MODIFICACIONES LIBRERÍA DW1000

Adaptaciones y extensiones surgidas a raíz de la realización de un ejemplo de centralización de los datos en un anchor maestro.

Jaime Pérez Pérez, 20259

ÍNDICE

Introducción

En este documento se explicarán los métodos y funciones añadidos a la librería DW1000: Los objetivos logrados tras estas modificaciones son:

1. Solicitar a otros dispositivos que cambien su modo de funcionamiento
2. Solicitar a otros dispositivos que envíen sus medidas registradas
3. Envíar dichas medidas cuando sea solicitado por un dispositivo “maestro”.

Además, se ven las principales funciones desarrolladas en el sketch que se cargan en las placas utilizadas.

Modificaciones previas

Cambio en la nomenclatura de los tipos de dispositivos

Para poner en contexto las modificaciones previas realizadas, puramente de sintaxis, hay que entender el protocolo de comunicación entre dispositivos.
Las placas se comunican vía UWB utilizando el procedimiento conocido como Two Way Ranging. En este proceso, una de las placas es la encargada de “buscar” a la otra. Envía mensajes de *polling* hasta que la placa encargada de responder le contesta.
Esta primera comunicación ya es parte del TWR. Dentro de esos mensajes que se envían, se encuentran codificados los instantes de tiempo en los que fueron recibidos los anteriores, y enviados estos. De esta manera, cuando el dispositivo que ha iniciado la conversación le envía el segundo mensaje al que contesta, éste segundo ya puede conocer la distancia entre ellos.
Para hacerlo, utiliza los instantes de tiempo, o *timestamps*, que se han ido enviando en la conversación. Los usa para calcular cuánto tiempo han estado esos mensajes en el aire. Sabiendo que se desplazan a la velocidad de la luz, se puede calcular directamente la distancia entre las dos placas.

Conocido este contexto, ahora se explican las modificaciones de nomenclatura realizadas en la librería.
En la versión anterior, se asumía que los dispositivos móviles, o *Tags* eran los encargados de iniciar la conversación, mientras que los fijos, o *Anchors*, eran los encargados de responder.
Siguiendo esa suposición, se iniciaban los dispositivos como *Anchors* o como *tags* en función del comportamiento deseado.

El código existente era el siguiente:

```
void DW1000RangingClass::startAsTag(char address[], const byte mode[], const
bool randomShortAddress) {

void DW1000RangingClass::startAsAnchor(char address[], const byte mode[],
const bool randomShortAddress) {
```

Durante el desarrollo del código de centralización de los datos, surgía el inconveniente de que el ancla “maestra” tuviera que solicitar a las anclas “esclavas” que cambiaran su modo de funcionamiento y que le enviaran sus datos. Es decir, se necesitaba que un ancla fuera la encargada de iniciar la comunicación.

Para solventarlo, y evitar confusiones, se ha desasociado el comportamiento de iniciar la comunicación con los *tags*, y el de responder con los *Anchors.* Para hacerlo, se ha sustituido la nomenclatura de los métodos para iniciar los dispositivos

```
void DW1000RangingClass::startAsInitiator(char address[], const byte mode[],
const bool randomShortAddress) {

void DW1000RangingClass::startAsInitiator(char address[], const byte mode[],
const bool randomShortAddress) {
```

Para hacerlo, además, se modificaron los tipos de dispositivos declarados en la librería

```
#define TAG 0
#define ANCHOR 1
```

```
#define INITIATOR 0
#define RESPONDER 1
```

Evitado limpiado de dispositivos al reconocer otro

En la versión anterior, no se esperaba que existieran más de un tag (posteriormente llamado “responder”) en el sistema. Este comportamiento inestabilizaba el sistema, eliminando todos los dispositivos cada vez que se añadía uno nuevo a cualquier iniciador.

```
if(addDevice) {
        if(_type == ANCHOR) //for now let's start with 1 TAG
        {
            _networkDevicesNumber = 0;
        }
```

Esta sección de código, ubicada dentro del método *addNetworkDevices*, ubicado en el archivo DW1000Ranging.cpp fue eliminado.

