APROXIMACIÓN METODOLÓGICA CON PROPÓSITO ACADÉMICO PARA EL DESARROLLO DE DISPOSITIVOS WEARABLE PARA EXTREMIDADES INFERIORES DEL CUERPO HUMANO

ANEXOS

YESID FELIPE TOMBÉ CASTILLO
JUAN ESTEBAN BEDOYA RAMÍREZ

Director: MSc. Marlon Felipe Burbano Fernández
Codirector: PhD. Gustavo Adolfo Ramírez González

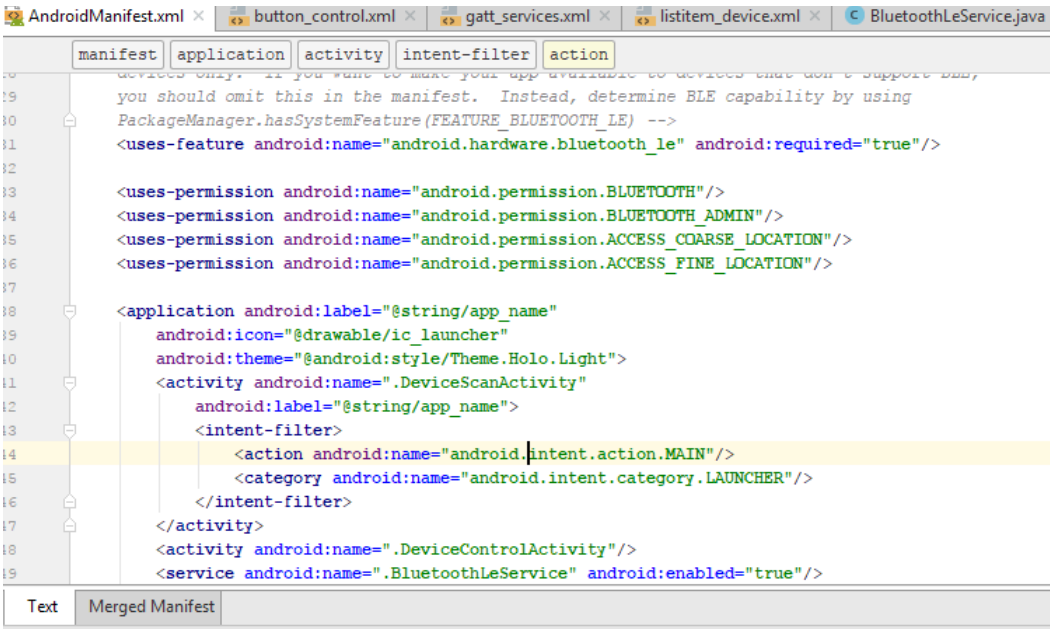
Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Telemática
Popayán, Mayo de 2018

1. ANEXO A

1.1. Documentación de la aplicación Android desarrollada en el módulo primario wearable

La documentación de Android Studio junto con sus códigos proveen una excelente ayuda para poder utilizar las librerías BLE. Se comienza utilizando varios códigos ejemplos encontrados en internet, los cuales no se pudieron implementar, en la mayoría de los casos se encontraban con errores de versiones del software, y cuando se lograba compilar, el escaneo de dispositivos en advertising no desplegaba ningún resultado. Finalmente, con los tutoriales directos de Android, al importar *"BLUETOOTH_LE_GATT_CHARACTERISTICS"* se avanzó rápidamente, mientras el error persistía, de no poder encontrar ningún dispositivo.

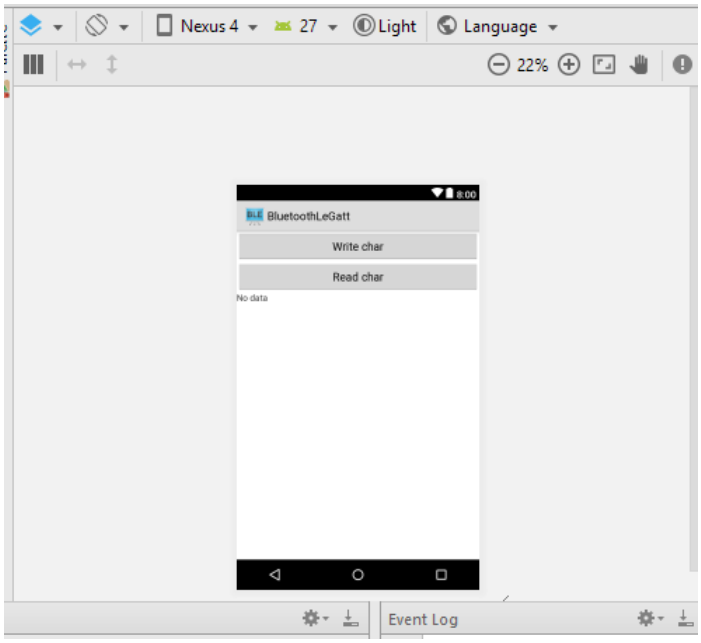
Al revisar la documentación, se encontró que si se requiere utilizar dispositivos BLE en Android es necesario el uso de permisos dentro del manifest.xml



Estos permisos se piden dentro del androidmanifest.xml y cuando se inicie la aplicación serán requeridos para que esta pueda correr. Los permisos tipo BLE, son indispensables para poder realizar cualquier tipo de pruebas, mientras los de localización solo se necesitan cuando se utilizan filtros para conectarse a servicios determinados, es necesario reiterar que al iniciar la aplicación, si no se han dado permisos BLE, la aplicación

fallará, mientras que los permisos de localización no generarán esta reacción. Además, el hecho de no haber sido permitidos producirán que no se observe ningún elemento escaneado, si se quieren dar permisos de localización se debe ir a esta dirección: *SETTINGS/APPLICATIONS/APPLICATIONS_MANAGER/(TU_APLICACION)/PERMISSONS* una vez en este punto se activa la localización.

Al haber completado los pasos anteriores fue posible empezar a escanear dispositivos BLE, y siguiendo los tutoriales, se creó una página de lectura y escritura de características vista en el siguiente cuadro:



Aún al haber realizado el tutorial los diferentes errores que se encuentran fueron debidos a la mala escritura de las funciones, utilizadas cuando se presionan los botones.

Para realizar pruebas de la aplicación se hace uso de la aplicación *BLE TOOLS*, la cual mediante el uso de un segundo dispositivo móvil, permitió la emulación de un instrumento BLE, con servicios y tres características, la primera da opciones de lectura, la segunda de lectura y escritura y la tercera permite la lectura con notificación. Es necesario explicar que cada característica posee descriptores, los cuales solo en CLIENT_CONFIG_MANAG, da opciones de notificación, al utilizar esta ID, en las funciones de Android. Así, para habilitar la notificación se logró la lectura de dispositivos con notificación.

Algunas falencias de los tutoriales encontrados, como por ejemplo, el que se siguió daba un código incompleto y solo ofrecía opciones de validación para cada una de las opciones, en estas, si la UUID del servicio no se encontraba dentro del dispositivo encontrado, siempre se devolvía a la primera imagen puesta en este documento, mientras que si se encontraba,

tampoco realiza ninguna acción notable, excepto si se utiliza el debugger de Android, que permitirá revisar el estado de la máquina con las banderas puestas en el código de la manera: *Log.w(TAG, ,msg: "aquí voy")*. Esto se debe a que, aunque las funciones de lectura están implementadas dentro del código, no se encuentra resolviendo ninguna función, simplemente están en reposo, y en las funciones creadas para leer, solo se obtiene la referencia del servicio y de la característica deseada.

1.1.1. 1. BluetoothLeService.java

Esta clase java es proporcionada en los ejemplos de Android sobre BLE, con mínimas modificaciones puede ser implementada para realizar todas las funciones de *CALLBACK* de BLE. El BLE es explicado en la sección de BLE, este código solo se explican las zonas más importantes para recibir la comunicaciones del *GATT SERVER*.

Para poder leer es necesario llamar dentro de la función a *"readCharacteristic (mcharacteristic)"* e ingresar la referencia encontrada en la función:

```
public void readCustomCharacteristic(){
    Log.w(TAG,  msg: "start");


    if(mBluetoothAdapter == null || mBluetoothGatt == null){
        Log.w(TAG,  msg: "Bluetooth Adapter Not Initialized");
        return;

    }
    /*check if the service is available on the device*/
    BluetoothGattService mCustomService = mBluetoothGatt.getService(UUID.fromString("0000fff0-0000-1000-8000-00805f9b34fb"));
    if (mCustomService == null){
        Log.w(TAG,  msg: "Custom BLE service not found");
        return;
    }
    /*check if the chacateristic is available on the device*/
    BluetoothGattCharacteristic mReadCharacteristic = mCustomService.getCharacteristic(UUID.fromString("0000fff4-0000-1000-8000-00805f9b34fb"))
    if ( mReadCharacteristic == null){
        Log.w(TAG,  msg: "Custom Characteristic Not found");
        return;
    }
    Log.w(TAG,  msg: "Change ReadChar");
   readCharacteristic(mReadCharacteristic);
```

Esto nos mandara a la función readCharacteristic.

```
     * @param characteristic The characteristic to read from.
     */
    public void readCharacteristic(BluetoothGattCharacteristic characteristic) {
        if (mBluetoothAdapter == null || mBluetoothGatt == null) {
            Log.w(TAG,  msg: "BluetoothAdapter not initialized");
            return;
        }
        mBluetoothGatt.readCharacteristic(characteristic);

    }
```

En donde realmente se llamará la función necesaria, cabe mencionar que Android funciona de manera asíncrona, de tal manera no se sigue un orden secuencial en las funciones, todas las funciones de nombre onXXXX van son llamadas por defecto sin la necesidad de ser invocadas, en el caso anterior al llamar a "readCharacteristic ()", al finalizar esta se empieza automáticamente a correr la función "onReadCharacteristic ()".

```
    @Override
    public void onCharacteristicRead(BluetoothGatt gatt,
                                     BluetoothGattCharacteristic characteristic,
                                     int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
            setCharacteristicNotification(characteristic, enabled: true);
        }
    }

    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
                                        BluetoothGattCharacteristic characteristic) {
        broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
    }
;

rivate void broadcastUpdate(final String action) {
    final Intent intent = new Intent(action);
    sendBroadcast(intent);
```

Donde se utilizan funciones de *"broadcastUpdate"* para cambiar el TextView, de la pantalla, en esta zona se utiliza el *"setCharacteristicNotification(characteristic, true)"*, donde se envía la referencia de la característica y se activa, la opción de notificación en el descriptor de la característica del servicio, la opción de notificación debe ser previamente programada en el DISPOSITIVO CENTRAL, y dentro de nuestro DISPOSITIVO PERIFÉRICO se podrá activar, en esta forma se logra cambiar los resultados, de la característica automáticamente sin necesidad de presionar el botón de lectura constantemente.

```
public void setCharacteristicNotification(BluetoothGattCharacteristic characteristic,
                                          boolean enabled) {
    if (mBluetoothAdapter == null || mBluetoothGatt == null) {
        Log.w(TAG, msg: "BluetoothAdapter not initialized");
        return;
    }
    mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);

    // This is specific to Heart Rate Measurement.
    //if (UUID_HEART_RATE_MEASUREMENT.equals(characteristic.getUuid())) {
        BluetoothGattDescriptor descriptor = characteristic.getDescriptor(
                UUID.fromString(SampleGattAttributes.CLIENT_CHARACTERISTIC_CONFIG));
        Log.w(TAG, msg: "ENABLE NOTIFICATION");
        descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
        mBluetoothGatt.writeDescriptor(descriptor);
    //}

}
```

Ahora, los datos recibidos por el bluetooth, vienen en una cadena de datos, los cuales tienen que ser descifrados directamente en el código, donde se podrán recibir hasta 20 bytes de comunicación, mientras que en el advertising se pueden utilizar hasta 31 bytes y con el uso de *"SCAN RESPONSE"*, se duplica, aún así el uso de *"ADVERTISING"* no satisface los requerimientos planteados debido a su lentitud, de recolección de datos, ya que no es posible en las actuales recomendaciones planteadas por Nordic, el reemplazar los datos durante advertising sino que se debe reiniciar todos los datos y no los que se desean. Para poder descifrar los datos se utiliza un método genérico ofrecido por Android con el siguiente código:

```
        } else {
            format = BluetoothGattCharacteristic.FORMAT_UINT8;
            Log.d(TAG,  msg: "Heart rate format UINT8.");
        }
        final int heartRate = characteristic.getIntValue(format,  offset: 1);
        Log.d(TAG, String.format("Received heart rate: %d", heartRate));
        intent.putExtra(EXTRA_DATA, String.valueOf(heartRate));
    } else {
        // For all other profiles, writes the data formatted in HEX.
        final byte[] data = characteristic.getValue();
        Log.w(TAG,  msg: "Conversion de datos");
        if (data != null && data.length > 0) {
            final StringBuilder stringBuilder = new StringBuilder(data.length);
            for(byte byteChar : data)
                stringBuilder.append(String.format("%02X ", byteChar));
            intent.putExtra(EXTRA_DATA,  value: new String(data) + "\n" + stringBuilder.toString());
        }
    }
    sendBroadcast(intent);
}
```

1.1.2. 2. Database.java

La base de datos implementada en Android es *SQLite*. De esta manera, se crea la base de datos *Accelerometer.db*, en donde se guardan todos los datos relacionados con la aceleración del dispositivo, los resultados matemáticos obtenidos por los procesos de filtraje, y los datos del histograma obtenidos de cada resultado. Esta base de datos es creada en el método onCreate() de la clase DeviceControlActivity.java al hacer llamada al constructor de la base de datos como se observa:

```
public DataBase(Context context) {
    super(context, DATABASE_NAME,  factory: null,  version: 12);
    SQLiteDatabase db = this.getWritableDatabase();
}
```

En el cual necesita el contexto, el nombre y la versión para poder ser creada, el contexto será dado al crear llamando al constructor en la actividad donde sea necesario, una base de datos creada en una actividad no podrá ser utilizada en otra actividad ya que el contexto

será diferente, el nombre de la base de datos es escogido por el programador, en nuestro caso:

```
public static final String DATABASE_NAME = "Accelerometer.db";
```

Mientras que la versión de la base de datos, deberá ser cambiada cada vez que esta sea alterada, por ejemplo al agregar más tablas, o cambiar nombres de las tablas o sus columnas, en caso de no hacerse se podrán presentar errores cuando se prueba el programa; en el momento de llamar al constructor se llamará el método create () de la base de datos:

```
public void onCreate(SQLiteDatabase db) {//creates the databases
    db.execSQL("create table if not exists "+TABLE_NAME+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, RAW_DATA TEXT)");
    db.execSQL("create table if not exists "+TABLE_NAME_X+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, X_DATA TEXT,Y_DATA TEXT,Z_D
    db.execSQL("create table if not exists "+TABLE_NAME_H+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, H_DATA REAL)");
    db.execSQL("create table if not exists "+TABLE_NAME_HM+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, HM_DATA REAL)");
    db.execSQL("create table if not exists "+TABLE_NAME_HI+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, HI_DATA REAL)");
    db.execSQL("create table if not exists "+TABLE_NAME_HB+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, HB_DATA REAL)");
    db.execSQL("create table if not exists "+TABLE_NAME_Z+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, Z_DATA TEXT)");
    db.execSQL("create table if not exists "+TABLE_NAME1+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, MAF_DATA REAL)");
    db.execSQL("create table if not exists "+TABLE_NAME2+"( ID INTEGER PRIMARY KEY AUTOINCREMENT, BMF_DATA REAL )");
    db.execSQL("create table if not exists "+TABLE_NAME3+"(ID INTEGER PRIMARY KEY AUTOINCREMENT, IOT1_DATA REAL)");
}
```

En donde se han creado las tablas con comandos tipo SQL, para cada uno de los valores necesarios que se deseen guardar, cada tabla debe tener su llave primaria, la cual está definida por el ID y el comando de autoincremento, mientras que las demás columnas solo necesitan un nombre y el tipo de datos que van a guardar, en caso de ser texto con variables tipo “TEXT” se podrán crear en caso de tener algún valor numérico usar “REAL”, se pueden crear el número de columnas que desee el programador.

