195L failed!");
  }
  else {
    SerialUSB.println("Initilization of BQ24195L succeeded!");
  }

  if (!LoRa.begin(868E6)) {      // Initicializa a 868 MHz
    SerialUSB.println("LoRa init failed. Check your connections.");
    while (true);
  }

  Serial.println("\nLoRa Dump Registers");
  SerialUSB.println("\nDefault configuration:");
  LoRa.dumpRegisters(SerialUSB);


  LoRa.setSignalBandwidth(500E3);// 7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3
                                 // 41.7E3, 62.5E3, 125E3, 250E3, 500E3
                                 // Multiplicar por dos el ancho de banda
                                 // supone dividir a la mitad el tiempo de Tx
  LoRa.setSpreadingFactor(8);    // [6, 12] Aumentar el spreading factor incrementa
                                 // de forma significativa el tiempo de Tx
                                 // SPF = 6 es un valor especial
                                 // Ver tabla 12 del manual del SEMTECH SX1276
  LoRa.setSyncWord(0x12);        // Palabra de sincronización privada por defecto
                                 // para SX127X
                                 // Usaremos la palabra de sincronización para
                                 // crear diferentes redes privadas por equipos
  LoRa.setCodingRate4(5);        // [5, 8] 5 da un tiempo de Tx menor
  LoRa.setPreambleLength(8);     // Número de símbolos a usar como preámbulo

  LoRa.setTxPower(3, PA_OUTPUT_PA_BOOST_PIN); // Rango [2, 20] en dBm
                                // Importante seleccionar un valor bajo para pruebas
                                // a corta distancia y evitar saturar al receptor

  SerialUSB.println("\nCurrent configuration:");
  LoRa.dumpRegisters(SerialUSB); // Ver sección 4 del manual del SEMTECH SX1276
                                 // Nos permite ver cómo esta configuración se
                                 // refleja a nivel de registros

  LoRa.sleep();                  // Ver tabla 16 del manual del SEMTECH SX1276
}
```

```
void loop()  { }
```

En el código proporcionado para este ejemplo debe notarse el uso de la librería Arduino BQ24295L [4]. Esta librería, aún en fase beta, nos permite controlar el integrado encargado de gestionar la batería del MKR WAN 1310. La pestaña BQ24195L_PMIC contiene los detalles de configuración de este aspecto de la placa y cómo podemos habilitar la descarga y recarga de la batería o supervisar algunos aspectos de la misma. Sorprendentemente, no existe ninguna forma de consultar el nivel de carga (SOC, state of charge) de la batería a través de esta librería. Evidentemente, este detalle es resoluble, por ejemplo, conectando un simple circuito divisor de voltaje a uno de los pines analógicos del micro.

Transmisión y recepción de mensajes

Otros protocolos de comunicación por radio en banda ICM incluyen la dirección MAC del destinatario y del remitente en la estructura del paquete. LoRa contempla esta posibilidad en el estándar, pero la librería LoRa no la implementa. En este apartado exploramos dos posibles soluciones a este problema.

Una primera posibilidad, recomendada en algunos foros, se basa en usar los dos primeros octetos del mensaje para implementar una solución que nos permite enviar un paquete a un destinatario específico. La idea es muy simple y consiste en usar los dos primeros bytes del mensaje (payload) para consignar la dirección del destinatario y del remitente.

No obstante, esta “solución” presenta importantes limitaciones. Por una parte, es evidente que el paquete puede ser recibido por cualquier dispositivo LoRa que se encuentre en el radio de alcance de la señal. Por otra, la viabilidad de este esquema de direccionamiento recae exclusivamente en el desarrollador, que debe garantizar la coherencia del esquema de asignación de direcciones a dispositivos.

Una alternativa más interesante se basa en la utilización de la palabra de sincronización. La sección 2.1.13.6. (Packet Filtering) del manual del SX1276 [2] indica que la palabra de sincronización (SyncWord) puede usarse como filtro para independizar diferentes redes. Las radios que pertenezcan a una misma red deberán establecer la misma palabra de sincronización. De otra forma, las radios no podrán comunicarse.

La palabra de sincronización puede tener de uno a ocho bytes, de los cuales ninguno puede ser nulo (0x00). No obstante, la librería Arduino LoRa solo soporta palabras de sincronización de un único octeto.

Adicionalmente, pueden activarse otros filtros que se pueden basar en direcciones o en la longitud del paquete, si este es fijo. Consultar la sección 2.1.13.6 en [2] para más información.

La red está llena de foros especializados donde se comenta que el uso de palabras de sincronización arbitrarias puede afectar a la robustez de las comunicaciones, básicamente una pérdida de radio de

alcance. Nosotros ignoraremos este riesgo potencial y usaremos diferentes palabras de sincronización para aislar las radios de cada grupo de prácticas.

El siguiente ejemplo muestra cómo es posible componer mensajes LoRa y enviarlos, al tiempo que se pueden recibir mensajes de forma asíncrona. Es importante destacar algunos aspectos:

- El envío de mensajes se realizará entre equipos que compartan la misma palabra de sincronización, 0x12 en este ejemplo, que deberá adaptarse entre los grupos que participen en la misma red privada.
- Además, cada nodo deberá recibir una dirección única en el rango 0x00 – 0xFE que servirá para identificarle dentro de la red privada. Por convenio, la dirección 0xFF se usará como dirección de difusión o broadcast.
- En este ejemplo, la recepción de mensajes se resuelve de forma asíncrona, mediante el callback receive() que, por convenio, debe ser una función de tipo void sin parámetros. El método onReceive() de la clase nos permite asociar la función receive() como callback.
- Nótese que la recepción y la transmisión de mensajes es mutuamente exclusiva. En este ejemplo, el transceiver está normalmente en modo recepción, modo que se suspenderá al iniciar una transmisión y mientras esta no se complete.
- La composición del mensaje comienza invocando `LoRa.beginPacket()` y concluye con `LoRa.endPacket()`. El mensaje se compone byte a byte y tenemos completa libertad para definir la organización interna del mensaje. El único límite que es necesario observar es el tamaño del paquete que no debe superar en ningún caso los 222 bytes para un SPF igual a 7. El uso de factores de dispersión mayores puede exigir acortar el tamaño del paquete por debajo de este valor.
- El argumento por defecto del método `LoRa.endPacket()`es false. Esto significa que no retornará hasta que la transmisión del paquete LoRa no hay concluido. Esto permite estimar aproximadamente el tiempo requerido para la transmisión.
- En este ejemplo no se controla de forma explícita que se respete el límite del 1% sobre el ciclo de trabajo.

IMPORTANTE: A partir de ese ejemplo es necesario tener conectada la antena antes de cargar y ejecutar estos ejemplos. NUNCA transmitir sin tener la antena conectada. El conector U.FL de la antena es extremadamente frágil por lo que cada grupo debe buscar una forma de asegurar la antena para evitar que pueda dañarse. Asimismo, debe ponerse cuidado a la hora de insertar o extraer el conector U.FL del zócalo de la placa MKR 1310.