Envío de los datos a través de UWB

A la hora de enviar datos vía UWB entre los dispositivos, el procedimiento es siempre el mismo. Se prepara el mensaje a enviar, incluyendo todas las partes necesarias (cabecera MAC - tipo de mensaje - Carga del mensaje), y se envía utilizando el método transmit.
A este método se le envía el buffer "*data".* Pero en la versión anterior, enviaba una versión desactualizada del mismo: enviaba "datas".

Se corrigió este error, enviando la última versión del buffer generado para cada comunicación entre dispositivos.

```
void DW1000RangingClass::transmit(byte datas[]) {
    DW1000.setData(data, LEN_DATA);
    // Versión anterior de la librería -> DW1000.setData(datas, LEN_DATA);
    DW1000.startTransmit();
}
```

Nueva versión de la librería. Código aportado

Motivación

La actualización de la librería surge como consecuencia de tratar de lograr la centralización de los datos en un solo dispositivo.
Para ello, el comportamiento deseado era el siguiente:
Hasta ahora, se tenía acceso a las distancias medidas entre iniciadores y respondedores, cada uno mostrando sus mediciones realizadas.
El objetivo buscado era que un dispositivo maestro supiera la distancia existente entre todos los dispositivos del sistema. Al menos, la distancia entre todas las anclas y los tags.

Las mediciones solo se pueden realizar desde un iniciador a un respondedor. El ancla maestra actuará en todo momento como un iniciador. Mide su distancia hacia todos los dispositivos (que actúan como respondedores) y guarda los datos.
Posteriormente, deberá pedirle a las anclas esclavas del sistema que cambien de modo. Pasarán a actuar como iniciadores. De esta manera, podrán realizar sus mediciones con los tags.

Periódicamente, el ancla maestra pedirá a las esclavas que le manden estas mediciones almancenadas

Tipos de mensajes

Considerando el funcionamiento buscado, se han añadido 3 tipos de mensajes reconocidos en la librería:

Declaraciones en el encabezado. Archivo DW1000Ranging.h

```
/ Nuevos mensajes: control del flujo de operación:
#define MODE_SWITCH 6  // Para solicitar un cambio de modo. De iniciador a responder o viceversa. Este mensaje es enviado por el maestro
#define REQUEST_DATA 7 // Enviado por el maestro. Le pide a los esclavos que le envíen los datos que hayan guardado (estos datos incluyen la distancia entre dicho esclavo con el resto de dispositivos respondedores)
#define DATA_REPORT 8 // Enviado por el esclavo de vuelta al maestro. Le envía los datos que le han sido solicitados
```

Almacenado de datos

La parte más importante de esta funcionalidad buscada es precisamente, el guardado y enviado de los datos. Para hacerlo, se ha definido una estructura en la que se almacenan los datos utilizados:

Declaración del struct en el encabezado (DW1000Ranging.h)

```
// Struct creado para guardar los datos de cada medición:
struct Measurement {
    uint16_t short_addr_origin;
    uint16_t short_addr_dest;
    float distance;     // Última distancia medida (en metros)
    float rxPower;      // Última potencia medida (en dBm)
    bool active;        // Almacena si el dispositivo de destino está activo
};
```

Envío de los mensajes nuevos

Todos los mensajes se realizan por UWB. Para hacerlo, utilizamos el método transmit() previamente existente en la librería. La única modificación necesaria es el contenido a transmitir.

Independientemente del tipo de mensaje, la estructura de las transmisiones siempre es la misma:

Dirección MAC: ***Corta***	***Larga***	***Tipo de mensaje***	***Payload***

```
#define SHORT_MAC_LEN 9
#define LONG_MAC_LEN 15
```

```
#define POLL 0
#define POLL_ACK 1
#define RANGE 2
#define RANGE_REPORT 3
#define RANGE_FAILED 255
#define BLINK 4
#define RANGING_INIT 5

#define MODE_SWITCH 6 //To
#define REQUEST_DATA 7 //
#define DATA_REPORT 8 //
```

```
#define LEN_DATA 90
```

Por simplicidad, los 3 tipos de transmisiones nuevas llevarán una cabecera creada con una dirección Mac corta.