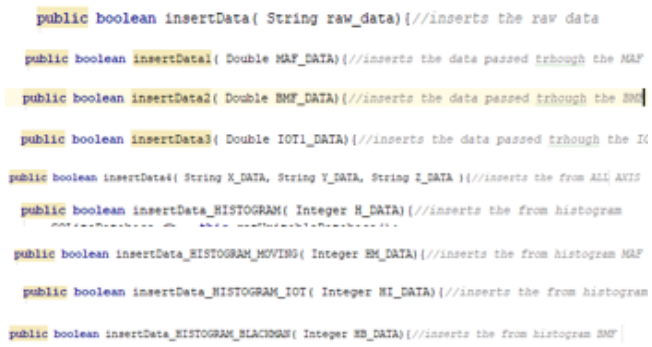
Para poder implementar la base de datos en Android es necesario implementar otro método obligatorio.

```
@Override
public void onUpgrade(SQLiteDatabase db, int i, int il) {
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME1);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME2);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME3);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME_X);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME_H);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME_HM);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME_HI);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME_HB);
    db.execSQL("DROP TABLE IF EXISTS "+ TABLE_NAME_Z);
    onCreate(db);
}
```

El método *onUpgrade ()*, será llamado cada vez que la versión de la base de datos sea cambiado, el cual se encargará de liberar cualquier información que el dispositivo móvil

haya guardado, al borrar las anteriores bases de datos con comandos SQL “DROP”, y no se haya borrado inclusive al desinstalar la aplicación, esto debido al sistema de guardado de preferencias de Android para las aplicaciones instaladas, es debido a esto que se pueden llegar a observar datos extraños en los guardados por la base de datos, los cuales nunca fueron encontrados.

Para poder insertar los datos se utilizan varias funciones tipos INSERT (), para cada uno de las operaciones matemáticas. Estas son:

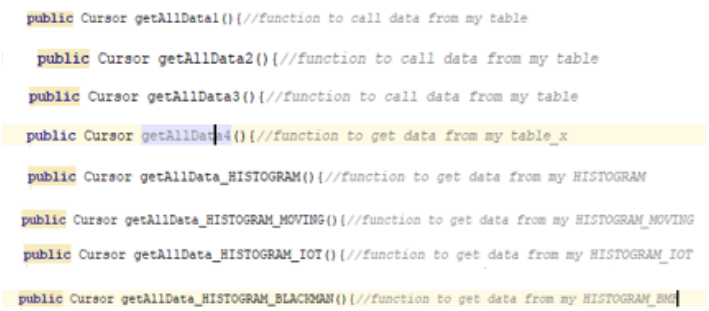


Las cuales contienen una estructura similar entre ellas, donde se varían, las columnas donde será insertado los datos y en la tabla, el código siguiente es el encontrado dentro del primer insertData ():

```
public boolean insertData( String raw_data){//inserts the raw data
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues contentValues = new ContentValues();
    contentValues.put(COL2, raw_data);
    long result = db.insert(TABLE_NAME,  nullColumnHack: null, contentValues);
    if (result == -1){
        return false;
    }
    else
        return true;
}
```

La función db.insert () retorna un valor menos uno en caso de no haberse logrado insertar los datos dentro de la base de datos, este valor será utilizado para determinar si se pudo o no realizar la operación con éxito devolviendo un valor falso o verdadero.

Una vez insertados los datos es necesario poder recuperarlos. Para esto se crean funciones *getAllData ()*, en donde se recuperaran todos los datos insertados en la tabla deseada, las funciones implementadas en este programa para recuperar los datos son:



Las estructuras de todas las funciones getAllData () son muy similares entre sí teniendo como ejemplo la primera función getAllData ():

```
public Cursor getAllData(){//function to call data from my table
    SQLiteDatabase db = this.getWritableDatabase();//make instance to the database
    Cursor res = db.rawQuery( sql: "select * from "+ TABLE_NAME,  selectionArgs: null);//get all the data
    return res;//returns my values
}
```

Todos los datos obtenidos de la base de datos vendrán en forma de cursor, este puede ser pasado a una lista o utilizado como esta.

La última función implementada dentro de la clase DataBase.Java es el método *Save-List()*, el cual es implementado debido a la naturaleza aleatoria del BLE, enviar los datos directamente a la base de datos puede generar inconvenientes, ya que mientras está ingresando un dato puede que ya haya llegado otro y de esa forma se pierda información, por este motivo se crea una lista la cual utiliza listas globales para almacenar hasta tres iteraciones de la misma función, en caso de que se haya almacenado correctamente tres iteraciones de los datos recibidos directamente del BLE, se llamará a las funciones de inserción de datos, la función implementada se observa a continuación:

Siempre es importante limpiar las lista con el comando .clear(), si no se efectúa los datos se empiezan a repetir, además el valor que se termina guardando en la primera tabla es la magnitud total de la aceleración la cual es calculada en la función AccelerationVector(), en donde se ingresaran los tres valores de cada uno de los ejes. Cabe mencionar que la lista la cual se ingresa debe poseer todos los valores de la lista de bytes recibidos por el BLE, en forma tal que la posición 0 es el primer valor recogido del eje X, la posición 3 del eje Y, y la posición 6 la del eje Z, se llama hasta 3 veces la función AccelerometerData() porque el paquete de 20 bytes alcanza a llevar 3 muestras de cada eje, ocupando 18 bytes.

```
ublic boolean saveList(String data, String data1, String data2, List<String> dataACC){

    int i = 0;//counter
    //Float datatoFloat = Float.parseFloat(String.valueOf(data));
    boolean question = false;//item to get, out of the for() the boolean element
    if(CounterBleData < 3) {//if the list is not full then add data to the list
        if (data != null){

            //savedList.add(dataACC.get(0));
            //savedList.add(dataACC.get(1));
            //savedList.add(dataACC.get(2));
            String value1 = String.valueOf(myFilter.Acceleration_Vector(dataACC.get(0),dataACC.get(3),dataACC.get(6)));
            String value2 = String.valueOf(myFilter.Acceleration_Vector(dataACC.get(1),dataACC.get(4),dataACC.get(7)));
            String value3 = String.valueOf(myFilter.Acceleration_Vector(dataACC.get(2),dataACC.get(5),dataACC.get(8)));
            savedList.add(value1);
            savedList.add(value2);
            savedList.add(value3);
        //savedList.set(CounterBleData+1, data);
        CounterBleData++;//increases the counter till reach the desire count
        return false;}

    }
    else if (CounterBleData == 3)//if the list is full then do
        for (i = 0; i<= savedList.size()-1; i++ ) {// go trhoughout all the elements of the list
            CounterBleData = 0;

                boolean insert = insertData(savedList.get(i));//save all the elements into the database
                if (insert = true) {
                    question = true;
                }


        }

    savedList.clear();//clear list

        return true;
```

1.1.3. 3. Filters.java

3.1. Variables La clase java Filters, es donde se realizan todas las operaciones matemáticas, implementación de filtros y conversión de números, en esta clase se encuentra la lógica matemática necesaria para los tres filtros digitales pasa baja, "IOT", "Moving Average Filter" y "BlackMan Filter", además de las lógicas utilizadas para crear los histogramas y sus subprocesos de "Binning". Cada una de las funciones devuelve un resultado, el cual es guardado en la base de datos dentro de la clase DeviceControlActivity(). Las variables que se utilizan para ejecutar el código dentro de esta clase son:

```
List<Double> ListaData = new ArrayList<~>();//creates a list to hold the data of the maf
List<Double> ListaDataIOT = new ArrayList<~>();//creates a list to hold the data of the IOT
List<Double> MovAv = new ArrayList<~>();
List<Double> MovIOT = new ArrayList<~>();
List<Integer> MovBinned = new ArrayList<~>();//creates a list to hold the data after being binned
List<Integer> MovHistogram = new ArrayList<~>();//creates a list to hold the data after passing trhough the hist
List<Double> ListMeanVar = new ArrayList<~>();//creates a list to hold the data after passing trhough the Mean a
List<Double> BlackMan1 = new ArrayList<~>();//creates a list to hold the data got from the convolution
public Double[] H = new Double[101];
```

El uso de listas es imperativo para poder manipular los valores de los filtros, esto debido a que al trabajarse con bancos de información, se tendrán largos LOG's de información que

deben ser analizados, las listas y array's anteriores fueron todos inicializados en el código anterior para impedir errores de objetos vacíos, es importante recordar que al trabajar con listas o array's se deben limpiar antes de utilizarlas o llenarlas de cero, el único array utilizado es el H con 101 espacios de tamaño, si se desea limpiar una lista se utiliza:

```
MovAv.clear();//clean the list
```

Donde se pone el nombre de la lista y se usa el clear (), en caso de querer iniciar un array en ceros como es hecho en los filtros posteriores se usa:

```
Arrays.fill(H, val: 0.0);//inicia el array en zero's
```

Este comando llena de ceros todo el array, después de obtener todas las listas necesarias para trabajar se procede a la implementación de filtros.

3.2. Moving Average Filter El primer filtro que se implementa es el filtro *"Moving Average Filter"* de hasta 4 valores de pro mediación, el cual es explicado en la sección de proceso digital de señales. El código del filtro se observa en:

```
public List<Double> MAF1(Cursor Data) {
```

En donde se ingresan valores tipo Cursor los cuales son obtenidos de las bases de datos, como los paquetes de datos contenidos dentro de sus tablas, el cual será convertido y manipulado para obtener la señal deseada. En caso de que se pida a esta función sin haber información previa dentro de la base de datos no se realizará ninguna de las operaciones para el cálculo de la información. Esto aumenta la robustez del sistema, se utilizan una serie de condicionales para lograr esto:

```
if (Data.getCount() != 0)
```

Donde se pregunta si el número de datos dentro del Cursor es superior a cero, si se logra superar esta fase se procede a implementar la lógica del filtro, en donde primero se pasa todos los datos del Cursor a una lista, siempre es necesario limpiar la lista antes de manipularlas:

```
MovAv.clear();//clean the list
for (Data.moveToFirst(); !Data.isAfterLast(); Data.moveToNext()) {
// The Cursor is now set to the right position
    MovAv.add(Double.parseDouble(Data.getString( i: 1)));
    }
```

Para poder utilizar MAF, es necesario recorrer todos los datos contenidos en la lista, a excepción de los últimos 4 en donde no habrá datos suficientes para aplicar la lógica del Moving Average Filter se ve a continuación:

```
for (int i = 0; i < MovAv.size() - 4; i++) {// for some reason the there is garbage data at the beggining,i is se
//res.moveToNext();
//while (res.moveToNext()) {//put the list into the table of MAF
//z++;
//x = res.getInt(0);

    y = (MovAv.get(z) + MovAv.get(z + 1) + MovAv.get(z + 2) + MovAv.get(z + 3)) / 4;//FUNCTION FOR MOVING AVERAGE
    z++;//
    Log.w(TAG,  msg: "MOVING AVERAGE FILTER: " + y + "primer numer: "+MovAv.get(z)+"ultimo numero: "+MovAv.get(z +
    ListData.add(y);


    Log.w(TAG,  msg: "congratulations");
```

Como se puede observar se recorren todos los datos de la lista con un ciclo for, la fórmula de Moving Average Filter es:

$$y[i] = \frac{1}{M} \sum_{j=0}^{M-1} x[i+j]$$

Esta fórmula es utilizada con un M igual 4 y cada valor fue posteriormente guardado en una nueva lista, los comandos tipo Log.w (), son utilizados en conjunto con el debugger para poder observar qué sucede durante la ejecución del programa", el tamaño de M determina que tan suavizado sea la señal. Si es muy pequeño su efecto será nulo sobre la información, mientras que si es demasiado grande se perderán datos importantes de la señal. La anterior función devolverá una lista de datos tipo Double:

```
turn ListData;
```

3.3. IOT Filter El segundo filtro el cual fue implementado en este programa es el filtro ofrecido por ioT, el cual fue utilizado en un proyecto de realidad virtual, para sus productos, el código original puede ser encontrado en:

https://www.built.io/blog/applying-low-pass-filter-to-android-sensor-s-readings

La implementación del filtro se observa en el siguiente método:

```
public List<Double> IOT(Cursor Data ){
```

Al igual que el filtro anterior debe recibir los datos de una base de datos en formato Cursor y devolver un listado de datos tipo *"Double"*, en donde se definen desde el principio algunas variables, en especial la variable u la cual define la frecuencia de corte del filtro. Esta frecuencia de corte viene a menudo en todos los filtros como una fracción de la frecuencia de muestreo, un ejemplo de esto es si se muestrea a 50 Hz, y se tiene una u = 0.3 se estan cortando todas las frecuencias superiores a 15 Hz. La frecuencia de corte en este filtro se define en:

```
Double x,u, y;//VARIABLES WHICH WILL BE USED FOR MATH PURPOSE

u= 0.3;// cut  off frequency of the sampling rate
```

Debe tenerse en cuenta que esto no es siempre, en algunos casos la frecuencia de corte tiene relaciones diferentes con la frecuencia de muestreo. Se debe entender que porque se lea en un libro, proyecto o trabajo, una afirmación esto no da libertad al desarrollador de dejar de pensar o investigar por su cuenta. El proceso siguiente es similar al filtro anterior donde se implementan medidas de seguridad:

```
if (Data.getCount() != 0) {
```

Se pasa la información dentro del objeto Cursor a una lista, lo más recomendable es no usar la misma lista para distintas operaciones, ya que si no se limpian correctamente en los puntos necesarios empezara a mandar información tipo basura:

```
MovIOT.clear();//clean the list
for (Data.moveToFirst(); !Data.isAfterLast(); Data.moveToNext()) {
    // The Cursor is now set to the right position
    MovIOT.add(Double.parseDouble(Data.getString( i: 1)));
}
```

Los siguientes cálculos son sacados del blog de ioT, su código se ve a continuación:

```
ListaDataIOT.add(u*MovIOT.get(0));
for (int i = 1; i < MovIOT.size(); i++) {
    //res.moveToNext();
    //while (res.moveToNext()) {//put the list into the table of MAF
    //z++;
    //x = res.getInt(0);
    //
    y = (ListaDataIOT.get(i-1) + u*(MovIOT.get(i) - MovIOT.get(i - 1)));
    //z++;//
    //Log.w(TAG, "MOVING AVERAGE FILTER: " + y + "primer numer: "+MovAv.g
    ListaDataIOT.add(y);


    Log.w(TAG,  msg: "congratulations");

}
```

Donde se inicia el primer valor de la lista y luego se procede a recorrer todos los valores de la lista MovIOT, y aplicar a la lista de salida la respectiva operación. Finalmente se entrega la lista del filtro IOT:

```
return ListaDataIOT;
```

La fórmula original extraída de ioT es:

```
<code>for i from 1 to n
y[i] := y[i-1] + α * (x[i] - y[i-1])
</code>
```

Los resultados encontrados de este filtro serán discutidos en la sección de resultados, pero la impresión inmediata son la reducción excesiva de amplitud, que aunque disminuye considerablemente la desviación estándar, y puede ser útil para proyectos donde se requiera estabilidad, no es la más recomendable para almacenamiento de información.