```
/* ----------------------------------------------------------------------
 *  Ejemplo MKR1310_LoRa_SendReceive_WithReceiveCallback
 *  Práctica 3
 *  Asignatura (GII-IoT)
 *
 *  Basado en el ejemplo LoRaDuplexCallback de la librería
 *  LoRa demuestra cómo es posible resolver la recepción
 *  de mensajes de forma asíncrona.
```

```
 *
 *  Este ejemplo requiere de una versión modificada
 *  de la librería Arduino LoRa (descargable desde
 *  CV de la asignatura.
 *
 *  También usa la librería Arduino_BQ24195
 *  https://github.com/arduino-libraries/Arduino_BQ24195
 * ----------------------------------------------------------------------
 */

#include <SPI.h>
#include <LoRa.h>
#include <Arduino_PMIC.h>

#define TX_LAPSE_MS          10000

// NOTA: Ajustar estas variables
const uint8_t localAddress = 0xB0;   // Dirección de este dispositivo
uint8_t destination = 0xFF;          // Dirección de destino, 0xFF es
                                     // la dirección de broadcast

// ----------------------------------------------------------------------
// Setup function
// ----------------------------------------------------------------------
void setup()
{
  Serial.begin(9600);
  while (!Serial);

  Serial.println("LoRa Duplex with callback");

  // Es posible indicar los pines para CS, reset e IRQ pins (opcional)
  // LoRa.setPins(csPin, resetPin, irqPin);// set CS, reset, IRQ pin

  if (!init_PMIC()) {
    Serial.println("Initilization of BQ24195L failed!");
  }
  else {
    Serial.println("Initilization of BQ24195L succeeded!");
  }

  if (!LoRa.begin(868E6)) {      // Initicializa LoRa a 868 MHz
    Serial.println("LoRa init failed. Check your connections.");
    while (true);
  }
```

```
  // Configuramos algunos parámetros de la radio
  LoRa.setSignalBandwidth(125E3);// 7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3
                                 // 41.7E3, 62.5E3, 125E3, 250E3, 500E3
                                 // Multiplicar por dos el ancho de banda
                                 // supone dividir a la mitad el tiempo de Tx
  LoRa.setSpreadingFactor(7);    // [6, 12] Aumentar el spreading factor incrementa
                                 // de forma significativa el tiempo de Tx
                                 // SPF = 6 es un valor especial
                                 // Ver tabla 12 del manual del SEMTECH SX1276
  LoRa.setSyncWord(0x12);        // Palabra de sincronización privada por defecto
                                 // para SX127X. Usaremos la palabra de
                                 // sincronización para crear diferentes redes
                                 // privadas por equipos
  LoRa.setCodingRate4(5);        // [5, 8] 5 da un tiempo de Tx menor
  LoRa.setPreambleLength(8);     // Número de símbolos a usar como preámbulo

  LoRa.setTxPower(3, PA_OUTPUT_PA_BOOST_PIN); // Rango [2, 20] en dBm
                                 // Importante seleccionar un valor bajo para
                                 // pruebas a corta distancia y evitar saturar al
                                 // receptor

  // Indicamos el callback para cuando se reciba un paquete
  LoRa.onReceive(onReceive);

  // Nótese que la recepción está activada a partir de este punto
  LoRa.receive();