Cambio de modo

Declaración del método en el .h

```
// Para solicitar un cambio en el modo de operación
void transmitModeSwitch(bool toInitiator, DW1000Device* device = nullptr);
```

El método recibirá dos parámetros:

1. Un booleano que controla si la solicitud es para cambiar a modo iniciador (si el valor es 'False', el cambio se realiza a "responder)
2. Objeto de tipo dispositivo. Indica a quién le pide el cambio de modo.

El parámetro de tipo *device* tiene un valor nulo por defecto. Si no se adjunta nada, valdrá *nullptr*, lo que significa que el mensaje se enviará via broadcast (a todos los dispositivos que estén escuchando). Esto se explica dentro del propio código.

Implementación del método en el .cpp

```
void DW1000RangingClass::transmitModeSwitch(bool toInitiator, DW1000Device*
device){

    //1: se prepara la placa para una nueva transmisión.
    transmitInit(); // Resetea el ack flag y reestablece parámetros por
defecto (power, data rate, preámbulo, etc).

    byte dest[2]; //Aquí guardaré la dirección del destinatario

    //2: Seleccionar destinatario: Unicast vs Broadcast
    if (device == nullptr){
        // Si no se manda un device, se hace Broadcasting
        // El mensaje se envía a todos los que estén escuchando
        dest[0] = 0xFF;
        dest[1] = 0xFF;
        //Según el estándar del IEEE, la shortAddress 0xFF 0xFF está reservada
para broadcast. Todos los receptores recibirán el mensaje.
    }
    else{
        //Si sí que se pasa un device, se envía solo a ese.
        // Memcpy -> Copia en el destino, desde el origen, el número de bytes
        memcpy(dest,device->getByteShortAddress(),2);
   }
    //3: Generar la cabecera del mensaje
```

```
    _globalMac.generateShortMACFrame(data, _currentShortAddress, dest);

    //4: Insertar el payload (el cuerpo del mensaje)
    /* Byte #0 = tipo del mensaje enviado: MODE_SWITCH
       Byte #1 -> Indica si se solicita que cambie a “initiator” o a
      “responder”   */

    data[SHORT_MAC_LEN] = MODE_SWITCH;
    data[SHORT_MAC_LEN+1] = toInitiator ? 1:0;

    transmit(data); // el buffer de datos (que se ha ido generando), se envía
por UWB
}
```

Solicitud de datos

Declaración del método en el .h:

```
void transmitDataRequest(DW1000Device* device = nullptr);
```

Al igual que antes, el parámetro *device* tiene un valor nulo por defecto.

Implementación del método en el .cpp:

```
void DW1000RangingClass::transmitDataRequest(DW1000Device* device){

    //El método funciona igual que el transmitModeSwitch. Consultar ese para
cualquier duda.

    transmitInit();
    byte dest[2];

    if (device == nullptr){
        dest[0] = 0xFF;
        dest[1] = 0xFF;
    }
    else{
        memcpy(dest,device->getByteShortAddress(),2);
    }

    _globalMac.generateShortMACFrame(data, _currentShortAddress, dest);
    data[SHORT_MAC_LEN] = REQUEST_DATA;

    transmit(data); //the data is sent via UWB
}
```

Envío de datos

Una vez el esclavo ha recibido la petición de envío de los datos, deberá enviárselos al maestro. El funcionamiento de este método es algo distinto:

Declaración en el encabezado (DW1000Ranging.h)

```
void transmitDataReport(Measurement* measurements, int numMedidas,
DW1000Device* device = nullptr);
```

Este recibirá 3 parámetros:

1. Un puntero a un struct del tipo Measurement (medidas). De esta manera, el método tendrá acceso a los datos guardados por el esclavo, para poder codificarlos y enviárselos al maestro

2. Número de medidas que se van a enviar. Cada medida incluirá los dispositivos que se comunican, la distancia entre ellos, y la potencia RX medida en esa comunicación.
   Este parámetro es útil tanto para el esclavo que está enviando los datos, como para el maestro que los recibirá.
   Su uso está explicado dentro del código. Sirve para comprobar que la longitud de los datos a enviar sea adecuada, y para recorrer el struct de medidas tantas veces como medidas haya.