3.4. BlackMan Filter El siguiente filtro utilizado en el código es el filtro por sincronización de ventanas, utilizando la ventana de BlackMan. Este filtro será explicado en la sección de diseño digital, como referencia se tiene el libro de *"The Scientist and Engineer's Guide to Digital Signal Processing"*, capítulo 16. El filtro ha sido implementado en:

```
public List<Double> BlackManFilter(Cursor Data )
```

Para este filtro es necesario definir las variables:

```
Double x,u, y, pi, sum;//VARIABLES WHICH WILL BE USED FOR MATH PURPOSES
int M;
pi = 3.14159265;
u= 0.15;// cut  off frequency of the sampling rate between 0 and 0.5
x = 0.0;
y = 0.0;
M = 100; // sets the length of the filter to 101 points
sum = 0.0;
```

Donde M es el tamaño del filtro, en este caso un tamaño de 100 y u continuará siendo la frecuencia de corte, en este caso especial de filtro u solo debe ir entre los rangos de $0 < u < 0.5$, debido a la simetría de este filtro. Se continua con la seguridad:

```
if (Data.getCount() != 0) {
```

Luego se procede a insertar los datos del Cursor dentro de una lista:

```
MovAv.clear();//clean the list
for (Data.moveToFirst(); !Data.isAfterLast(); Data.moveToNext()) {
    // The Cursor is now set to the right position
    MovAv.add(Double.parseDouble(Data.getString( i: 1)));
}
```

Para poder implementar la lógica del filtro es necesario el uso de *"array's"*. Uno de ellos ya ha sido iniciado pero falta iniciar otro:

```
Double[] BlackMan = new Double[MovAv.size()];
Arrays.fill(BlackMan, val: 0.0);
Arrays.fill(H, val: 0.0);//inicia el array en zero's
```

El array H contiene el filtro ventana, el cual es calculado utilizando las primeras 100 muestras del banco de datos de ahí M = 100, el segundo array BlackMan es donde se guardan los datos obtenidos después del filtraje. Ambos deben ser iniciados en 0 todos sus valores. Al haber realizado los pasos anteriores es posible empezar la lógica del filtro:

```
for (int i = 0; i <= 100; i++) {//calculates the filter kernel
    //res.moveToNext();
    //while (res.moveToNext()) {//put the list into the table of MAF
    //z++;
    //x = res.getInt(0);
    //
    if (i-M/2 == 0){
        H[i] = 2*pi*u;
        Log.w(TAG, msg: "Media");
    }
    else
        H[i] = Math.sin(2*pi*u*(i-M/2))/(i-M/2);
        H[i] = H[i]*(0.54-0.46*Math.cos(2*pi*i/M));

            }

Log.w(TAG, msg: "BLACKMAN FILTER  OBTENIDO");

for (int il= 0; il <= 100; il++){//normalize the low pass filter kernel for unity dc gain
    sum = sum + H[il];
}
for (int i2= 0; i2 <= 100; i2++){
    H[i2] = H[i2]/sum;
}
BlackMan1.clear();//makes suer there is nothing within the blackman1 list
for (int i3= 100; i3 < MovAv.size(); i3++){//convolve the input signal with the fiilter kernel
    BlackMan[i3]=  0.0;
    for (int i4= 0; i4 <= 100; i4++){
        BlackMan[i3]= BlackMan[i3]+ MovAv.get(i3-i4)*H[i4];//+ MovAv.get(i3-i4)*H[i4];

    }

    BlackMan1.add(BlackMan[i3]);
}
```

Primero se procede a calcular el filtro, después se realiza la normalización de los datos. Finalmente se inicia limpiando la lista BlackMan1 y se realiza un proceso de convolución

entre el filtro encontrado y los datos a partir del índice 100. Por último se agrega a una lista la cual será devuelta por la función:

```
return BlackMan1;
```

Los resultados de este filtros serán presentados en la etapa de pruebas, aun así las primeras impresiones de los resultados indican que este filtro ofrece el mejor filtraje sin afectar la señal, aunque la pérdida de los primeros 100 datos puede ser un valor considerable dependiendo de la aplicación.

3.5. Histograma Todas las señales obtenidas en este proyecto han sido puestas bajo un nivel de procesamiento, en el cual se busca garantizar que las pruebas den los datos más fiables, debido a esto es necesario obtener valores numéricos los cuales soporten las decisiones tomadas, en este aspecto entran el valor promedio y la desviación estándar, valores intrínsecos de toda señal, los cuales pueden ser utilizados de varias maneras dependiendo de las necesidades del programador, en este caso utilizados para determinar la precisión del mismo movimiento diferentes veces. Desafortunadamente el cálculo de estas constantes utilizando la señal original ya sea filtrada o bruta, plantea una pesada carga de procesamiento para el sistema, es por eso que se utiliza la herramienta del histograma el cual cuenta cada una de las veces que uno de los valores posibles de la señal ha aparecido, la cual reducirá el análisis de una señal de cientos de miles o inclusive millones de muestras, a cientos o miles posibilidades de la señal, el histograma también es explicado en el módulo de diseño digital de señales, aunque la lógica para obtener un histograma es sencilla, los datos recolectados puede que no estén preparados para su conteo.

Al pasar por los diferentes filtros, conversores de magnitud o puede que los datos y vengan en este formato desde el sensor, se podrán recibir números de punto flotante y no los enteros, esto hará que las posibles resultados de una variable de 256 posibles resultados en entero, se conviertan en millones de posibles valores si se tiene en cuenta la posibilidad 200.345 o la posibilidad 134.23, en vez de ir en uno en uno, para poder resolver este problema se hace uso del concepto de *"BINNING"*, el cual aproxima los valores de punto flotante en un rango de datos definido por el programador, ejemplo todos los valores dentro del rango (20-22), serán pasados automáticamente al bin 11, el número de bin's es decisión del programador. En este caso el proceso de binning se hace en las siguientes líneas de código:

```
public List<Integer> Binned(Cursor Data ){
```

Donde se reciben los datos de una base de datos cualquiera, se convierte a una lista:

```
MovBinned.clear();//clean the list
for (Data.moveToFirst(); !Data.isAfterLast(); Data.moveToNext()) {
    // The Cursor is now set to the right position
    Double binnedPRE = Double.parseDouble(Data.getString( i: 1));//
    int binnedPOST = (int) Math.rint(binnedPRE); //turns the value
    MovBinned.add(binnedPOST);
}
```

Aplicando un bin equivalente a un bin por posible valor de la muestra, ejemplo el valor 456.23 será convertido a 456, 455.75 será convertido a 456, esto gracias a la librería Math de java en donde al utilizar la función:

```
int binnedPOST = (int) Math.rint(binnedPRE);
```

Todos los datos punto flotante son aproximados a su entero más cercano y agregado a la lista tipo Integer, la cual será devuelta por la función:

```
return MovBinned
```

Estos datos son pasados eventualmente a la función de cálculo del histograma visto en:

```
public List<Integer> Histogram(List<Integer> Data){//logic for the histogram
    int possibilities = Collections.max(Data);//get the the max value of the
    Integer[] HistogramHolder = new Integer[possibilities+1];
    Arrays.fill(HistogramHolder, val: 0);//initialize my array with zeros
    MovHistogram.clear();//ALWAYS CLEAN THE DATA FOR YOUR LIST'S
    for (int i = 0; i < Data.size(); i++) {//put the cursor into  list
        //Log.w(TAG, "histogram numero: "+Data.get(i));
        if (Data.get(i)>0) {
            HistogramHolder[Data.get(i)] = HistogramHolder[Data.get(i)] + 1;
        }
    }
    for (int il = 0; il < possibilities; il++){//make the list for containin
        MovHistogram.add(HistogramHolder[il]);
    }

    MovHistogram.add(Data.size());
    Log.w(TAG, msg: "size 1 phase: "+Data.size());
    MovHistogram.add(possibilities);
    Log.w(TAG, msg: "possibilities 1 phase: "+possibilities);
    return MovHistogram;


}
```

Se obtienen los posibles valores de la señal la cual da el máximo valor de la lista ingresada, esto es posible hacer debido a que la señal a la cual se aplica los histogramas es la

magnitud de la aceleración obtenida de las 3 aceleraciones de cada eje x, y, z. Si se desea utilizar este histograma para señales negativas es necesario modificar el código. Debido a la facilidad que representa usar array's para estos cálculos matemáticos se inicializa uno con un tamaño igual al máximo de la señal más uno ya que el array también debe tener en cuenta el cero, en caso de no hacerlo así no funcionara el código, finalmente el array será pasado a una lista de enteros, para poder realizar el cálculo del valor promedio y de la desviación estándar se hace necesarios valores obtenidos en esta función. Por eso los últimos dos valores de la lista siempre guardan:

```
MovHistogram.add(Data.size());
```

El tamaño total de la información antes de ser pasada por el histograma y además necesitará el número de posibilidades que puede obtener la señal original:

```
MovHistogram.add(possibilities)
```

Finalmente la señal retorna la lista:

```
return MovHistogram;
```

3.6. Valor Promedio y Desviación Estándar Los valores promedio y desviación estándar, o en ingles, *"Mean"* y *"Standard Deviation"* proporcionan la información sobre el valor promedio de la señal y como la señal oscila alrededor de ésta. El cálculo de esta puede llegar a ser un alto peso de procesamiento para la señal pero gracias al histograma se puede realizar con la siguiente función la cual es implementada en HistogramActivity.java:

```
public List<Double> Mean(List<String> HistogramHolder){//logic used to the get the mean a
    //FIRST VALUE OF THE LIST WILL BE THE MEAN
    //SECOND VALUE OF THE LSIT WILL BE THE SD
    Double mean = 0.0;
    Double variance = 0.0;
    Double StandardDeviation = 0.0;
    ListMeanVar.clear();
    Integer size = HistogramHolder.size();
    Integer Datasize = Integer.parseInt(HistogramHolder.get(size-2));
    Integer possibilities = Integer.parseInt(HistogramHolder.get(size-1));
    for (int i2 = 0; i2 < possibilities; i2++){
        mean = mean + i2*Integer.parseInt(HistogramHolder.get(i2));
    }
    mean = mean/Datasize;
    ListMeanVar.add(mean);

    for (int i3 = 0; i3 < possibilities; i3++){
        variance = variance+Integer.parseInt(HistogramHolder.get(i3))*Math.pow(i3-mean, 2)
    }
    variance = variance/(Datasize-1);
    StandardDeviation = Math.sqrt(variance);
    ListMeanVar.add(StandardDeviation);

    return ListMeanVar;
}
```

Con las ecuaciones del valor promedio igual a:

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} x_i$$

Y la ecuación de la desviación estándar:

$$\sigma^2 = \frac{1}{N-1}\sum_{i=0}^{N-1}(x_i - \mu)^2$$

Estas ecuaciones han sido modificadas en el código final para mejorar su eficiencia.

1.1.4. 4. DeviceControlActivity.java

En esta clase se crean todas las funciones que se conectan directamente con los botones del layout Button, esta clase java hereda de actividad, contiene las clases override *onCreate*

(), onResume (), onPause (), onDestroy (), onCreateOptionsMenu (), onOptionsItemSelected (), las cuales se encargan de las funcionalidades básicas de BLE, a continuación se presenta una lista de sus funciones:

OnCreate (): cada vez que se haga llamado a la actividad Device Control Activity, lo primero que realizara es entrar en el onCreate, siendo este su punto inicial:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.button_control);
    myDb = new DataBase( context: this);//calling the constructor to create the data base
    data_value = (TextView) findViewById(R.id.data_value);//makes instance to the text
    Button3 = (Button) findViewById(R.id.Button3);
    //Button6 = (Button) findViewById(R.id.Button6);
    graph = (GraphView) findViewById(R.id.graph);//makes the instance to the graph
    graph.getViewport().setXAxisBoundsManual(true);
    graph.getViewport().setMinX(100);
    graph.getViewport().setMaxX(1000);
    //graph.getViewport().setScrollable(true);
    graph.getViewport().setScalable(true);
    GridLabelRenderer glr = graph.getGridLabelRenderer();
    glr.setPadding(32);// change the size of the table on graph view

    final Intent intent = getIntent();
    mDeviceName = intent.getStringExtra(EXTRAS_DEVICE_NAME);
    mDeviceAddress = intent.getStringExtra(EXTRAS_DEVICE_ADDRESS);
    sqLiteDatabase = myDb.getWritableDatabase();
    // Sets up UI references.
    //((TextView) findViewById(R.id.device_address)).setText(mDeviceAddress);
   // mGattServicesList = (ExpandableListView) findViewById(R.id.gatt_services_list);
  //  mGattServicesList.setOnChildClickListener(servicesListClickListner);
  //  mConnectionState = (TextView) findViewById(R.id.connection_state);
    mDataField = (TextView) findViewById(R.id.data_value);

    getActionBar().setTitle(mDeviceName);
    getActionBar().setDisplayHomeAsUpEnabled(true);
    Intent gattServiceIntent = new Intent( packageContext: this, BluetoothLeService.class)
    bindService(gattServiceIntent, mServiceConnection, BIND_AUTO_CREATE);
    OnClickView();// call the funcition to display the data stored in my DataBase
}
```

Esta función hace llamado a las funciones de inicio, como el constructor de la base de datos, para lo cual solo necesita el contexto donde se utilizara, con un THIS. Toda base de datos en esta actividad no puede ser utilizada en otras actividades, si se requiere pasar información se debe utilizar Intents. Para la graficación se hace uso de la librería android GraphView, la cual ha sido previamente descargada en .jar, instalada en la carpeta de

librerías del proyecto, y seleccionado la opción de agregar como librería, al dar clic derecho en ella y seleccionar esta opción. Para poder utilizar la librería es necesario escribir los imports:

```
import com.jjoe64.graphview.GraphView;
import com.jjoe64.graphview.GridLabelRendere
import com.jjoe64.graphview.series.DataPoint
import com.jjoe64.graphview.series.LineGraph
```

Cuando es posible utilizar las funciones de GraphView en el método onCreate, se implementan donde se ha obtenido una instancia de la gráfica puesta en el layout Button y se ha configurado sus parámetros en el resto, haciendo que sea posible expandir la gráfica, con cuadros mínimos de 100 muestras, y máximos de 1000 muestras. Por encima de 1000 en el eje X, la señal se saldrá de la vista y tendrá que utilizarse el deslice de pantalla.

Los Intent en este método son utilizados con el fin de obtener información enviada desde la clase DeviceScanActivity (), donde se recibe el nombre del dispositivo y la dirección del mismo. Luego, se determina la nueva vista del proyecto, una vez terminadas todas las funciones del onCreate, la actividad entra en estado continuo que viene después de su creación, de ahí se llama al método *onResume ()*.

```
protected void onResume() {
    super.onResume();
    registerReceiver(mGattUpdateReceiver, makeGattUpdateIntentFilte
    if (mBluetoothLeService != null&&myDb.getAllData()==null) {
        final boolean result = mBluetoothLeService.connect(mDeviceA
        Log.d(TAG,  msg: "Connect request result=" + result);
    }
}
```

En este punto se crea un registerReceiver() que es leído dentro del BroadcastUpdate en el método OnReceive(), al cual se le enviara el contexto y un intent que contiene los diferentes posibles estados de la conexión. Se emplean métodos de seguridad, preguntando si existe el servicio buscado, y exige para tener una conexión con la base de datos principal tiene que estar vacía desde el comienzo. De este modo, si se pueden superar estos filtros entonces se llama a la conexión. Siempre que se cree un registerReceiver es necesario cerrarlo en alguna parte del código, en general es recomendado hacerlo cuando se para la actividad en el método override *onPause()*.

```
@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mGattUpdateReceiver);
}
```

Cuando se cierra la aplicación por completo se hace llamado al método *onDestroy()* donde lo único que hace es resetear el servicio encontrado a nulo, y hacer la desconexión del servicio.

```
@Override
protected void onDestroy() {
    super.onDestroy();
    unbindService(mServiceConnection);
    mBluetoothLeService = null;
}
```

Para hacer una conexión con el servicio previamente creado en la clase java BluetoothLeService.java es necesario implementar el siguiente código.