  Serial.println("LoRa init succeeded\n");
}

// ----------------------------------------------------------------------
// Loop function
// ----------------------------------------------------------------------
void loop()
{
  static uint32_t lastSendTime_ms = 0;
  static uint16_t msgCount = 0;
  static uint32_t txInterval_ms = TX_LAPSE_MS;

  if ((millis() - lastSendTime_ms) > txInterval_ms) {
    char message[50];

    snprintf(message, sizeof(message),"Message no. %03d from 0x%02X",
             msgCount, localAddress);
```

```
    uint32_t beginTX_ms = millis();
    sendMessage(message, uint8_t(strlen(message)), msgCount);
    uint32_t endTX_ms = millis();
    Serial.print("Sent message ' ");
    Serial.print(message);
    Serial.print("' in ");
    Serial.print(endTX_ms - beginTX_ms);
    Serial.println(" msecs\n");

    lastSendTime_ms = millis();
    txInterval_ms = random(TX_LAPSE_MS) + 1000;

    // Reactivamos la recepción de mensajes, que se desactiva
    // en segundo plano mientras se transmite
    LoRa.receive();
  }
}

// ---------------------------------------------------------------------
// Sending message function
// ---------------------------------------------------------------------
void sendMessage(char* outgoing, uint8_t msgLength, uint16_t &msgCount)
{
  LoRa.beginPacket();                   // Comenzamos el empaquetado del mensaje
  LoRa.write(destination);              // Añadimos el ID del destinatario
  LoRa.write(localAddress);             // Añadimos el ID del remitente
  LoRa.write((uint8_t)(msgCount >> 7)); // Añadimos el Id del mensaje (MSB primero)
  LoRa.write((uint8_t)(msgCount & 0xFF));
  LoRa.write(msgLength);               // Añadimos la longitud en bytes del mensaje
  LoRa.print(outgoing);                // Añadimos el mensaje/payload
  LoRa.endPacket();                    // Finalizamos el paquete y lo enviamos
  msgCount++;                          // Incrementamos el contador de mensajes
}

// ---------------------------------------------------------------------
// Callback para recibir mensajes
// ---------------------------------------------------------------------
void onReceive(int packetSize)
{
  if (packetSize == 0) return;          // Si no hay mensajes, retornamos

  // Leemos los primeros bytes del mensaje
  char buffer[50];                      // Buffer para almacenar el mensaje
  int recipient = LoRa.read();          // Dirección del destinatario
```

```
  uint8_t sender = LoRa.read();         // Dirección del remitente
                                        // msg ID (High Byte first)
  uint16_t incomingMsgId = ((uint16_t)LoRa.read() << 7) |
                           (uint16_t)LoRa.read();

  uint8_t incomingLength = LoRa.read(); // Longitud en bytes del mensaje

  uint8_t receivedBytes = 0;            // Leemos el mensaje byte a byte
  while (LoRa.available() && (receivedBytes < uint8_t(sizeof(buffer)-1))) {
    buffer[receivedBytes++] = (char)LoRa.read();
  }
  buffer[receivedBytes] = '\0';         // Terminamos la cadena

  if (incomingLength != receivedBytes) {// Verificamos la longitud del mensaje
    Serial.print("Receiving error: declared message length " +
                 String(incomingLength));
    Serial.println(" does not match length " + String(receivedBytes));
    return;
  }

  // Verificamos si se trata de un mensaje en broadcast o es un mensaje
  // dirigido específicamente a este dispositivo.
  // Nótese que este mecanismo es complementario al uso de la misma
  // SyncWord y solo tiene sentido si hay más de dos receptores activos
  // compartiendo la misma palabra de sincronización
  if ((recipient & localAddress) != localAddress ) {
    Serial.println("Receiving error: This message is not for me.");
    return;
  }

  // Imprimimos los detalles del mensaje recibido
  Serial.println("Received from: 0x" + String(sender, HEX));
  Serial.println("Sent to: 0x" + String(recipient, HEX));
  Serial.println("Message ID: " + String(incomingMsgId));
  Serial.println("Message length: " + String(incomingLength));
  Serial.println("Message: " + String(buffer));
  Serial.print("RSSI: " + String(LoRa.packetRssi()));
  Serial.println(" dBm\nSNR: " + String(LoRa.packetSnr()));
  Serial.println();
}
```

Es posible introducir algunas modificaciones en el programa anterior para conseguir que la el módulo de radio notifique mediante una interrupción cuando ha concluido la transmisión del paquete. Esto es importante, en particular cuando los tiempos de transmisión empiezan a ser de cientos de milisegundos, para evitar bloquear el micro durante este periodo. El siguiente ejemplo es una variación del anterior para conseguir que la transmisión de sea bloqueante.

El primer paso es habilitar un callback, que se invocará cada vez que se complete la transmisión de un paquete, con el método onTxDone() al que se pasará como argumento el nombre de la función que actuará como callback, TxFinished() en el siguiente ejemplo.

A continuación, se invoca la terminación y transmisión del paquete en modo no bloqueante con `LoRa.endPacket(true)`. También se modifica la función loop() para no activar el modo de recepción o iniciar el envío de un nuevo paquete hasta que la anterior transmisión haya concluido.

Adicionalmente, hemos introducido un mecanismo de control sobre el ciclo de trabajo (duty-cycle), o la proporción entre el tiempo que dura la transmisión del paquete y el intervalo entre transmisiones. Se ha supuesto un límite del 1% para el ciclo de trabajo. Se ha mantenido una cierta aleatoriedad en el periodo de emisión para verificar que el mecanismo de control del ciclo de trabajo opera correctamente. El siguiente ejemplo incorpora estos cambios destacados en negrita.