3. De nuevo, un parámetro de tipo *device*, en el que se indica el dispositivo al que se va a enviar el data report.

Implementación del método en el .cpp:

```
void DW1000RangingClass::transmitDataReport(Measurement* measurements, int
numMeasures, DW1000Device* device) {

    // Primero, se obtiene la dirección de destino de igual manera que en los
métodos anteriores.
    byte dest[2];
    if (device == nullptr) {
        dest[0] = 0xFF;
        dest[1] = 0xFF;
    }
    else {
        memcpy(dest, device->getByteShortAddress(), 2);
    }
```

```
    transmitInit(); // Inicio una nueva transmisión.

    //Genero la dirección Mac. Elijo hacerlo en formato corto.
    _globalMac.generateShortMACFrame(data, _currentShortAddress, dest);

    // El siguiente byte lleva codificado el tipo de mensaje.
    data[SHORT_MAC_LEN] = DATA_REPORT;

    // La variable index me ayuda a recorrer el cuerpo del mensaje
    uint8_t index = SHORT_MAC_LEN + 1;

    // Los siguientes 2 bytes llevan la dirección del dispositivo desde el que
se envía el data report.
    data[index++] = _currentShortAddress[0];
    data[index++] = _currentShortAddress[1];

    // 1 byte más para el numero de medidas que se van a enviar.
    data[index++] = numMeasures;

    // Antes de enviar, compruebo si hay espacio suficiente dentro
    del buffer data:

    size_t totalPayloadSize = 3 + numMeasures * 10;
    // cada medida ocupa 10 bytes, y hay 3 más para el shortAddress y el
messagetype
    size_t totalMessageSize = SHORT_MAC_LEN + 1 + totalPayloadSize;
    // Y el tamaño total es la longitud de la cabecera + la del payload

    if (totalMessageSize > LEN_DATA) {
        if (DEBUG) {
            Serial.println("Error: DATA_REPORT exceeds the size of the data[]
buffer");
        }
        return; //Si no hay espacio, no se envía el mensaje.
    }
```

```
    // Ahora codifico el cuerpo del mensaje a enviar:
    // Se necesitan 10 bytes por medida:
    // 2 para el shortAddress del dispositivo con el que se ha medido
    // 4 para el float que guarda la distancia medida
    // 4 para la medida del RX power

    for (uint8_t i = 0; i < numMeasures; i++) {
      // Recorro data introduciendo todos estos datos en las posiciones
adecuadas.
        memcpy(data + index, &measurements[i].short_addr_dest, 2); index += 2;
        memcpy(data + index, &measurements[i].distance, 4); index += 4;
        memcpy(data + index, &measurements[i].rxPower, 4);   index += 4;
    }

    transmit(data); // Y por último, envío el mensaje.
}
```

Recepción de los mensajes nuevos

La segunda mitad del proceso es la recepción del mensaje. Una vez se recibe y se lee de qué tipo de mensaje se trata, el código “lanza” una acción u otra utilizando unos callbacks.

Callbacks

La declaración de los callbacks se realiza en el documento de cabecera (*DW1000Ranging.h).* La declaración muestra el tipo de función que reciben por parámetro, indicando los parámetros que estas esperan recibir.

En el archivo de código que se sube a las placas, se enlazan los callbacks a funciones definidas dentro del sketch.
En el mismo archivo de la librería, se define una variable de tipo static void que apuntará a la función enlazada al callback. De esta manera, posteriormente, en la implementación en el *.cpp*, se emplearán esas variables para lanzar las funciones enlazadas.

Cambio de modo

1: Declaro la variable que apuntará a la función dentro de la parte privada

```
static void (* _handleModeChangeRequest)(bool toInitiator);
```

2: Declaro el método en la parte pública de la DW1000RangingClass

```
    //Callback para el mode change. Llama a una función que recibe un booleano como parámetro.
    static void attachModeChangeRequest(void (* handleModeChange)(bool toInitiator)){ _handleModeChangeRequest = handleModeChange;}
```

Solicitud de datos

```
static void (* _handleDataRequest)(byte* shortAddress);
```

```
static void attachDataRequest(void (*handleDataRequest)(byte* shortAddress)){ _handleDataRequest = handleDataRequest; }
```

Envío de los datos

```
static void (* _handleDataReport)(byte* dataReport);
```

```
static void attachDataReport(void (*handleDataReport)(byte* dataReport)){
_handleDataReport = handleDataReport;}
```

La función enlazada a este callback recibirá un parámetro de tipo byte, en el que estarán codificadas las medidas enviadas.