```
private final ServiceConnection mServiceConnection = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName componentName, IBinder service) {
        mBluetoothLeService = ((BluetoothLeService.LocalBinder) service).getService();
        if (!mBluetoothLeService.initialize()) {
            Log.e(TAG,  msg: "Unable to initialize Bluetooth");
            finish();
        }
        // Automatically connects to the device upon successful start-up initializatic
        mBluetoothLeService.connect(mDeviceAddress);
    }

    @Override
    public void onServiceDisconnected(ComponentName componentName) {
        mBluetoothLeService = null;
    }
};
```

El cual se encarga de manejar las operaciones cuando se intenta realizar una conexión al servicio, o cuando se intenta desconectar del servicio. En caso de no poder iniciar la conexión al servicio, se acaba la actividad con el comando *finish()*. Por otro lado al encontrar el servicio se envía el comando *connect()*, si se establece la conexión. La información recibida, es el envío de paquetes de otras actividades entrando a esta clase.

```
private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver()
```

El único método implementado en este Broadcast Receivers el método override *OnReceive()*.
Dentro de este método lo primero que se hace es comprobar que acción se realiza. La acción que contiene el intent ha sido llenada con el método anterior *makeGattUpdateIntentFilter*,

```
public void onReceive(Context context, Intent intent) {
```

el cual agrega las posibles acciones que tendrá el intent, en caso de pedir una conexión se realiza:

```
if (BluetoothLeService.ACTION_GATT_CONNECTED.equals(action)) {
    mConnected = true;
    updateConnectionState("Connected");
    invalidateOptionsMenu();
```

Donde modifican las variables necesarias para establecer una conexión, y se prepara para redibujar mi actionBar. Si se pide una desconexión, se implementa:

```
} else if (BluetoothLeService.ACTION_GATT_DISCONNECTED.equals(action))
    mConnected = false;
    updateConnectionState("Disconnected");
    invalidateOptionsMenu();
    clearUI();
```

Donde se pasan las variables ha desconectado, se redibuja la barra de acción y se limpia cualquier hilo que siga conteniendo información. Si la acción encontrada dice que hay información disponible llegando del dispositivo BLE periférico se ejecuta:

```
} else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action))
```

Dentro de este condicional, se plantean medidas de seguridad para asegurar que haya llegado información. De esta manera:

```
if (intent.getStringArrayListExtra(BluetoothLeService.EXTRA_DATA) != null ) {
    if (intent.getStringArrayListExtra(BluetoothLeService.EXTRA_DATA).size()>0) {
```

En caso de determinar que realmente hay información dentro de llegada y no basura, se procede a obtener la información enviada de la clase java BluetoothLEService.java. Para hacer esto se llama a una lista llamada plain, en donde se agregan los valores del intent, siempre es importante limpiar las listas antes de usarlas.

```
plain.clear();
plain.addAll(intent.getStringArrayListExtra(BluetoothLeService.EXTRA_DATA));
```

Toda la información obtenida de esta lista, se pasa por las clases *saveList_x()* que guarda los datos de los 3 ejes de aceleración por separado y *saveList()* que guarda los datos de la magnitud total encontrada del acelerómetro, en la clase java DataBase.

```
Boolean isInserted_X = myDb.saveList_X(plain.get(0), plain.get(1), plain.get(2), plain);
Boolean isInserted = myDb.saveList(plain.get(0), plain.get(1), plain.get(2), plain);//d
```

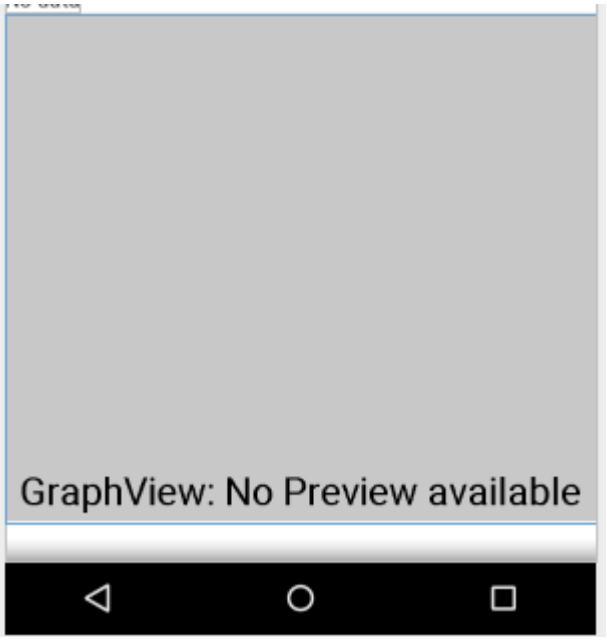
Donde lo más importante es el ingreso de la lista completa plain, de ahí se saca toda la información necesaria para guardar en la base de datos. La forma en que se hace esto, es al recibir estas funciones las listas plain, guardan los valores encontrados en una lista más grande, organizando cada elemento de la lista como una lectura de uno de los ejes del acelerómetro, una vez hecho esto lo guarda en una lista y espera a que este proceso se repita 5 veces. Cuando haya repetido la quinta vez, se procederá a guardar en las bases de datos respectivas de cada uno de estos, la razón de este ciclo de espera es debido a la naturaleza aleatoria en que llegan los datos BLE, que llegan a traer inconvenientes a la hora de guardar información en bases de datos. Por lo tanto es recomendado guardarlo primero en una lista y luego mandar toda la lista a la base de datos, en este caso se guardan 5 veces el paquete encontrado del intent. Cada paquete contiene 3 muestras de cada eje, entonces cada lista agrega hasta 15 muestras de magnitud completa de la aceleración por carga a la base de datos y en la función saveList_X() el número es mayor, se guardan 45 valores cada vez que se cargue la lista a la base de datos. Estos métodos han sido designados como tipo booleanos, en caso de lograr enviar datos a la base de datos, retornan un valor TRUE, si se ingresa un nuevo se pregunta si es TRUE y se procede a graficar.

```
if (isInserted == true) {
```

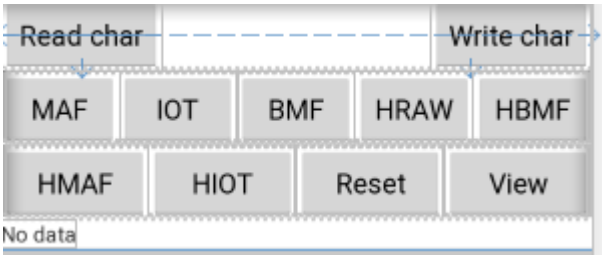
Para poder graficar se emplea el siguiente código

```
if (isInserted == true) {
    graph.removeAllSeries();
    series = new LineGraphSeries<DataPoint>();//calls function from database and make
    double y, x;

    x = -5.0;
    y = 10000;
    Cursor res = myDb.getAllData();
    if (myDb.getAllData().getCount() != 0 || myDb.getAllData().getCount() != count1)

        count1 = myDb.getAllData().getCount();
        while (res.moveToNext()) {
            x = res.getInt( i: 0);
            y = Double.parseDouble(res.getString( i: 1));
            //y = Math.sin(x);
            series.appendData(new DataPoint(x, y), scrollToEnd: true, res.getCount());
        }
        //graph.addSeries(series);//shows the graph
    } else return;
```

Donde se pide la información ingresada a la base de datos principal y mediante el uso de un WHILE, se recorre toda la función. Para graficar se utiliza la columna 0 de la tabla en donde se guarda el ID que será el eje x y la columna 1 que es el eje y, donde se han guardado los datos de la magnitud de la aceleración bruta. Desafortunadamente al hacer pruebas, la graficación en tiempo real funcionó bien hasta que la señal lleno el espacio límite de 1000 muestras asignado previamente, una vez la señal se pasa de este valor, aunque sigue siendo posible graficarlo mediante el uso de deslizamientos, el desempeño del software cae, y empieza a bloquearse. Por tanto se ha deshabilitado la graficación en tiempo real, la gráfica se visualiza en la siguiente vista.



Una vez hablado de la recepción de datos es debido hablar de los botones que realizan las funcionalidades del API, la lista de botones de la aplicación se puede observar en la siguiente imagen:



El primer botón a discutir es el View, al presionarlo se ejecuta el siguiente código:

```
public void OnClickView(){//method use when click button VIEW
    Button3.setOnClickListener(
            (view) → {
                    Cursor res = myDb.getAllData();//calls the function getALlData
                    if (res.getCount()==0){
                        //we get here if there is nothing on the table or no result
                        showMessage( title: "Error",  Message: "no data in the dataBase");
                        return;
                    }
                    StringBuffer data_BLE_DataBase_Raw = new StringBuffer();//if there is d
                    while (res.moveToNext()){//moves trhoughout the table
                        data_BLE_DataBase_Raw.append("Id: "+ res.getString( i: 0)+"\n");//ge
                        data_BLE_DataBase_Raw.append("Data: "+ res.getString( i: 1)+"\n\n");
                    }

                    //show all DATA

                    showMessage( title: "Data", data_BLE_DataBase_Raw.toString());
            }
    );
```

Donde se pide la información de la base de datos principal, se emplean medidas de seguridad. Además se pregunta si existe información, en caso de no haber, se envía un mensaje al usuario diciéndole que no hay información en la base de datos. En caso de existir información se agregan todos los datos de la base de datos a un *StringBuffer* y se despliega en una ventana emergente (Toast) mediante la función showMessage, la cual fue implementada más adelante en el código.

Al presionar el botón MAF, se llama la información de la base de la tabla RawAccelerometer_Table_RAW y filtrarla mediante la función MAF1 definida en la clase Filters.java

```
public void OnclickMAF(View v){
```

Una vez dentro se implementan medidas de seguridad para verificar que la información exista, es debido recordar que es un proceso pesado este cálculo, y por eso en vez de calcular la señal cada vez que se presiona la información, solo se hace el cálculo la primera vez que se presione el botón, y guardara la información en la nueva tabla respectiva de estos datos. Las siguientes veces que se presione el botón, simplemente se llamará a la información ya recolectada siendo mucho más rápido el despliegue de información por eso se pregunta si la tabla que contiene la información del MAF debe estar vacía la primera vez que se realiza la operación. Este tipo de medidas de seguridad se verán en la implementación de todos los filtros e histogramas y no se vuelve a hablar de ello.

```
Cursor res2 = myDb.getAllData1();
if (res != null && res.getCount()>0) {
    if (res2.getCount()==0) {
```

Si se ha logrado pasar las medidas de seguridad, el programa está listo para poder empezar a realizar la lógica para el cálculo de la señal procesadas por los filtros, si ese es el caso es debido recordar que estos son procesos de alto peso computacional para el sistema. Si se implementan directamente en el código, la eficiencia de este será reducida, y no se podrá hacer nada mientras se calculan los datos, es debido a esto que se utilizan los hilos secundarios, para que el programa corra en el *"background"* estas operaciones complejas mientras que el hilo principal sigue su rumbo. Es importante mencionar que no se puede hacer que un hilo secundario afecte de alguna manera la vista del usuario, debido a esto se hace uso de handle, los cuales son llamados una vez acabe la operación dentro del hilo secundario. El código de esta operación se observa a continuación:

```
    Runnable r = () → {
            Cursor res = myDb.getAllData();//call all of the
            DataAccMAf.addAll(myFilter.MAF1(res));//gets all
            for (int i1 = 0; i1 < DataAccMAf.size(); i1++) {
                myDb.insertData1(DataAccMAf.get(i1));
            }
            handlerMAF.sendEmptyMessage( what: 0);
    };
    Thread MAFThread = new Thread(r);
    MAFThread.start();
}
```

Donde se pide la información de la base de datos principal, se pasa a una lista los valores obtenidos del filtro MAF, el cual fue explicado en el capítulo de filtros. Se guarda la información en la tabla designada para estos datos y finalmente se hace llamado al handle, siempre enviar un cero, esto hará saber al handle que ya puede comenzar a trabajar. El handle se observa a continuación:

```
Handler handlerMAF = handleMessage(msg) → {
```

Para la visualización de la información de MAF, se muestra por encima de la señal original para poder realizar una comparación entre las dos. Para hacer esto, primero se remueve cualquier gráfica anterior a este dibujo, luego se llamará a una función implementada, que dibuja la señal llamada showGraphRaw (), se crea una nueva serie para la nueva señal y se pone el color de la nueva serie en rojo, esto se hace en:

```
graph.removeAllSeries();
shoeGraphRaw();
seriesMAF = new LineGraphSeries<DataPoint>();
seriesMAF.setColor(Color.RED);
```

La siguiente parte del código se encarga de graficar esta señal y obtener los datos de la base de datos de MAF, esto ya fue explicado y no se profundizará en lo que hace la función.

```
Cursor res1 = myDb.getAllData1();//gets all data from database

if (res1.getCount() != 0) {
    //count1 = myDb.getAllData1().getCount();
    //for (int i = 0; i<500; i++) {
    res1.moveToFirst();//goes back to the first data of the database
    while (res1.moveToNext()) {
        x = res1.getInt( i: 0);
        //y = res.getDouble(1);
        //x = x+0.1;
        y = Double.parseDouble(res1.getString( i: 1));
        //y = Math.sin(x);
        seriesMAF.appendData(new DataPoint(x, y),  scrollToEnd: true, res1.getCount());
    }
    graph.addSeries(seriesMAF);//shows the graph
    myFilter.Binned(res1);
} else
    showMessage( title: "Error",  Message: "no data in the dataBase");
return;
```

El siguiente botón que se analiza es IOT, en donde se aplica el filtro IOT estudiado ya descrito en el capítulo de filtros. La implementación es la misma que para MAF, su código es:

```
Cursor res2 = myDb.getAllData3();
if (res != null && res.getCount()>0 ) {//verifies that there is da
    if (res2.getCount()==0) {

        Runnable r = () → {
                Cursor res = myDb.getAllData();//call all of the d
                DataAccIOT.addAll(myFilter.IOT(res));//gets all da
                for (int i1 = 0; i1 < DataAccIOT.size(); i1++) {
                    myDb.insertData3(DataAccIOT.get(i1));
                }
                handlerIOT.sendEmptyMessage( what: 0);
        };
        Thread IOTThread = new Thread(r);
        IOTThread.start();
    }
    else
        handlerIOT.sendEmptyMessage( what: 0);
        showMessage( title: "FILTER",  Message: "IOT");//in case the d
    return;
}
else
    showMessage( title: "Error",  Message: "no data in the dataBase");
return;
```

```
Handler handlerIOT = handleMessage(msg) → {
        EventIntentHistogram = 2;//second case for the intent
        //graph.removeAllSeries();//cleans any series
        graph.removeAllSeries();
        shoeGraphRaw();

        seriesIOT = new LineGraphSeries<DataPoint>();//calls function from database and make
        seriesIOT.setColor(Color.GRAY);

        double y, x;//initialize variables that will hold the math operations
        //DataAccMAf.addAll(myFilter.MAF1(res));

        x = -5.0;
        y = 10000;
        StringBuffer data_BLE_DataBase_Raw = new StringBuffer();//if there is data creates a
        //Log.w(TAG, "DATOS BASE: :" + DataAccIOT.get(0));
        Cursor res1 = myDb.getAllData3();//gets all data from database
        if (res1.getCount() != 0) {
            //count1 = myDb.getAllData1().getCount();
            //for (int i = 0; i<500; i++) {
            res1.moveToFirst();//goes back to the first data of the database
            while (res1.moveToNext()) {
                x = res1.getInt( i: 0);
                //y = res.getDouble(1);
                //x = x+0.1;
                y = Double.parseDouble(res1.getString( i: 1));
                //y = Math.sin(x);
                seriesIOT.appendData(new DataPoint(x, y),  scrollToEnd: true, res1.getCount());
            }
            graph.addSeries(seriesIOT);//shows the graph
        } else
            showMessage( title: "Error",  Message: "no data in the dataBase");
        return;
```

Al igual que en MAF, se usa hilos, y gráficas para poder observar la nueva forma de la señal después de pasar por este tipo de filtro, el color de la nueva señal pasada por este filtro es gris.

El siguiente botón a discutir es el botón BMF, el cual implementa el filtro *"BLACKMAN FILTER"*, explicado en la sección de filtros. Su implementación es igual a los dos filtros anteriores:

```
public void OnclickBMF(View v){
    Cursor res = myDb.getAllData();//call all of the data base data of
    Cursor res2 = myDb.getAllData2();
    if (res != null && res.getCount()>0 ) {//verifies that there is dat
        if (res2.getCount()==0) {

            Runnable r = () → {
                    Cursor res = myDb.getAllData();//call all of the da
                    DataAccBMF.addAll(myFilter.BlackManFilter(res));//
                    for (int il = 0; il < DataAccBMF.size(); il++) {
                    myDb.insertData2(DataAccBMF.get(il));
                            }
                    handlerBMF.sendEmptyMessage( what: 0);
            };
            Thread BMAThread = new Thread(r);
            BMAThread.start();
        }
        else
            handlerBMF.sendEmptyMessage( what: 0);
            showMessage( title: "FILTER",  Message: "BMF");//in case the da
            return;
    }
    else
        showMessage( title: "Error",  Message: "no data in the dataBase");
        return;
}
```

```
Handler handlerBMF = handleMessage(msg) → {
        //graph.removeAllSeries();//cleans any series
        EventIntentHistogram = 1;//first case
        graph.removeAllSeries();
        shoeGraphRaw();
        seriesBMF = new LineGraphSeries<DataPoint>();//calls function from database and makes
        seriesBMF.setColor(Color.GREEN);
        double y, x;//initialize variables that will hold the math operations
        //DataAccMAf.addAll(myFilter.MAF1(res));

        x = -5.0;
        y = 10000;
        StringBuffer data_BLE_DataBase_Raw = new StringBuffer();//if there is data creates a
        //Log.w(TAG, "DATOS BASE: :" + DataAccBMF.get(0));
        Cursor res1 = myDb.getAllData2();//gets all data from database
        if (res1.getCount() != 0) {
            //count1 = myDb.getAllData1().getCount();
            //for (int i = 0; i<500; i++) {
            res1.moveToFirst();//goes back to the first data of the database
            while (res1.moveToNext()) {
                x = res1.getInt( i: 0)+46;
                //y = res.getDouble(1);
                //x = x+0.1;
                y = Double.parseDouble(res1.getString( i: 1));
                //y = Math.sin(x);
                seriesBMF.appendData(new DataPoint(x, y),  scrollToEnd: true, res1.getCount());
            }
            graph.addSeries(seriesBMF);//shows the graph
        } else
            showMessage( title: "Error",  Message: "no data in the dataBase");
            return;
};
```

Al igual que los filtros anteriores se utiliza hilos, para mejorar considerablemente la experiencia del usuario, con sus respectivos handle para ser capaz de graficar la señal resultante, el nuevo color de la señal es verde.

El siguiente botón a analizar en HRAW, el cual permite hacer el cálculo del histograma, con los valores brutos de la magnitud total de la aceleración, y enviar la información a la actividad HistogramActivity.java, cuyo único propósito de esta clase java es recibir la información del histograma, implementar las funciones matemáticas de promedio y desviación estándar, graficar el histograma y mostrar los datos de promedio y desviación estándar. Para el histograma se utilizan las mismas medidas de seguridad y el uso de hilos que en los filtros digitales pasa baja, aun así las funciones utilizada son distintas, estas se pueden ver en el siguiente código:

```
Log.w(TAG,  msg: "ENTRO EN EL HISTOGRAMA");
Cursor res = myDb.getAllData();//call all of the data base data of table raw data
DataAccBINNED.clear();
DataAccHISTOGRAM.clear();
DataAccBINNED.addAll(myFilter.Binned(res));//gets all data from filter Moving ave
DataAccHISTOGRAM.addAll(myFilter.Histogram(DataAccBINNED));
for (int i = 0; i < DataAccHISTOGRAM.size(); i++) {
    boolean insertHistogram = myDb.insertData_HISTOGRAM(DataAccHISTOGRAM.get(i));

}
//handlerHISTO.sendEmptyMessage(0);
reachHistogram();
```

Donde se obtiene la información de la base de datos de la información que se quiera calcular el histograma, se limpian listas, se utilizan las funciones Binned(), Histogram(), ambas definidas en la clase java myFilter.java, el valor del histograma se guarda en la tabla respectiva de este histograma. Una vez realizada toda esta lógica, a diferencia de los anteriores filtros, el histograma no gráfica directamente si no que lo encarga a otra actividad por esto no es necesario el uso de handle. En todo caso se debe crear una función la cual sea capaz de comunicarse con la actividad HistogramActivity.java, esto es invocando al método *reachHistogram()*. El código se observa en:

```
public void reachHistogram(){//function use to reach the histogram
    Cursor resHistogram = myDb.getAllData_HISTOGRAM();//call all of the data base data of table histogram dat
    //Cursor resHistogram = getHistogramDecision();//call all of the data base data of table histogram data b
    ListIntentHistogram.clear();//clears all the data
    for (resHistogram.moveToFirst(); !resHistogram.isAfterLast(); resHistogram.moveToNext()) {//put the curso
        // The Cursor is now set to the right position
        ListIntentHistogram.add(resHistogram.getString( i: 1));
    }
    final Intent ReachHistogram = new Intent( packageContext: this, HistogramActivity.class);//declares the list
    final int result = 1;
    ReachHistogram.putStringArrayListExtra( name: "callingActivity", (ArrayList<String>) ListIntentHistogram);
    startActivity(ReachHistogram);
```

Donde se hace llamado a la tabla que contiene al histograma, la cual es pasada a una lista que se envía mediante un intent. Para alcanzar la siguiente actividad se hace llamado a los comandos, con *startActivity()* se lanza la nueva actividad.

Ahora, el botón HBMF, se encarga del cálculo del histograma de los datos recolectados después de aplicar el filtro *"BLACKMAN FILTER"*. Cada filtraje y set de datos tiene su propia tabla, los demás aspectos de su implementación son iguales al cálculo del anterior histograma. Su código es:

```
public void OnclickHisNOW_BLACKMAN(View v){
    showMessage( title: "Histogram",  Message: "BMF DATA");
    Cursor res = myDb.getAllData2();//call all of the data base data of table raw data

    Cursor res2 = myDb.getAllData_HISTOGRAM_BLACKMAN();
    if (res != null&&res.getCount()>0) {//verifies that there is data in the data base
        if (res2.getCount()==0) {

            Runnable r = () → {
                    Log.w(TAG,  msg: "ENTRO EN EL HISTOGRAMA");
                    Cursor res = myDb.getAllData2();//call all of the data base data of table raw data
                    DataAccBINNED.clear();
                    DataAccHISTOGRAM.clear();
                    DataAccBINNED.addAll(myFilter.Binned(res));//gets all data from filter Moving average filte
                    DataAccHISTOGRAM.addAll(myFilter.Histogram(DataAccBINNED));
                    for (int i = 0; i < DataAccHISTOGRAM.size(); i++) {
                        boolean insertHistogram = myDb.insertData_HISTOGRAM_BLACKMAN(DataAccHISTOGRAM.get(i));

                    }
                    //handlerHISTO.sendEmptyMessage(0);
                    reachHistogram_BLACKMAN();
            };
            Thread HISTBLACKhread = new Thread(r);
            HISTBLACKhread.start();
            //reachHistogram_BLACKMAN();
        }
        else
            reachHistogram_BLACKMAN();
        //}
        //else
        //handlerHISTO.sendEmptyMessage(0);
        //showMessage("FILTER", "HISTOGRAM");//in case the data already exists deploy graph
        //return;
    }
```

El método usado para alcanzar la actividad es *ReachHistogram_ BLACKMAN()*:

```
public void reachHistogram_BLACKMAN(){//function use to reach the histogram
    Cursor resHistogram = myDb.getAllData_HISTOGRAM_BLACKMAN();//call all of the data base data of table histo
    //Cursor resHistogram = getHistogramDecision();//call all of the data base data of table histogram data ba
    ListIntentHistogram.clear();//clears all the data
    for (resHistogram.moveToFirst(); !resHistogram.isAfterLast(); resHistogram.moveToNext()) {//put the cursor
        // The Cursor is now set to the right position
        ListIntentHistogram.add(resHistogram.getString( i: 1));
    }
    final Intent ReachHistogram = new Intent( packageContext: this, HistogramActivity.class);//declares the list
    final int result = 1;
    ReachHistogram.putStringArrayListExtra( name: "callingActivity", (ArrayList<String>) ListIntentHistogram);
    startActivity(ReachHistogram);

}
```

El siguiente botón a analizar es HIOT, el cual hace el cálculo del histograma para los datos que fueron obtenidos después de presionar el botón IOT, esta característica es común en todos los botones de cálculo de histograma. Su código es:

```
public void OnclickHisNOW_IOT(View v){
    showMessage( title: "Histogram",  Message: "IOT DATA");
    Cursor res = myDb.getAllData3();//call all of the data base data of table raw data

    Cursor res2 = myDb.getAllData_HISTOGRAM_IOT();
    if (res != null&&res.getCount()>0) {//verifies that there is data in the data base
        if (res2.getCount()==0) {

            Runnable r = () → {
                    Log.w(TAG,  msg: "ENTRO EN EL HISTOGRAMA");
                    Cursor res = myDb.getAllData3();//call all of the data base data of table raw data
                    DataAccBINNED.clear();
                    DataAccHISTOGRAM.clear();
                    DataAccBINNED.addAll(myFilter.Binned(res));//gets all data from filter Moving average
                    DataAccHISTOGRAM.addAll(myFilter.Histogram(DataAccBINNED));
                    for (int i = 0; i < DataAccHISTOGRAM.size(); i++) {
                        boolean insertHistogram = myDb.insertData_HISTOGRAM_IOT(DataAccHISTOGRAM.get(i));

                    }
                    //handlerHISTO.sendEmptyMessage(0);
                    reachHistogram_IOT();
            };
            Thread HISTIOThread = new Thread(r);
            HISTIOThread.start();
            //reachHistogram_IOT();
        }
        else
            reachHistogram_IOT();
        //}
        //else
        //handlerHISTO.sendEmptyMessage(0);
        //showMessage("FILTER", "HISTOGRAM");//in case the data already exists deploy graph
        //return;
    }
    else
        showMessage( title: "Error",  Message: "no data in the dataBase");
    return;
```

La función para alcanzar el histograma es *reachHistogram_IOT*:

```
public void reachHistogram_IOT(){//function use to reach the histogram
    Cursor resHistogram = myDb.getAllData_HISTOGRAM_IOT();//call all of the data base data of table histogram
    //Cursor resHistogram = getHistogramDecision();//call all of the data base data of table histogram data ba
    ListIntentHistogram.clear();//clears all the data
    for (resHistogram.moveToFirst(); !resHistogram.isAfterLast(); resHistogram.moveToNext()) {//put the cursor
        // The Cursor is now set to the right position
        ListIntentHistogram.add(resHistogram.getString( i: 1));
    }
    final Intent ReachHistogram = new Intent( packageContext: this, HistogramActivity.class);//declares the list
    final int result = 1;
    ReachHistogram.putStringArrayListExtra( name: "callingActivity", (ArrayList<String>) ListIntentHistogram);
    startActivity(ReachHistogram);

}
```

El botón HMAF, el cual funciona de la misma manera que los anteriores botones de cálculo de histogramas:

```
public void OnclickHisNOW_MAF(View v){
    showMessage( title: "Histogram",  Message: "MAF DATA");
    Cursor res = myDb.getAllDataI();//call all of the data base data of table raw data

    Cursor res2 = myDb.getAllData_HISTOGRAM_MOVING();
    if (res != null&&res.getCount()>0) {//verifies that there is data in the data base
        if (res2.getCount()==0) {

            Runnable r = () → {
                    Log.w(TAG,  msg: "ENTRO EN EL HISTOGRAMA");
                    Cursor res = myDb.getAllDataI();//call all of the data base data of table raw data
                    DataAccBINNED.clear();
                    DataAccHISTOGRAM.clear();
                    DataAccBINNED.addAll(myFilter.Binned(res));//gets all data from filter Moving average fil
                    DataAccHISTOGRAM.addAll(myFilter.Histogram(DataAccBINNED));
                    for (int i = 0; i < DataAccHISTOGRAM.size(); i++) {
                        boolean insertHistogram = myDb.insertData_HISTOGRAM_MOVING(DataAccHISTOGRAM.get(i));

                    }
                    //handlerHISTO.sendEmptyMessage(0);
                    reachHistogram_MAF();
            };
            Thread HISMAFThread = new Thread(r);
            HISMAFThread.start();
            //reachHistogram_MAF();
        }
        else
            reachHistogram_MAF();
    }
    else
        showMessage( title: "Error",  Message: "no data in the dataBase");
    return;
```

Su función para alcanzar la actividad HistogramActivity.java es *reachHistogram_ MAF()*:

```
public void reachHistogram_MAF(){//function use to reach the histogram
    Cursor resHistogram = myDb.getAllData_HISTOGRAM_MOVING();//call all of the data base data of table histog
    //Cursor resHistogram = getHistogramDecision();//call all of the data base data of table histogram data b
    ListIntentHistogram.clear();//clears all the data
    for (resHistogram.moveToFirst(); !resHistogram.isAfterLast(); resHistogram.moveToNext()) {//put the curso
        // The Cursor is now set to the right position
        ListIntentHistogram.add(resHistogram.getString( i: 1));
    }
    final Intent ReachHistogram = new Intent( packageContext: this, HistogramActivity.class);//declares the list
    final int result = 1;
    ReachHistogram.putStringArrayListExtra( name: "callingActivity", (ArrayList<String>) ListIntentHistogram);
    startActivity(ReachHistogram);
```

Los botones WRITE y RESET, no se muestran a profundidad ya que no aportan un conocimiento significativo al trabajo. El botón *WRITE* se encarga de escribir todos los

datos recolectados de la señal, en un archivo CSV, dentro de la memoria del dispositivo. Con esto surgen 4 archivos, el primer archivo tendrá los valores de los 3 ejes y la magnitud total de la aceleración encontrada, el segundo archivo los valores filtrados por "MOVING AVERAGE FILTER", el tercer archivo con valores filtrados por "IOT" y el ultimo archivo tendrá los valores filtrados por "BLACKMAN FILTER". La obtención de la dirección donde se guardaran los datos se ve a continuación:

```
String baseDir = android.os.Environment.getExternalStorageDirectory().getAbsolutePath();
String fileName = "AnalysisData_raw.csv";
String fileName_MAF = "AnalysisData_MAF.csv";
String fileName_IOT = "AnalysisData_IOT.csv";
String fileName_BMF = "AnalysisData_BMF.csv";
String filePath = baseDir + File.separator + fileName;
String filePath_MAF = baseDir + File.separator + fileName_MAF;
String filePath_IOT = baseDir + File.separator + fileName_IOT;
String filePath_BMF = baseDir + File.separator + fileName_BMF;
String extStore = System.getenv( name: "EXTERNAL_STORAGE");
Cursor res = myDb.getAllData4();//gets the data from 3 axis
Cursor res1 = myDb.getAllData();//gets the data from the magnitud acc
Cursor res_MAF = myDb.getAllData1();
Cursor res_IOT = myDb.getAllData3();
Cursor res_BMF = myDb.getAllData2();
File f = new File(extStore );
CSVWriter writer;
CSVWriter writer_MAF;
CSVWriter writer_IOT;
CSVWriter writer_BMF;
```

Una vez iniciados los anteriores valores es posible crear los 4 archivos en formato CSV, cada archivo solo será creado si existen datos nuevos en la base de datos. Finalmente se habla del botón de RESET, en el cual se eliminan todos los valores de las listas y se reinician.