```
/* ----------------------------------------------------------------------
 *  Ejemplo MKR1310_LoRa_SendReceive_WithCallbacks
 *  Práctica 3
 *  Asignatura (GII-IoT)
 *
 *  Basado en el ejemplo MKR1310_LoRa_SendReceive_WithReceiveCallback,
 *  muestra cómo es posible resolver la transmisión
 *  y recepción de mensajes de forma asíncrona.
 *  Adicionalmente, monitorea el duty-cycle e intenta
 *  ajustar el intervalo entre la transmisión de paquetes
 *  para mantener el duty cycle por debajo del 1%.
 *
 *  Este ejemplo requiere de una versión modificada
 *  de la librería Arduino LoRa (descargable desde
 *  CV de la asignatura.
 *
 *  También usa la librería Arduino_BQ24195
 *  https://github.com/arduino-libraries/Arduino_BQ24195
 * ----------------------------------------------------------------------
 */
#include <SPI.h>
#include <LoRa.h>
#include <Arduino_PMIC.h>

#define TX_LAPSE_MS          10000

// NOTA: Ajustar estas variables
const uint8_t localAddress = 0xBB;     // Dirección de este dispositivo
uint8_t destination = 0xFF;            // Dirección de destino, 0xFF es la
dirección de broadcast
```

```
volatile bool txDoneFlag = true;       // Flag para indicar cuando ha finalizado
                                       // una transmisión

// ----------------------------------------------------------------------
// Setup function
// ----------------------------------------------------------------------
void setup()
{
  Serial.begin(9600);
  while (!Serial);

  Serial.println("LoRa Duplex with TxDone and Receive callbacks");

  // Es posible indicar los pines para CS, reset e IRQ pins (opcional)
  // LoRa.setPins(csPin, resetPin, irqPin);// set CS, reset, IRQ pin

    if (!init_PMIC()) {
    Serial.println("Initilization of BQ24195L failed!");
  }
  else {
    Serial.println("Initilization of BQ24195L succeeded!");
  }

  if (!LoRa.begin(868E6)) {      // Initicializa LoRa a 868 MHz
    Serial.println("LoRa init failed. Check your connections.");
    while (true);
  }

  // Configuramos algunos parámetros de la radio
  LoRa.setSignalBandwidth(125E3); // 7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3
                                  // 41.7E3, 62.5E3, 125E3, 250E3, 500E3
                                  // Multiplicar por dos el ancho de banda
                                  // supone dividir a la mitad el tiempo de Tx
  LoRa.setSpreadingFactor(7);     // [6, 12] Aumentar el spreading factor
                                  // incrementa de forma significativa el tiempo
                                  // de Tx. SPF = 6 es un valor especial
                                  // Ver tabla 12 del manual del SEMTECH SX1276
  LoRa.setSyncWord(0x12);         // Palabra de sincronización privada por defecto
                                  // para SX127X
                                  // Usaremos la palabra de sincronización para
                                  // crear diferentes
                                  // redes privadas por equipos
  LoRa.setCodingRate4(5);         // [5, 8] 5 da un tiempo de Tx menor
  LoRa.setPreambleLength(8);      // Número de símbolos a usar como preámbulo
```

```
  LoRa.setTxPower(3, PA_OUTPUT_PA_BOOST_PIN); // Rango [2, 20] en dBm
                                  // Importante seleccionar un valor bajo para
                                  // pruebas a corta distancia y evitar saturar al
                                  // receptor

  // Indicamos el callback para cuando se reciba un paquete
  LoRa.onReceive(onReceive);

  // Nótese que la recepción está activada a partir de este punto
  LoRa.receive();

  // Activamos el callback que nos indicará cuando ha finalizado la
  // transmisión de un mensaje
  LoRa.onTxDone(TxFinished);

  Serial.println("LoRa init succeeded\n");
}

// ------------------------------------------------------------------
// Loop function
// ------------------------------------------------------------------
void loop()
{
  static uint32_t lastSendTime_ms = 0;
  static uint16_t msgCount = 0;
  static uint32_t txInterval_ms = TX_LAPSE_MS;
  static uint32_t tx_begin_ms = 0;
  static bool transmitting = false;

  if (!transmitting && ((millis() - lastSendTime_ms) > txInterval_ms)) {
    char message[50];
    snprintf(message, sizeof(message),"Message no. %03d from 0x%02X",
             msgCount, localAddress);

    transmitting = true;
    txDoneFlag = false;
    tx_begin_ms = millis();

    sendMessage(message, uint8_t(strlen(message)), msgCount);
    Serial.print("Sending '");
    Serial.print(message);
    Serial.print("' ");
  }

  if (transmitting && txDoneFlag) {
```

```
    uint32_t TxTime_ms = millis() - tx_begin_ms;
    Serial.print("----> TX completed in ");
    Serial.print(TxTime_ms);
    Serial.println(" msecs");