Llamada a los callbacks: DW1000Ranging Loop

Dentro del loop de la librería, se comprueba si se han enviado o recibido mensajes. En función del tipo recibido/enviado, se ejecutan unas acciones u otras

Tipo de mensaje

El primer paso es comprobar qué tipo de mensaje es el que se recibe.
Para hacerlo, primero se comprueba si la longitud MAC recibida es larga o corta. En función de eso, extrae de la posición adecuada el código del mensaje recibido (devuelve el valor del tipo de mensaje declarado en el archivo de cabecera)

```
int16_t DW1000RangingClass::detectMessageType(byte datas[]) {
    if(datas[0] == FC_1_BLINK) {
        return BLINK;
    }
    else if(datas[0] == FC_1 && datas[1] == FC_2) {
        // Direccion MAC larga
        _lastFrameWasLong = true;
        return datas[LONG_MAC_LEN];
    }
    else if(datas[0] == FC_1 && datas[1] == FC_2_SHORT) {
        //Dirección MAC corta (poll, range, range report, etc..)
        _lastFrameWasLong = false;
        return datas[SHORT_MAC_LEN];
    }
    return -1; // Valor por defecto para evitar errores
```

Tras obtener el tipo de mensaje, el código lanza un callback u otro:

Cambio de modo

```
if(messageType == MODE_SWITCH){ //mensaje recibido de tipo mode switch
            int headerLen = _lastFrameWasLong ? LONG_MAC_LEN : SHORT_MAC_LEN;
            //Compruebo la longitud del mensaje y guardo la correcta
            bool toInitiator = (data[headerLen + 1] == 1);
            //extraigo el booleano de la siguiente posición de la cabecera
            if (_handleModeChangeRequest) {
               //y llamo a la función enlazada al callback pasándoselo
                (*_handleModeChangeRequest)(toInitiator);
            }
```

Solicitud de datos

```
void (* DW1000RangingClass::_handleDataRequest)(byte*) = 0;
```

```
else if(messageType == REQUEST_DATA){

            byte shortAddress[2]; // Para guardar el shortAddress
            _globalMac.decodeShortMACFrame(data, shortAddress); //Extrae el
shortAddress del data recibido.

            if(_handleDataRequest){
                // Llama al callback enlazado pasándole el shortAddress del
dispositivo que ha enviado la solicitud
                (* _handleDataRequest)(shortAddress);
            }
            return;
        }
```

Envío de los datos

```
else if(messageType == DATA_REPORT){
           // El maestro le ha pedido al esclavo que envíe sus datos
           if(_handleDataReport){
                (* _handleDataReport)(data);
            }
        }
```

Ejemplo completo. Código de las placas

Una vez comprendido el funcionamiento de la librería, el siguiente paso es comprender el código subido a las placas.

Almacenado de los datos

Cuando salta el callback asociado al final de una medición (*newRange,* previamente definido en la biblioteca), se llama a la función logMeasure.

```
void newRange(){
    uint16_t dest_sa = DW1000Ranging.getDistantDevice()->getShortAddress();
    float dist = DW1000Ranging.getDistantDevice()->getRange();
    float rx_pwr = DW1000Ranging.getDistantDevice()->getRXPower();
    logMeasure(own_short_addr,dest_sa, dist, rx_pwr);
}
```

```
void logMeasure(uint16_t own_sa,uint16_t dest_sa, float dist, float rx_pwr){

    // Primero compruebo si esa medida ya está registrada
    int index = searchDevice(own_sa,dest_sa);
    if(dist < 0){ dist = -dist;}
    if (index != -1){ // esto significa: se ha encontrado

        // Solo actualiza la distancia y la potencia
        measurements[index].distance = dist;
        measurements[index].rxPower = rx_pwr;
        measurements[index].active = true;

    }
    else if (amountDevices < MAX_DEVICES){

        // Si no se ha encontrado, declara una nueva entrada al struct.
        measurements[amountDevices].short_addr_origin = own_sa;
        measurements[amountDevices].short_addr_dest = dest_sa;
        measurements[amountDevices].distance = dist;
        measurements[amountDevices].rxPower = rx_pwr;
        measurements[amountDevices].active = true;
        amountDevices ++; // Y aumenta el nº de dispositivos en 1

    }
```