```
public void OnclickReset(View v){
```

1.1.5. 5. DeviceScanActivity.java

La clase java DeviceScanActivity, escanea y revisa que haya algún dispositivo BLE alrededor de él, enumerando en una lista los elementos encontrados, los cuales serán desplegados en una vista *llamada listitem_ device.xml*, donde se podrá ver el nombre de los dispositivos encontrados, en caso de presionar uno de estos, se inicia una conexión con el dispositivo y se revisa que los servicios contenidos concuerden con los que se están buscando. Para esto se usa las UUID's. Esta clase java heredara sus parámetros de listActivity, siendo una actividad tipo lista como se ve a continuación:

```
public class DeviceScanActivity extends ListActivity {
```

Las variables definidas en esta actividad tipo lista son:

```
private LeDeviceListAdapter mLeDeviceListAdapter;
private BluetoothAdapter mBluetoothAdapter;
private boolean mScanning;
private Handler mHandler;

private static final int REQUEST_ENABLE_BT = 1;
// Stops scanning after 10 seconds.
private static final long SCAN_PERIOD = 10000;
```

Las anteriores variables son definidas para manipular las librerías heredadas de BLE, el ejemplo más claro la llamada al *BluetoothAdapter*. En Android si no se crea una instancia del BluetoothAdapter, no se podrá trabajar con dicho dispositivo. El periodo de escaneo es de 10 segundos, después de esto deja de escanear. Una vez definidas las variables necesarias para trabajar, se empieza a analizar las funciones override, las cuales vienen por defecto para este tipo de actividad ListActivity(). Si desea ver las funciones override por defecto, se insertan al presionar botón derecho dentro de esta actividad y seleccionar generar código. Los métodos override de esta función son:

```
@Override
public void onCreate(Bundle savedInstanceState) {
```

Este método es general para toda actividad, será el primer método en ser llamado al crear esta actividad, en donde pondrá la barra de acción con el título respectivo de esta actividad y dará inicio a un handle:

```
getActionBar().setTitle("BLE Device Scan");
mHandler = new Handler();
```

Una vez iniciados estos parámetros, la aplicación debe preguntar si el dispositivo en el cual está siendo desplegada posee tecnologías BLE:

```
// Use this check to determine whether BLE is supported on the device.  Then you can
// selectively disable BLE-related features.
if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
    Toast.makeText( context: this, "BLE is not supported", Toast.LENGTH_SHORT).show();
    finish();
}
```

En caso de no soportar esta tecnología termina la actividad. Finalmente inicia el adaptador de bluetooth, y pregunta si realmente pudo ser iniciado el adaptador:

```
@Override
protected void onResume() {
    super.onResume();

    // Ensures Bluetooth is enabled on the device.  If Bluetooth is not currently enable
    // fire an intent to display a dialog asking the user to grant permission to enable
    if (!mBluetoothAdapter.isEnabled()) {
        if (!mBluetoothAdapter.isEnabled()) {
            Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        }
    }

    // Initializes list view adapter.
    mLeDeviceListAdapter = new LeDeviceListAdapter();
    setListAdapter(mLeDeviceListAdapter);
    scanLeDevice( enable: true);
}
```

Cuando el sistema pasa la fase inicial entra en el segundo método el *onResume()*, donde pregunta si ha sido iniciado el bluetooth. En caso de no haber iniciado se pregunta si quiere hacerlo o terminar la aplicación. Si lo ha hecho inicia el despliegue de la lista al empezar a escanear, la cual hace llamado a una función que se explica más adelante.

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {//crea una
    final BluetoothDevice device = mLeDeviceListAdapter.getDevice(position);//obtiene
    if (device == null) return;
    final Intent intent = new Intent( packageContext: this, DeviceControlActivity.class);
    intent.putExtra(DeviceControlActivity.EXTRAS_DEVICE_NAME, device.getName());
    intent.putExtra(DeviceControlActivity.EXTRAS_DEVICE_ADDRESS, device.getAddress());
    if (mScanning) {
        mBluetoothAdapter.stopLeScan(mLeScanCallback);
        mScanning = false;
    }
    startActivity(intent);
}
```

El siguiente método override es el método *onListItemClick()* el cual es llamado cada vez que se selecciona un dispositivo encontrado por BLE:

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {//crea una
    final BluetoothDevice device = mLeDeviceListAdapter.getDevice(position);//obtiene
    if (device == null) return;
    final Intent intent = new Intent( packageContext: this, DeviceControlActivity.class);
    intent.putExtra(DeviceControlActivity.EXTRAS_DEVICE_NAME, device.getName());
    intent.putExtra(DeviceControlActivity.EXTRAS_DEVICE_ADDRESS, device.getAddress());
    if (mScanning) {
        mBluetoothAdapter.stopLeScan(mLeScanCallback);
        mScanning = false;
    }
    startActivity(intent);
}
```

En caso de seleccionar uno de los dispositivos encontrados se aplican medidas de seguridad con condicionales y se crea un intent para poder comunicar esta actividad con la actividad DeviceControlActivity(). Con esto pasa a ser desplegada con su respectiva vista, para asegurarse de no desperdiciar energía se detiene el escaneo. Si se desea hacer más conexiones, se tiene que evitar parar el escaneo al conectarse a un dispositivo, ya que para poder conectar varios dispositivos hay que conectar uno a la vez. En este punto se manda a la siguiente actividad la información del nombre del dispositivo y su dirección.

Para poder desplegar los dispositivos encontrados se utiliza el código:

```
@Override
public View getView(int i, View view, ViewGroup viewGroup) {
    ViewHolder viewHolder;
    // General ListView optimization code.
    if (view == null) {
        view = mInflator.inflate(R.layout.listitem_device,  root: null);
        viewHolder = new ViewHolder();
        viewHolder.deviceAddress = (TextView) view.findViewById(R.id.device_address);
        viewHolder.deviceName = (TextView) view.findViewById(R.id.device_name);
        view.setTag(viewHolder);
    } else {
        viewHolder = (ViewHolder) view.getTag();
    }

    BluetoothDevice device = mLeDevices.get(i);
    final String deviceName = device.getName();
    if (deviceName != null && deviceName.length() > 0)
        viewHolder.deviceName.setText(deviceName);
    else
        viewHolder.deviceName.setText("Unknown device");
    viewHolder.deviceAddress.setText(device.getAddress());

    return view;
}
```

El cual muestra una lista de los dispositivos encontrados, en caso de no tener nombre el dispositivo simplemente aparece como *Unknown device*. Las configuraciones necesarias para el escaneo son:

```
private void scanLeDevice(final boolean enable) {
    if (enable) {
        // Stops scanning after a pre-defined scan period.
        mHandler.postDelayed(() → {
                mScanning = false;
                mBluetoothAdapter.stopLeScan(mLeScanCallback);
                invalidateOptionsMenu();
        }, SCAN_PERIOD);

        mScanning = true;
        mBluetoothAdapter.startLeScan(mLeScanCallback);
    } else {
        mScanning = false;
        mBluetoothAdapter.stopLeScan(mLeScanCallback);
    }
    invalidateOptionsMenu();
}
```

Donde se hace uso de un handle para detener el escáner en caso de que pase el tiempo definido en *SCAN PERIOD*.

1.1.6. 6. HistogramActivity.java

Una actividad muy sencilla cuya única función es obtener los datos mandados desde DeviceControlActivity con el uso de Intents, llamar a la función dentro de Filter.java, para calcular el valor promedio y la desviación estándar y desplegar estos valores en una gráfica. Además de tener un botón el cual permite regresar a la actividad de DeviceControlActivity.java.

El código de esta aplicación no se explica a fondo por la simpleza de esta y todo lo que contiene ya ha sido abordado en el resto del documento. Esta actividad solo cuenta con métodos override *onCreate* y *onResume* que son indispensables para cualquier actividad. Su método onCreate() es:

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.mav_graph);
    TextMean = (TextView) findViewById(R.id.ViewMean);
    TextStandard = (TextView) findViewById(R.id.ViewStandard);
    graph = (GraphView) findViewById(R.id.graphHistogram);//ma
    graph.getViewport().setXAxisBoundsManual(true);
    graph.getViewport().setMinX(300);
    graph.getViewport().setMaxX(3000);
    graph.getViewport().setScrollable(true);
    GridLabelRenderer glr = graph.getGridLabelRenderer();
    glr.setPadding(32);// change the size of the table on grap

}
```

Mientras su método onResume es:

```
protected void onResume() {
    super.onResume();
    DataIntentHISTOGRAM.clear();
    Intent activityThatCalled = getIntent();
    DataIntentHISTOGRAM.addAll(activityThatCalled.getStringArrayListExtra( name: "callingActivity"));

    graph.removeAllSeries();
    seriesHistogram = new LineGraphSeries<DataPoint>();//calls function from database and makes it ready t
    seriesHistogram.setColor(Color.GREEN);
    double y, x;//initialize variables that will hold the math operations
    //DataAccMAf.addAll(myFilter.MAF1(res));

    x = 0;
    y = 10000;
    StringBuffer data_BLE_DataBase_Raw = new StringBuffer();//if there is data creates a chain of strings
    if (DataIntentHISTOGRAM.size() != 0) {
        //count1 = myDb.getAllData1().getCount();
        for (int i = 0; i < DataIntentHISTOGRAM.size()-2; i++) {
            x = i;
            y = Double.parseDouble(DataIntentHISTOGRAM.get(i));
            //y = Math.sin(x);
            seriesHistogram.appendData(new DataPoint(x, y), scrollToEnd: true, DataIntentHISTOGRAM.size());
        }
        graph.addSeries(seriesHistogram);//shows the graph
        DataMeanStandard.addAll(myFilter.Mean(DataIntentHISTOGRAM));//applies the logic to get the mean ar
        displayDataMean(DataMeanStandard.get(0).toString());//displays the mean
        displayDataStandard(DataMeanStandard.get(1).toString());//displays the standard deviation
    } else
        showMessage( title: "Error", Message: "no data in the dataBase");
    return;

}
```

Mediante estos métodos se puede visualizar el histograma de la señal, ya sea pura o filtrada por IOT, MAF o BMF. Además de sus respectivos valores promedios y desviación estándar.

2. ANEXO B

2.1. Documentación del desarrollo hardware en el módulo secundario wearable - nRF51822

El desarrollo del sistema se programa directamente en el lenguaje de programación nativo C, utilizado en los SDK de Nordic. Es el lenguaje de programación estructurado más popular para crear software de sistemas, construcción de intérpretes, compiladores, editores de texto, incluso aplicaciones. Se trata de un lenguaje de medio nivel, ya que dispone tanto de estructuras comunes de lenguajes de alto nivel, como de construcciones que permiten un control a muy bajo nivel.

En este documento se describe brevemente el código utilizado para programar el prototipo en el dispositivo *NRF51822*. Adicionalmente, en partes del código se agregan comentarios para mantener un orden y describir las instrucciones establecidas.

Se inicia añadiendo un archivo denominado "*includes.h*" en el cual se encuentran todas las librerías que se cargarán en el programa, esto con el fin de tener un código más limpio y ordenado. Enseguida se encuentran las líneas "*#pragma*" que se encarga de alinear correctamente la pila, esto con el fin de ordenar el almacenamiento de variables en la memoria.

A continuación se dispone a describir cada una de las funciones que se utilizan en el programa:

2.1.1. Static void power_manage(void)

Con esta función simplemente ordena al dispositivo, consumir tan sólo la energía que necesita para funcionar.

2.1.2. Int main(void)

Esta es la función principal del programa, donde se determinará toda la lógica que tendrá. Comienza iniciando el LOG del dispositivo para poder escribir en él y tener una especie de bitácora del sistema. Para ejecutarlo se inicia un checkeo del error, dependiendo del "err_code" (código de error) si lo tuviese.

Enseguida se inicializa el módulo TIMER del dispositivo, estableciendo su preescaler, el tamaño de la cola y el tiempo de espera para el planificador, el cual se establece en nulo. Finalmente se especifican e inicializan los leds del dispositivo para su uso posterior.

main.c

```c
#include <includes.h>


#pragma pack(push)  /* push current alignment to stack */
#pragma pack(1)     /* set alignment to 1 byte boundary */
#pragma pack(pop)   /* restore original alignment from stack */

/**@brief Function for doing power management.
 */
static void power_manage(void)
{
    uint32_t err_code = sd_app_evt_wait();
    APP_ERROR_CHECK(err_code);
}

/**
 * @brief Function for application main entry.
 */
int main(void)
{
    uint32_t err_code;
    // Initialize.
    err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    // Start execution.
    NRF_LOG_INFO("\r\n\r\n\r\n\r\n----------------------------------------------\r\n");
    NRF_LOG_INFO("BLE Beacon started\r\n");

    // Initialize timer module.
    APP_TIMER_INIT(APP_TIMER_PRESCALER, APP_TIMER_OP_QUEUE_SIZE, false);

    bsp_board_leds_init();
```

Seguidamente, se inicializan las funciones requeridas para el uso de Bluetooth Low Energy y el acelerómetro. En cuanto a la conexión Bluetooth se encuentran: *ble_stack_init*, *gap_params_init*, *advertising_init*, *gatt_init*, *services_init* y *conn_params_init*; las cuales se detallan más adelante.

```c

    // Initialize BLE
    ble_stack_init();
    gap_params_init();
    advertising_init();
    gatt_init();
    services_init();
    conn_params_init();

    // Initialize accelerometer
    accelerometer_init();

    for (int i = 0; i < 5; ++i) {
        bsp_board_led_on(0);
        nrf_delay_ms(50);
        bsp_board_led_off(0);
        nrf_delay_ms(50);
    }

    advertising_start();

    // Enter main loop.
    for (;; )
    {

        if (NRF_LOG_PROCESS() == false)
        {
            power_manage();
        }
    }
}
```

Para el sensor del acelerómetro se tiene la función *accelerometer_init* que se encarga de inicializar este componente dentro de la placa. Después encontramos un ciclo for, que sencillamente se encarga de encender el LED de la placa con el fin de indicar el inicio del programa, sólo se ejecuta al principio, apenas se conecte la fuente de alimentación.

La función *advertising_start* es la encargada de iniciar el advertising del módulo BLE, para buscar un dispositivo que se quiera emparejar con éste.