    // Ajustamos txInterval_ms para respetar un duty cycle del 1%
    uint32_t lapse_ms = tx_begin_ms - lastSendTime_ms;
    lastSendTime_ms = tx_begin_ms;
    float duty_cycle = (100.0f * TxTime_ms) / lapse_ms;

    Serial.print("Duty cycle: ");
    Serial.print(duty_cycle,1);
    Serial.println(" %\n");

    // Solo si el ciclo de trabajo es superior al 1% lo ajustamos
    // Dejamos random() solo para introducir cierta variabilidad
    // y verificar que el mecanismo corrector funciona
    if (duty_cycle <= 1.0f) {
      txInterval_ms = random(TX_LAPSE_MS) + 1000;
    } else {
      txInterval_ms = TxTime_ms * 100;
    }

    transmitting = false;

    // Reactivamos la recepción de mensajes, que se desactiva
    // en segundo plano mientras se transmite
    LoRa.receive();
  }
}

// ---------------------------------------------------------------------
// Sending message function
// ---------------------------------------------------------------------
void sendMessage(char* outgoing, uint8_t msgLength, uint16_t &msgCount)
{
  LoRa.beginPacket();                   // Comenzamos el empaquetado del mensaje
  LoRa.write(destination);              // Añadimos el ID del destinatario
  LoRa.write(localAddress);             // Añadimos el ID del remitente
  LoRa.write((uint8_t)(msgCount >> 7)); // Añadimos el Id del mensaje (MSB primero)
  LoRa.write((uint8_t)(msgCount & 0xFF));
  LoRa.write(msgLength);              // Añadimos la longitud en bytes del mensaje
  LoRa.print(outgoing);               // Añadimos el mensaje/payload
  LoRa.endPacket(true);               // Finalizamos el paquete, pero no esperamos a
                                      // finalice su transmisión
```

```
  msgCount++;                       // Incrementamos el contador de mensajes
}

// ---------------------------------------------------------------------
// Receiving message function
// ---------------------------------------------------------------------
void onReceive(int packetSize)
{
  if (packetSize == 0) return;          // Si no hay mensajes, retornamos

  // Leemos los primeros bytes del mensaje
  char buffer[50];                      // Buffer para almacenar el mensaje
  int recipient = LoRa.read();          // Dirección del destinatario
  uint8_t sender = LoRa.read();         // Dirección del remitente
                                        // msg ID (High Byte first)
  uint16_t incomingMsgId = ((uint16_t)LoRa.read() << 7) |
                           (uint16_t)LoRa.read();

  uint8_t incomingLength = LoRa.read(); // Longitud en bytes del mensaje

  uint8_t receivedBytes = 0;            // Leemos el mensaje byte a byte
  while (LoRa.available() && (receivedBytes < uint8_t(sizeof(buffer)-1))) {
    buffer[receivedBytes++] = (char)LoRa.read();
  }
  buffer[receivedBytes] = '\0';         // Terminamos la cadena
  if (incomingLength != receivedBytes) {// Verificamos la longitud del mensaje
    Serial.print("Receiving error: declared message length " +
                 String(incomingLength));
    Serial.println(" does not match length " + String(receivedBytes));
    return;
  }

  // Verificamos si se trata de un mensaje en broadcast o es un mensaje
  // dirigido específicamente a este dispositivo.
  // Nótese que este mecanismo es complementario al uso de la misma
  // SyncWord y solo tiene sentido si hay más de dos receptores activos
  // compartiendo la misma palabra de sincronización
  if ((recipient & localAddress) != localAddress ) {
    Serial.println("Receiving error: This message is not for me.");
    return;
  }

  // Imprimimos los detalles del mensaje recibido
  Serial.println("Received from: 0x" + String(sender, HEX));
  Serial.println("Sent to: 0x" + String(recipient, HEX));
```

```
  Serial.println("Message ID: " + String(incomingMsgId));
  Serial.println("Message length: " + String(incomingLength));
  Serial.println("Message: " + String(buffer));
  Serial.print("RSSI: " + String(LoRa.packetRssi()));
  Serial.println(" dBm\nSNR: " + String(LoRa.packetSnr()));
  Serial.println();
}