Solicitud de cambio

Periódicamente, el ancla maestra envía una solicitud de datos a los esclavos

```
void loop(){

    DW1000Ranging.loop();
    current_time = millis();

        if (IS_MASTER){ // Solo hace la llamada el maestro

            if(!report_pending && (current_time - last_switch >= switch_time)){
                // La solicitud se hace si no está esperando un data report
                // Y si ha pasado el tiempo suficiente
                last_switch = current_time;
                slaveIsInitiator = !slaveIsInitiator;
                // Cambia el estado, para pedirle que cambie al correcto
                Serial.print("CAMBIO A ");
                Serial.println(slaveIsInitiator ? "INITIATOR" : "RESPONDER");

                DW1000Ranging.transmitModeSwitch(slaveIsInitiator);
```

Recepción de solicitud de cambio

Esto solo le llega a los dispositivos esclavos, puesto que en el setup, solo éstos tienen enlazado el callback correspondiente.

```
void ModeChangeRequest(bool toInitiator){

    if(toInitiator == true){

        DW1000.idle();
        DW1000Ranging.startAsInitiator(DEVICE_ADDR,DW1000.MODE_LONGDATA_RANGE_LOWPOWER,
false);
       }
    else{

        DW1000.idle();
        DW1000Ranging.startAsResponder(DEVICE_ADDR,DW1000.MODE_LONGDATA_RANGE_LOWPOWER,
false);
      }
}
```

Solicitud de los datos

Y, también dentro del loop, pide periódicamente el data report a los dispositivos esclavos

```
if(!slaveIsInitiator && (current_time - last_report >= report_time)){

                Serial.println("DATA REQUEST ENVIADO");
                DW1000Ranging.transmitDataRequest();
                // No le paso un dispositivo → Hace Broadcast
                report_pending = true;
                last_report = current_time;

            }
```

Recepción de solicitud de datos

```
void DataRequest(byte* short_addr_requester){

    // Llamada cuando el maestro pide un data report
   uint16_t numMeasures = amountDevices;

    DW1000Device* requester =
DW1000Ranging.searchDistantDevice(short_addr_requester);
    // Busco al dispositivo que lo ha solicitado

    if(!requester){
        //si no lo encuentra, envía los datos por broadcast

DW1000Ranging.transmitDataReport((Measurement*)measurements,numMeasures,nullpt
r);
        return;
    }
    //Si sí lo encuentra, los envía por unicast

DW1000Ranging.transmitDataReport((Measurement*)measurements,numMeasures,reques
ter);
    // Le pasa el struct de Measurements que tiene almacenado
}
```

Decodificación del data report

Una vez el maestro recibe los datos, tiene que descodificar el paquete recibido

```
void DataReport(byte* data){

    Serial.println("DATA REPORT RECIBIDO");
    uint16_t index = SHORT_MAC_LEN + 1;
    uint16_t origin_short_addr = ((uint16_t)data[index] << 8) | data[index +
1];
    index += 2;

    uint16_t numMeasures = data[index++];

    // Primero comprueba si la longitud es válida
    if(numMeasures*10>LEN_DATA-SHORT_MAC_LEN-4){

        Serial.println("The Data received is too long");
        return;
    }

    for (int i = 0; i < numMeasures; i++) {

        uint16_t destiny_short_addr = ((uint16_t)data[index] << 8) |
data[index + 1];
        index += 2;

        float distance, rxPower;
        memcpy(&distance, data + index, 4); index += 4;
        memcpy(&rxPower,   data + index, 4); index += 4;

        logMeasure(origin_short_addr, destiny_short_addr, distance, rxPower);
    }
    showData();
    report_pending = false;
}
```