2.1.3. Includes.h

La siguiente fracción de código es del archivo includes.h que detalle y carga las librerías necesarias para la correcta ejecución del programa. Entre ellas se destacan las siguientes: *ble.h, ble_ radio_ notification.h, nrf.h, ble_ advdata.h, softdevice_ handler.h, ble_ advertising.h, nrf_ ble_ gatt.h, ble_ app.h, accelerometer_ app.h, ble_ custom_ service.h.*

También se definen las constantes para la inicialización del Temporizador (*Timer*).

includes.h

```
#ifndef _INCLUDES_H_
#define _INCLUDES_H_

#include <stdint.h>
#include <string.h>

#include <nrf_log.h>
#include "nrf_log_ctrl.h"
#include "ble_radio_notification.h"
#include <nordic_common.h>
#include <nrf.h>
#include <app_error.h>
#include <ble.h>
#include <ble_hci.h>
#include <ble_srv_common.h>
#include <ble_advdata.h>
#include <ble_advertising.h>
#include <ble_dis.h>
#include <ble_conn_params.h>
#include <softdevice_handler.h>
#include <app_timer.h>
#include <bsp.h>
#include <nrf_delay.h>
#include <bsp_btn_ble.h>
#include <nrf_ble_gatt.h>
#include "nrf_drv_gpiote.h"
#include <ble_conn_state.h>

#include "ble_app.h"
#include "accelerometer_app.h"
#include "ble_custom_service.h"

#define APP_TIMER_PRESCALER             0                                   /**< Value of the RTC1 PRESC
#define APP_TIMER_MAX_TIMERS            6                                   /**< Maximum number of simul
#define APP_TIMER_OP_QUEUE_SIZE         4                                   /**< Size of timer operation
```

2.1.4. Función ble_stack_init

Se encarga de inicializar el módulo que maneja el SoftDevice. También establece los parámetros para la conexión con Bluetooth Low Energy, chequea la configuración de la memoria RAM del dispositivo, habilita el dispositivo BLE y registra a través del SoftDevice los eventos BLE. Todos ellos cuentan con un chequeo de errores para informar el resultado en caso de haber algún tipo de error en la ejecución de las tareas.

```c
void ble_stack_init(void)
{
    NRF_LOG_INFO("%s:\r\n", (uint32_t)__FUNCTION__);
    uint32_t err_code;
    nrf_clock_lf_cfg_t clock_lf_cfg = NRF_CLOCK_LFCLKSRC;


    // Initialize the SoftDevice handler module.
    SOFTDEVICE_HANDLER_INIT(&clock_lf_cfg, NULL);

    ble_enable_params_t ble_enable_params;
    err_code = softdevice_enable_get_default_config(CENTRAL_LINK_COUNT,
                                                    PERIPHERAL_LINK_COUNT,
                                                    &ble_enable_params);
    APP_ERROR_CHECK(err_code);

    //Check the ram settings against the used number of links
    CHECK_RAM_START_ADDR(CENTRAL_LINK_COUNT,PERIPHERAL_LINK_COUNT);

    // Enable BLE stack.
    err_code = softdevice_enable(&ble_enable_params);
    APP_ERROR_CHECK(err_code);

    // Register with the SoftDevice handler module for BLE events.
    err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);
    APP_ERROR_CHECK(err_code);

}
```

2.1.5. Función gap_params_init

Establece los parámetros para la conexión Gapp del dispositivo Bluetooth, tales como: Modo de conexión, Nombre del dispositivo, tipo de apariencia, mínimo y máximo intervalo de conexión, latencia y tiempo de espera para terminar la conexión.

```c
void gap_params_init(void)
{
    NRF_LOG_INFO("%s:\r\n", (uint32_t)__FUNCTION__);

    uint32_t err_code;
    ble_gap_conn_sec_mode_t sec_mode;
    ble_gap_conn_params_t   gap_conn_params;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                          (const uint8_t *)DEVICE_NAME,
                                          strlen(DEVICE_NAME));
    APP_ERROR_CHECK(err_code);

    err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_UNKNOWN);
    APP_ERROR_CHECK(err_code);

    memset(&gap_conn_params, 0, sizeof(gap_conn_params));

    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency     = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);

}
```

De acuerdo con lo anterior, para este caso se establecen estos parámetros de la siguiente manera:

- *BLE_GAP_CONN_SEC_MODE_SET_OPEN* – Significa que no requiere protección para abrir el enlace
- *DEVICE_NAME = "Thesis"* – Nombre del periférico BLE

- *BLE_APPEARANCE_UNKNOWN* – Significa que tendrá una apariencia desconocida
- *MIN_CONN_INTERVAL = 50* – Intervalo mínimo de conexión, 50ms
- *MAX_CONN_INTERVAL = 70* – Intervalo máximo de conexión, 70ms
- *SLAVE_LATENCY = 0* – Retardo en los paquetes
- *CONN_SUP_TIMEOUT = 400* – Tiempo de espera

2.1.6. Función advertising_init

Es donde se constituyen los parámetros iniciales del advertising, se construyen y determinan los datos de advertising; además de fijar opciones de configuración BLE. A continuación se detallan los más importantes:

ble_app.c

```
132 void advertising_init(void)
133 {
134     NRF_LOG_INFO("%s:\r\n", (uint32_t)__FUNCTION__);
135     uint32_t      err_code;
136     ble_advdata_t advdata;
137     uint8_t       flags = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
138
139     ble_advdata_manuf_data_t manuf_specific_data;
140
141     manuf_specific_data.company_identifier = APP_COMPANY_IDENTIFIER;
142
143     manuf_specific_data.data.p_data = (uint8_t *) m_beacon_data;
144     manuf_specific_data.data.size   = 6;
145
146     // Build and set advertising data.
147     memset(&advdata, 0, sizeof(advdata));
148
149     advdata.include_appearance     = true;
150     advdata.name_type              = BLE_ADVDATA_FULL_NAME;
151     advdata.flags                  = flags;
152     advdata.p_manuf_specific_data = &manuf_specific_data;
153
154     err_code = ble_advdata_set(&advdata, NULL);
155     APP_ERROR_CHECK(err_code);
156
157     // Initialize advertising parameters (used when starting advertising).
158     memset(&m_adv_params, 0, sizeof(m_adv_params));
159
160     ble_adv_modes_config_t options;
161     memset(&options, 0, sizeof(options));
162     options.ble_adv_fast_enabled  = true;
163     options.ble_adv_fast_interval = APP_ADV_INTERVAL;
164     options.ble_adv_fast_timeout  = APP_ADV_TIMEOUT_IN_SECONDS;
165
166     err_code = ble_advertising_init(&advdata, NULL, &options, on_adv_evt, NULL);
167     APP_ERROR_CHECK(err_code);
168 }
```

- *Flags = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE* – Son las banderas de advertising e indica que el dispositivo estará en modo general detectable (GENERAL DISCOVERABLE MODE)
- *APP_COMPANY_IDENTIFIER* – Refiere al identificador de la compañía, en este caso, Nordic Semiconductor ASA
- *include_appearance = true* – Determina si la apariencia será incluida
- *name_type = BLE_ADVDATA_FULL_NAME* Incluye el nombre completo del dispositivo en los datos de advertising

- *ble_adv_fast_enabled = true* – Significa que se encuentra habilitado el modo de advertising rápido

- *ble_adv_fast_interval = APP_ADV_INTERVAL* – Es el intervalo de advertising. Se ha fijado en 1000 ms

- *ble_adv_fast_timeout = APP_ADV_TIMEOUT_IN_SECONDS* – Es el tiempo de espera para dejar el modo advertising. En este caso, está puesto en 0 seg, de esta manera siempre estará en este modo

2.1.7. Función gatt_init

Simplemente inicia el módulo GATT

```
ble_app.c
176 /*GATT Module init*/
177 void gatt_init(void)
178 {
179     NRF_LOG_INFO("%s:\r\n", (uint32_t)__FUNCTION__);
180     ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, gatt_evt_handler);
181     APP_ERROR_CHECK(err_code);
182 }
```

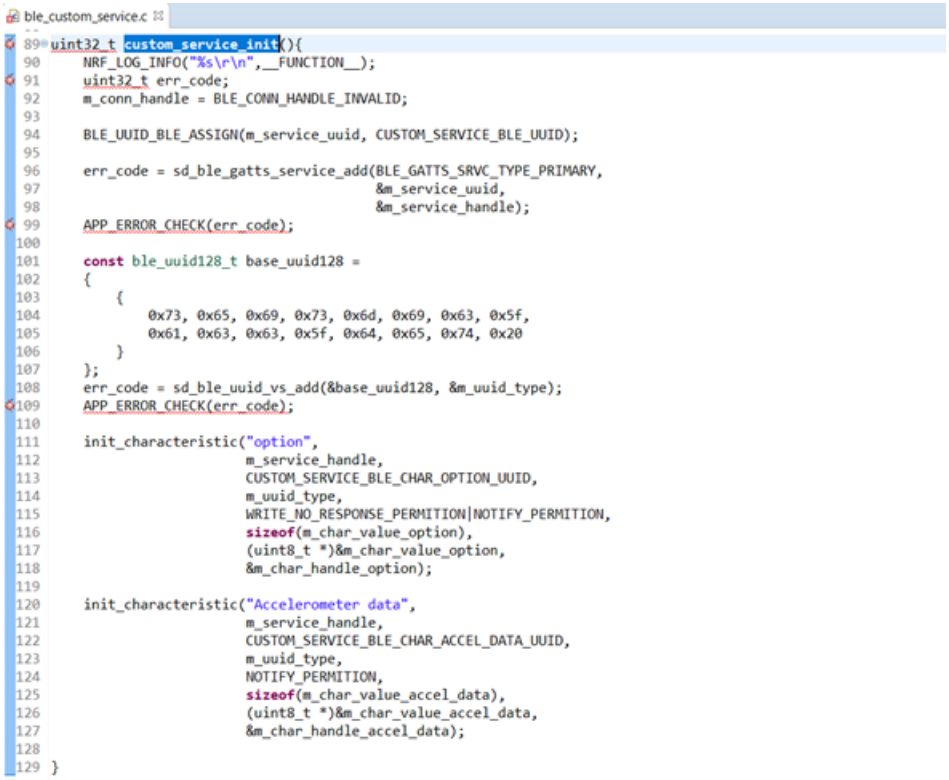
2.1.8. Función services_init

Inicializa los servicios GATT que tendrá el dispositivo BLE. Aquí invoca la función *custom_service_init*.

```
ble_app.c
251 void services_init(){
252     NRF_LOG_INFO("%s:\r\n", (uint32_t)__FUNCTION__);
253     custom_service_init();
254 }
```

2.1.9. Función custom_service_init

Establece los servicios personalizados que tendrá una vez establecida la conexión GATT entre el dispositivo central y el periférico. En este caso, creamos dos servicios denominados *option* y *Accelerometer data*, el primero es un servicio de prueba y el segundo es el que se utilizará para la transferencia de los datos del acelerómetro.



```
ble_custom_service.c
89  uint32_t custom_service_init(){
90      NRF_LOG_INFO("%s\r\n",__FUNCTION__);
91      uint32_t err_code;
92      m_conn_handle = BLE_CONN_HANDLE_INVALID;
93
94      BLE_UUID_BLE_ASSIGN(m_service_uuid, CUSTOM_SERVICE_BLE_UUID);
95
96      err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY,
97                                          &m_service_uuid,
98                                          &m_service_handle);
99      APP_ERROR_CHECK(err_code);
100
101     const ble_uuid128_t base_uuid128 =
102     {
103         {
104             0x73, 0x65, 0x69, 0x73, 0x6d, 0x69, 0x63, 0x5f,
105             0x61, 0x63, 0x63, 0x5f, 0x64, 0x65, 0x74, 0x20
106         }
107     };
108     err_code = sd_ble_uuid_vs_add(&base_uuid128, &m_uuid_type);
109     APP_ERROR_CHECK(err_code);
110
111     init_characteristic("option",
112                         m_service_handle,
113                         CUSTOM_SERVICE_BLE_CHAR_OPTION_UUID,
114                         m_uuid_type,
115                         WRITE_NO_RESPONSE_PERMITION|NOTIFY_PERMITION,
116                         sizeof(m_char_value_option),
117                         (uint8_t *)&m_char_value_option,
118                         &m_char_handle_option);
119
120     init_characteristic("Accelerometer data",
121                         m_service_handle,
122                         CUSTOM_SERVICE_BLE_CHAR_ACCEL_DATA_UUID,
123                         m_uuid_type,
124                         NOTIFY_PERMITION,
125                         sizeof(m_char_value_accel_data),
126                         (uint8_t *)&m_char_value_accel_data,
127                         &m_char_handle_accel_data);
128
129 }
```

Primeramente, se asigna el Handle que manejará los servicios junto con su respectiva UUID. En este caso, por ser un servicio desconocido (personalizado) se fija con el valor de 0x9800. Este valor permite modificar los primeros 4 bytes del valor estándar (BASE UUID) establecido por bluetooth.org, el cual es: 00000000-0000-1000-8000-00805F9B34FB.

Así el servicio tendrá la siguiente UUID: 00009800-0000-1000-8000-00805F9B34FB

También se añaden parámetros como $BLE\_GATTS\_SRVC\_TYPE\_PRIMARY$, el cual define el servicio como primario.

Se observa además una constante que define la UUID del dispositivo BLE en valores hexadecimales: 20746564-5F63-6361-5F63-696D73696573

Ahora, se agregan las características al servicio creado. En la característica *Accelerometer data*, se observa:

- $m\_service\_handle$ – Handle del servicio respectivo
- $CUSTOM\_SERVICE\_BLE\_CHAR\_ACCEL\_DATA\_UUID = 0x9802$ – UUID de la característica
- $m\_uuid\_type = uint8\_t$ – Tipo de UUID de 8 bits
- $NOTIFY\_PERMITION$ – Permiso únicamente de notificación

- *sizeof(m_char_value_accel_data)* – Tamaño de los datos de la característica, es decir, el valor de los datos del acelerómetro. Calcula el tamaño de la estructura que viene dado por 3 valores de cada uno de los ejes del sensor, donde cada valor de un eje viene dado dentro de 2 bytes

- *(uint8_t *)&m_char_value_accel_data* – Puntero del valor de los datos de la característica

- *&m_char_handle_accel_data* – Puntero del Handle, que manejará los datos del acelerómetro

2.1.10. Función conn_params_init

Define los parámetros BLE de conexión. Entre ellos se destacan:

```
ble_app.c
232 void conn_params_init(void)
233 {
234     uint32_t               err_code;
235     ble_conn_params_init_t cp_init;
236
237     memset(&cp_init, 0, sizeof(cp_init));
238
239     cp_init.p_conn_params                  = NULL;
240     cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
241     cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
242     cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
243     cp_init.disconnect_on_fail             = false;
244     cp_init.evt_handler                    = on_conn_params_evt;
245     cp_init.error_handler                  = conn_params_error_handler;
246
247     err_code = ble_conn_params_init(&cp_init);
248     APP_ERROR_CHECK(err_code);
249 }
```

- *first_conn_params_update_delay* – Tiempo desde el inicio del evento (conectar o iniciar la notificación) hasta que se llame al sd_ble_gap_conn_param_update. Establecido en 5000ms

- *next_conn_params_update_delay* – Tiempo entre cada llamada a sd_ble_gap_conn_param_update después de la primera. Valor recomendado 30000ms según las especificaciones Bluetooth

- *max_conn_params_update_count* – Número de intentos antes de abandonar la negociación. Su valor es 3

- *disconnect_on_fail = FALSE* – Puesta en FALSE, una actualización de parámetros de conexión fallida no causará una desconexión automática

- *evt_handler = on_conn_params_evt* – Manejador de eventos a ser llamado para manejar eventos en los parámetros de conexión

- *error_handler = conn_params_error_handler* – Función a llamar en caso de error

2.1.11. Función accelerometer_init

Se fijan los parámetros del Handle del acelerómetro (accelHandle) y se inicializan. Los parámetros más relevantes si identifican a continuación:

accelerometer_app.c

```
int accelerometer_init(void){
    NRF_LOG_INFO("%s\r\n",__FUNCTION__);
    uint32_t err_code;

    /* Setup sensor handle. */
    accelHandle.who_am_i      = LIS2DH12_WHO_AM_I;
    accelHandle.ifType        = 1; /* I2C interface */
    accelHandle.spiDevice     = SPIM0_SS_PIN;
    accelHandle.instance      = 0;
    accelHandle.isInitialized = 0;
    accelHandle.isEnabled     = 1;
    accelHandle.isCombo       = 0;
    accelHandle.pData         = ( void * )&ACCELERO_Data;
    accelHandle.pVTable       = ( void * )&LIS2DH12_Drv;
    accelHandle.pExtVTable    = ( void * )&LIS2DH12_ExtDrv;

    ACCELERO_Data.pComponentData = ( void * )&LIS2DH12_0_Data;
    ACCELERO_Data.pExtData       = 0;
    LIS2DH12_Drv.Init(&accelHandle);

    /* Set high resolution mode. */
    if ( LIS2DH12_SetMode( ( void * )&accelHandle, LIS2DH12_HIGH_RES ) == MEMS_ERROR )
    {
        NRF_LOG_ERROR("LIS2DH12_SetMode");
      return COMPONENT_ERROR;
    }

    err_code = app_timer_create(&m_accel_pulling_data_id, APP_TIMER_MODE_REPEATED, accel_pulling_data_handler);
    APP_ERROR_CHECK(err_code);

    return 0;
}
```