void TxFinished()
{
  txDoneFlag = true;
}
```

Envío y recepción de mensajes binarios

Habida cuenta de cómo se incrementa el tiempo de transmisión con la longitud de los paquetes, en redes LoRa resulta de capital importancia tratar de minimizar el volumen de datos a transmitir. Por ejemplo, con paquetes de datos que incluyan datos reales resulta ventajoso incluir estos datos, no como cadenas de caracteres ASCII, sino en formato binario. Sobre esta idea básica, de conformar el payload en un formato binario, es posible aplicar otras que reduzcan aún más la longitud del paquete, pero la estrategia óptima dependerá de las condiciones que se deriven de cada escenario concreto de aplicación. El portal de LoRa para desarrolladores incluye este interesante enlace [5], donde se revisan varias ideas para reducir el tamaño del payload.

CayenneLPP [6] es una librería que permite empaquetar y desempaquetar datos estereotipados (temperatura, humedad, latitud, longitud, …) de forma cómoda y segura en paquetes binarios compactos. Un aspecto interesante de esta librería es que es posible conformar el contenido de un paquete de datos de forma dinámica. Los tipos de datos que incluyen siguen el IPSO Alliance Smart Objects Guidelines, que identifica cada tipo de dato con un “Object ID” pero limitado a un único octeto. El uso de esta librería no logra una compactación máxima del payload, pero sí facilita la integración de una red de sensores que comuniquen paquetes binarios con redes LoRaWAN

En este apartado exploraremos el uso de paquetes binarios para transmitir datos sobre la configuración de las radios y sobre los niveles de RSSI y SNR de los paquetes recibidos. Estos datos podrían servir como base para trazar una táctica de minimización de la duración de las transmisiones.

Veamos primero cómo podemos compactar la configuración de la radio en sus parámetros fundamentales:

- Ancho de banda (A): 7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3, 41.7E3, 62.5E3, 125E3, 250E3, 500E3 se pueden identificar con un identificador [0, 9] (4 bits), cero para 7.8E3, 9 para 500E3.
- Factor de dispersión (F): [6, 12] → [0, 6] (3 bits)
- Tasa de codificación (C): [5, 8] → [0, 3] (2 bits)
- Potencia de transmisión (T): [2, 20 dBm] → [0, 18] (5 bits)
- Preámbulo: se supondrá fija a 8 símbolos.

Es fácil ver que podemos transmitir estos datos de forma compacta, empleando solo 14 bits, mediante dos octetos:

Primer byte: A3 A2 A1 A0 F2 F1 F0 X

Segundo byte: C1 C0 T4 T3 T2 T1 T0 X

donde el ancho de banda ocuparía los 4 bits altos del primer octeto, el factor de dispersión los tres siguiente y así sucesivamente. X indica un bit sin contenido (nulo).

El ejemplo siguiente envía esta información y añade también el RSSI y el SNR del último paquete recibido, cada uno de ellos en un octeto.

```
/* ----------------------------------------------------------------------
 *  Ejemplo MKR1310_LoRa_SendReceive_Binary
 *  Práctica 3
 *  Asignatura (GII-IoT)
 *
 *  Basado en el ejemplo MKR1310_LoRa_SendReceive_WithCallbacks,
 *  muestra cómo es posible comunicar los parámetros de
 *  configuración del transceiver entre nodos LoRa en
 *  formato binario *
 *
 *  Este ejemplo requiere de una versión modificada
 *  de la librería Arduino LoRa (descargable desde
 *  CV de la asignatura.
 *
 *  También usa la librería Arduino_BQ24195
 *  https://github.com/arduino-libraries/Arduino_BQ24195
 * ----------------------------------------------------------------------
 */

#include <SPI.h>
#include <LoRa.h>
#include <Arduino_PMIC.h>

#define TX_LAPSE_MS          10000

// NOTA: Ajustar estas variables
const uint8_t localAddress = 0xBB;     // Dirección de este dispositivo
uint8_t destination = 0xFF;            // Dirección de destino, 0xFF es la
dirección de broadcast

volatile bool txDoneFlag = true;       // Flag para indicar cuando ha finalizado
una transmisión
volatile bool transmitting = false;
```

```
// Estructura para almacenar la configuración de la radio
typedef struct {
  uint8_t bandwidth_index;
  uint8_t spreadingFactor;
  uint8_t codingRate;
  uint8_t txPower;
} LoRaConfig_t;

double bandwidth_kHz[10] = {7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3,
                            41.7E3, 62.5E3, 125E3, 250E3, 500E3 };

LoRaConfig_t thisNodeConf   = { 6, 10, 5, 2};
LoRaConfig_t remoteNodeConf = { 0,  0, 0, 0};
int remoteRSSI = 0;
float remoteSNR = 0;

// ----------------------------------------------------------------------
// Setup function
// ----------------------------------------------------------------------
void setup()
{
  Serial.begin(115200);
  while (!Serial);

  Serial.println("LoRa Duplex with TxDone and Receive callbacks");
  Serial.println("Using binary packets");

  // Es posible indicar los pines para CS, reset e IRQ pins (opcional)
  // LoRa.setPins(csPin, resetPin, irqPin);// set CS, reset, IRQ pin


  if (!init_PMIC()) {
    Serial.println("Initilization of BQ24195L failed!");
  }
  else {
    Serial.println("Initilization of BQ24195L succeeded!");
  }

  if (!LoRa.begin(868E6)) {      // Initicializa LoRa a 868 MHz
    Serial.println("LoRa init failed. Check your connections.");
    while (true);
  }

  // Configuramos algunos parámetros de la radio
  LoRa.setSignalBandwidth(long(bandwidth_kHz[thisNodeConf.bandwidth_index]));
```

```
                                // 7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3
                                // 41.7E3, 62.5E3, 125E3, 250E3, 500E3
                                // Multiplicar por dos el ancho de banda
                                // supone dividir a la mitad el tiempo de Tx

  LoRa.setSpreadingFactor(thisNodeConf.spreadingFactor);
                                // [6, 12] Aumentar el spreading factor incrementa
                                // de forma significativa el tiempo de Tx
                                // SPF = 6 es un valor especial
                                // Ver tabla 12 del manual del SEMTECH SX1276

  LoRa.setCodingRate4(thisNodeConf.codingRate);
                                // [5, 8] 5 da un tiempo de Tx menor

  LoRa.setTxPower(thisNodeConf.txPower, PA_OUTPUT_PA_BOOST_PIN);
                                 // Rango [2, 20] en dBm.  Importante
                                 // seleccionar un valor bajo para pruebas
                                 // a corta distancia y evitar saturar al receptor
  LoRa.setSyncWord(0x12);        // Palabra de sincronización privada por defecto
                                 // para SX127X
                                 // Usaremos la palabra de sincronización para
                                 // crear diferentesredes privadas por equipos
  LoRa.setPreambleLength(8);     // Número de símbolos a usar como preámbulo


  // Indicamos el callback para cuando se reciba un paquete
  LoRa.onReceive(onReceive);

  // Activamos el callback que nos indicará cuando ha finalizado la
  // transmisión de un mensaje
  LoRa.onTxDone(TxFinished);

  // Nótese que la recepción está activada a partir de este punto
  LoRa.receive();

  Serial.println("LoRa init succeeded.\n");
}

// ----------------------------------------------------------------------
// Loop function
// ----------------------------------------------------------------------
void loop()
{
  static uint32_t lastSendTime_ms = 0;
  static uint16_t msgCount = 0;
```

```
static uint32_t txInterval_ms = TX_LAPSE_MS;
static uint32_t tx_begin_ms = 0;

if (!transmitting && ((millis() - lastSendTime_ms) > txInterval_ms)) {

  uint8_t payload[50];
  uint8_t payloadLength = 0;

  payload[payloadLength]    = (thisNodeConf.bandwidth_index << 4);
  payload[payloadLength++] |= ((thisNodeConf.spreadingFactor - 6) << 1);
  payload[payloadLength]    = ((thisNodeConf.codingRate - 5) << 6);
  payload[payloadLength++] |= ((thisNodeConf.txPower - 2) << 1);

  // Incluimos el RSSI y el SNR del último paquete recibido
  // RSSI puede estar en un rango de [0, -127] dBm
  payload[payloadLength++] = uint8_t(-LoRa.packetRssi() * 2);
  // SNR puede estar en un rango de [20, -148] dBm
  payload[payloadLength++] = uint8_t(148 + LoRa.packetSnr());

  transmitting = true;
  txDoneFlag = false;
  tx_begin_ms = millis();

  sendMessage(payload, payloadLength, msgCount);
  Serial.print("Sending packet ");
  Serial.print(msgCount++);
  Serial.print(": ");
  printBinaryPayload(payload, payloadLength);
}

if (transmitting && txDoneFlag) {
  uint32_t TxTime_ms = millis() - tx_begin_ms;
  Serial.print("----> TX completed in ");
  Serial.print(TxTime_ms);
  Serial.println(" msecs");

  // Ajustamos txInterval_ms para respetar un duty cycle del 1%
  uint32_t lapse_ms = tx_begin_ms - lastSendTime_ms;
  lastSendTime_ms = tx_begin_ms;
  float duty_cycle = (100.0f * TxTime_ms) / lapse_ms;

  Serial.print("Duty cycle: ");
  Serial.print(duty_cycle, 1);
  Serial.println(" %\n");
```

```
    // Solo si el ciclo de trabajo es superior al 1% lo ajustamos
    if (duty_cycle > 1.0f) {
      txInterval_ms = TxTime_ms * 100;
    }

    transmitting = false;

    // Reactivamos la recepción de mensajes, que se desactiva
    // en segundo plano mientras se transmite
    LoRa.receive();
  }
}

// ----------------------------------------------------------------------
// Sending message function
// ----------------------------------------------------------------------
void sendMessage(uint8_t* payload, uint8_t payloadLength, uint16_t msgCount)
{
  while(!LoRa.beginPacket()) {          // Comenzamos el empaquetado del mensaje
    delay(10);
  }
  LoRa.write(destination);              // Añadimos el ID del destinatario
  LoRa.write(localAddress);             // Añadimos el ID del remitente
  LoRa.write((uint8_t)(msgCount >> 7)); // Añadimos el Id del mensaje (MSB primero)
  LoRa.write((uint8_t)(msgCount & 0xFF));
  LoRa.write(payloadLength);          // Añadimos la longitud en bytes del mensaje
  LoRa.write(payload, (size_t)payloadLength); // Añadimos el mensaje/payload
  LoRa.endPacket(true);               // Finalizamos el paquete, pero no esperamos
                                      // a que finalice su transmisión
}

// ----------------------------------------------------------------------
// Receiving message function
// ----------------------------------------------------------------------
void onReceive(int packetSize)
{
  // Esto no debería ocurrir, pero ocurre. Parece que la librería
  // LoRa tiene un error (tipo "race condition") que hace que la
  // interrupción asociada a la finalización de una transmisión
  // TxDone no se emite en ocasiones.
  if (transmitting && !txDoneFlag) txDoneFlag = true;

  if (packetSize == 0) return;          // Si no hay mensajes, retornamos

  // Leemos los primeros bytes del mensaje
```

```
uint8_t buffer[10];                    // Buffer para almacenar el mensaje
int recipient = LoRa.read();           // Dirección del destinatario
uint8_t sender = LoRa.read();          // Dirección del remitente
                                       // msg ID (High Byte first)
uint16_t incomingMsgId = ((uint16_t)LoRa.read() << 7) |
                          (uint16_t)LoRa.read();

uint8_t incomingLength = LoRa.read(); // Longitud en bytes del mensaje

uint8_t receivedBytes = 0;             // Leemos el mensaje byte a byte
while (LoRa.available() && (receivedBytes < uint8_t(sizeof(buffer)-1))) {
  buffer[receivedBytes++] = (char)LoRa.read();
}

if (incomingLength != receivedBytes) {// Verificamos la longitud del mensaje
  Serial.print("Receiving error: declared message length " +
               String(incomingLength));
  Serial.println(" does not match length " + String(receivedBytes));
  return;
}

// Verificamos si se trata de un mensaje en broadcast o es un mensaje
// dirigido específicamente a este dispositivo.
// Nótese que este mecanismo es complementario al uso de la misma
// SyncWord y solo tiene sentido si hay más de dos receptores activos
// compartiendo la misma palabra de sincronización
if ((recipient & localAddress) != localAddress ) {
  Serial.println("Receiving error: This message is not for me.");
  return;
}

// Imprimimos los detalles del mensaje recibido
Serial.println("Received from: 0x" + String(sender, HEX));
Serial.println("Sent to: 0x" + String(recipient, HEX));
Serial.println("Message ID: " + String(incomingMsgId));
Serial.println("Payload length: " + String(incomingLength));
Serial.print("Payload: ");
printBinaryPayload(buffer, receivedBytes);
Serial.print("\nRSSI: " + String(LoRa.packetRssi()));
Serial.print(" dBm\nSNR: " + String(LoRa.packetSnr()));
Serial.println(" dB");

// Actualizamos remoteNodeConf y lo mostramos
if (receivedBytes == 4) {
  remoteNodeConf.bandwidth_index = buffer[0] >> 4;
```

```
    remoteNodeConf.spreadingFactor = 6 + ((buffer[0] & 0x0F) >> 1);
    remoteNodeConf.codingRate = 5 + (buffer[1] >> 6);
    remoteNodeConf.txPower = 2 + ((buffer[1] & 0x3F) >> 1);
    remoteRSSI = -int(buffer[2]) / 2.0f;
    remoteSNR  =  int(buffer[3]) - 148;

    Serial.print("Remote config: BW: ");
    Serial.print(bandwidth_kHz[remoteNodeConf.bandwidth_index]);
    Serial.print(" kHz, SPF: ");
    Serial.print(remoteNodeConf.spreadingFactor);
    Serial.print(", CR: ");
    Serial.print(remoteNodeConf.codingRate);
    Serial.print(", TxPwr: ");
    Serial.print(remoteNodeConf.txPower);
    Serial.print(" dBm, RSSI: ");
    Serial.print(remoteRSSI);
    Serial.print(" dBm, SNR: ");
    Serial.print(remoteSNR,1);
    Serial.println(" dB\n");
  }
  else {
    Serial.print("Unexpected payload size: ");
    Serial.print(receivedBytes);
    Serial.println(" bytes\n");
  }
}

void TxFinished()
{
  txDoneFlag = true;
}

void printBinaryPayload(uint8_t * payload, uint8_t payloadLength)
{
  for (int i = 0; i < payloadLength; i++) {
    Serial.print((payload[i] & 0xF0) >> 4, HEX);
    Serial.print(payload[i] & 0x0F, HEX);
    Serial.print(" ");
  }
}
```

Ejercicio propuesto

El ejemplo anterior puede servir de base para plantear un esquema que permita optimizar, entiéndase minimizar, los tiempos de transmisión entre dos nodos de una red privada LoRa. La idea es ir ajustando progresivamente los parámetros de configuración de las radios con el objetivo de reducir el tiempo de transmisión, pero garantizando que el SNR y RSSI de los paquetes recibidos se mantiene por encima de ciertos umbrales.

Nótese que será necesario definir un pequeño protocolo de comunicación entre los nodos para poder coordinar los ajustes en la configuración de las radios. Uno de los nodos, el nodo maestro, deberá llevar la iniciativa a la hora de activar cambios en la configuración. Estos cambios deberán notificarse al otro nodo, o nodo esclavo, antes de cambiar efectivamente la configuración de cualquiera de las radios, para que los nodos puedan seguir comunicando por radio.

Nota

Se ha detectado un problema cuando se usan interrupciones para notificar el final de una transmisión. El síntoma es que la transmisión nunca se confirma porque el callback asociado a TxDone nunca se llega a invocar. A partir de ese momento, el nodo no enviará ni recibirá paquetes.

Ver https://github.com/sandeepmistry/arduino-LoRa/issues/400

Una posible medida de continuidad es no usar transmisiones asíncronas, que – evidentemente – no es la solución óptima, en particular cuando la transmisión tenga una duración importante.

Referencias

[1] Información general sobre el transceiver de radio Semtech SX1276

[2] Manual del módulo de radio Semtech SX1276

[3] Librería Arduino LoRa

[4] Librería Arduino BQ24195

[5] Reducción del payload en el portal de LoRa.Developers

[6] Librería Cayenne LPP