- $who\_am\_i = LIS2DH12\_WHO\_AM\_I$ – Identifica el tipo de dispositivo del acelerómetro
- $ifType = 1$ – Define el canal de comunicación, en este caso, SPI
- $isEnabled = 1$ – Parámetro para encender el sensor
- $isCombo = 0$ – Significa si el sensor viene compuesto de más sensores
- $pData$ – Puntero de los datos
- $pVTable$ – Puntero de la tabla virtual
- $pExtVTable$ – Puntero de la tabla virtual extendida
- Se establece el modo de alta resolución del acelerómetro, a través de la función LIS2DH12_SetMode y se fija el valor LIS2DH12_HIGH_RES

Así mismo, se crea un Timer, el cual se reiniciará cada vez que caduque la función **accel_pulling_data_handler**, que realiza la siguiente acción:

accelerometer_app.c

```
30 static void accel_pulling_data_handler(){
31     //NRF_LOG_INFO("%s\r\n",__FUNCTION__);
32     SensorAxes_t accelAxes32;
33     SensorAxesInt16_t accelAxes16;
34
35     LIS2DH12_Drv.Get_Axes(&accelHandle, &accelAxes32);
36     accelAxes16.AXIS_X = (int16_t)accelAxes32.AXIS_X;
37     accelAxes16.AXIS_Y = (int16_t)accelAxes32.AXIS_Y;
38     accelAxes16.AXIS_Z = (int16_t)accelAxes32.AXIS_Z;
39
40     if (accelerometer_counter_data_char > 2 ){
41         notify_accel_data(accelerometer_data_char);
42         memset(&accelerometer_data_char,0,sizeof(accelerometer_data_char));
43         accelerometer_counter_data_char = 0;
44     }
45     accelerometer_data_char.accelx[accelerometer_counter_data_char] = accelAxes16.AXIS_X;
46     accelerometer_data_char.accely[accelerometer_counter_data_char] = accelAxes16.AXIS_Y;
47     accelerometer_data_char.accelz[accelerometer_counter_data_char] = accelAxes16.AXIS_Z;
48     NRF_LOG_DEBUG("Accel X:%d Y:%d Z:%d\r\n",accelerometer_data_char.accelx[accelerometer_counter_data_char],
49                                             accelerometer_data_char.accely[accelerometer_counter_data_char],
50                                             accelerometer_data_char.accelz[accelerometer_counter_data_char]);
51
52     accelerometer_counter_data_char++;
53 }
```

Primero recupera los valores de los ejes del sensor a través de la estructura del driver LIS2DH12_Drv.Get_Axes, estos valores los extrae en cada uno de los ejes, es decir, en variables creadas al principio: *accelAxes16.AXIS_X, accelAxes16.AXIS_Y y accelAxes16.AXIS_Z*

Enseguida, ejecuta un condicional con el fin de formar un paquete de 3 valores de cada uno de los ejes, por lo tanto, con la ayuda de un contador *accelerometer_counter_data_char* el cual si es mayor a 2, es decir, ya ha recolectado 3 veces el valor del acelerómetro, entonces invoca la función de notificación **notify_accel_data**, la cual envía los datos a través de la conexión BLE.

Las líneas siguientes realizan la asignación de los valores a sus respectivas variables según el índice que viene siendo el contador, ejemplo:

accelerometer_data_char.accelx[accelerometer_counter_data_char] = accelAxes16.AXIS_X

Finalmente, aumenta el valor del contador en 1.

accelerometer_app.c

```
30 static void accel_pulling_data_handler(){
31     //NRF_LOG_INFO("%s\r\n",__FUNCTION__);
32     SensorAxes_t accelAxes32;
33     SensorAxesInt16_t accelAxes16;
34
35     LIS2DH12_Drv.Get_Axes(&accelHandle, &accelAxes32);
36     accelAxes16.AXIS_X = (int16_t)accelAxes32.AXIS_X;
37     accelAxes16.AXIS_Y = (int16_t)accelAxes32.AXIS_Y;
38     accelAxes16.AXIS_Z = (int16_t)accelAxes32.AXIS_Z;
39
40     if (accelerometer_counter_data_char > 2 ){
41         notify_accel_data(accelerometer_data_char);
42         memset(&accelerometer_data_char,0,sizeof(accelerometer_data_char));
43         accelerometer_counter_data_char = 0;
44     }
45     accelerometer_data_char.accelx[accelerometer_counter_data_char] = accelAxes16.AXIS_X;
46     accelerometer_data_char.accely[accelerometer_counter_data_char] = accelAxes16.AXIS_Y;
47     accelerometer_data_char.accelz[accelerometer_counter_data_char] = accelAxes16.AXIS_Z;
48     NRF_LOG_DEBUG("Accel X:%d Y:%d Z:%d\r\n",accelerometer_data_char.accelx[accelerometer_counter_data_char],
49                                             accelerometer_data_char.accely[accelerometer_counter_data_char],
50                                             accelerometer_data_char.accelz[accelerometer_counter_data_char]);
51
52     accelerometer_counter_data_char++;
53 }
```

ble_custom_service.c

```
235 void notify_accel_data(accel_data_char_t _accel_data){
236     // Send value if connected and notifying
237     NRF_LOG_DEBUG("%s\r\n",(uint32_t)__FUNCTION__);
238 
239     m_char_value_accel_data = _accel_data;
240     uint16_t len = sizeof(_accel_data);
241     ble_gatts_hvx_params_t hvx_params;
242     memset(&hvx_params, 0, sizeof(hvx_params));
243     hvx_params.handle = m_char_handle_accel_data.value_handle;
244     hvx_params.type = BLE_GATT_HVX_NOTIFICATION;
245     hvx_params.offset = 0;
246     hvx_params.p_len = &len;
247     hvx_params.p_data = (uint8_t *) &m_char_value_accel_data;
248 
249     sd_ble_gatts_hvx(m_conn_handle, &hvx_params);
250 }
```

De este modo, las funciones *accelerometer_ enable_ notification* y *accelerometer_ disable_ notification*, inician el timer con el intervalo adecuado.

*accelerometer_app.c

```
88 void accelerometer_enable_notification(){
89     app_timer_start(m_accel_pulling_data_id, ACCEL_PULLING_INTERVAL, NULL);
90 }
91 
92 void accelerometer_disable_notification(){
93     memset(&accelerometer_data_char,0,sizeof(accelerometer_data_char));
94     accelerometer_counter_data_char = 0;
95     app_timer_stop(m_accel_pulling_data_id);
96 }
```

3. ANEXO C

3.1. Conceptos Generales

3.1.1. Tipos de dispositivos BLE

Los elementos BLE que no son capaces de crear la comunicación con el clásico Bluetooth, son denominados SINGLE MODE DEVICES, los cuales solo pueden comunicarse entre ellos; estos no son aptos para realizar aplicaciones normales del Bluetooth, como es la transmisión de sonido en instrumentos de audio, o altos niveles de transferencia de datos para envió de archivos; los dispositivos que contienen BLE y Bluetooth, son conocidos como DUAL MODE DEVICES. Las conexiones entre los diferentes tipos de aparatos BLE se muestran en la tabla 2.1:

	Single-Mode	**Dual-Mode**	**Clásico**
Single-Mode	LE	LE	Ninguno
Dual-Mode	LE	Clásico	Clásico
Clásico	Ninguno	Clásico	Clásico

Tabla 2.1. Conexiones entre dispositivos BLE

3.1.2. Optimización de diseño por BLE

Pensada para el bajo dispendio de potencia, cuyo diseño fue guiado por BLUETOOTH SIG, compañía encargada completamente de la estandarización comercial del Bluetooth, yendo desde la capa más básica de categoría física, hasta las más altas de nivel de aplicación; ha diseñado esta nueva tecnología, con la meta de optimizar cada una de las diferentes niveleslos diferentes niveles del proceso, para un consumo de ultra baja energía, ejemplo:

La capa física que contiene los parámetros de radiocomunicación fue optimizada para ser más flexible, demandando menos potencia al transmitir y recibir datos. Las conexiones son más veloces en la capa de conexión.

Se creó el advertising, el cual permite trabajar sin la necesidad de conexiones, lo que reduce el tiempo utilizado entre la conexión y él envió de datos.

Detección rápida de fuentes de ruidos en la banda de funcionamiento de 2.45 GHz, la cual tiene una alta congestión ya que comparte con otros estándares como son BLUETOOTH CLASSIC, IEEE 802.11, IEEE 802.11b, IEEE 802.11g, IEEE 802.11n, y IEEE 802.15.4,

BLE.

Reduce el tiempo consumido al implementar la revisión de errores de bit en los paquetes enviados CRC[1], debido a que BLE no está hecho para largos envíos de información, la duración destinada a estas comprobaciones es bastante inferior.

Menores niveles de potencia transmitida, esto conlleva a que la sensibilidad del receptor debe ser más alta, para poder recibir esos bajos potencias.

La mayor optimización de BLE con respecto al Bluetooth clásico, es que se puede durar cientos de horas de trabajo continuo, mientras que el Bluetooth clásico está pensado para trabajar unas cuantas horas.

3.1.3. Conceptos básicos BLE

Tiempo es energía: El objetivo principal de BLE es utilizar el menor tiempo posible para alguna clase de movimiento, ya sean transmisiones de radio, comprobaciones de envió o recepción de datos; cualquier tipo de tarea realizada por BLE consume energía, por eso toda acción fundamental debe ser comprimida al menor periodo necesario.

Las acciones importantes y repetitivas que son optimizadas para ampliar la vida útil del aparato son:

Descubrimiento de dispositivos visto en la figura 2.1.

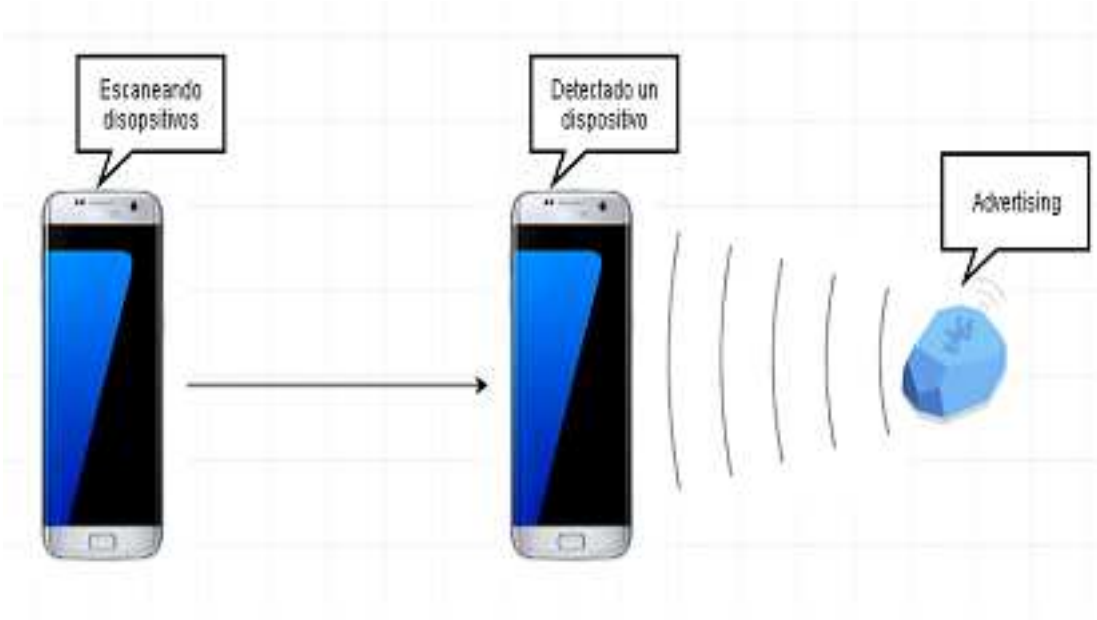


Figura 2.1. Escaneo de dispositivos

[1]CRC: Código de detección de errores para redes digitales.

Conexión de dispositivos visto en la figura 2.2.

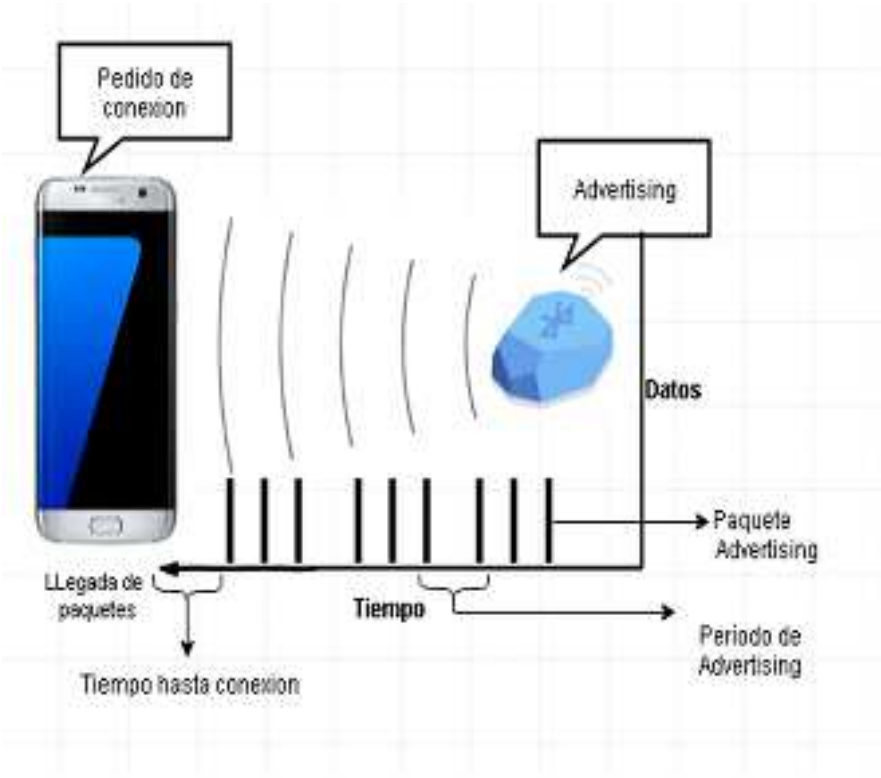


Figura 2.2. Conexión de dispositivos

Estas tareas son optimizadas de la siguiente manera:

La optimización del escaneo de elementos se hace mediante el envío paquetes de advertising por tres canales diferentes cada cierto corto periodo de periodo definido por el diseñador, a diferencia del Bluetooth tradicional el cual es visible todo el tiempo.

Para crear una conexión se debese debe escuchar inmediatamente los paquetes de advertising, en caso contrario se genera un puerto de escucha a fin de descubrir datos de advertising.

El envío de tres paquetes cada período de advertising es utilizado por motivos de robustez y bajo dispendio de energía. utilizando un canal de información por paquete; no se utiliza un canal porque en caso de ser bloqueado toda la radiodifusión seria acabada, ni tampoco más de tres, ya que habría un consumo excesivo de recursos.

Escoger los elementos que escuchan y envían información, ya que el elemento que esté buscando por instrumentos en "advertising" estará en funcionamiento durante un tiempo más largo, debido a esto, se debe asignar esta tarea al artilugio que contenga la mayor cantidad de energía, ejemplo celulares que vengan con "BLUETOOTH Bluetooth 4.0".

La transmisión de paquetes está limitada a 20 bytes, esto obliga a utilizar codificación eficiente, enviando cantidades más extensas de información en un periodo menor de tiempo sin la necesidad de recalibrar parámetros de radio, evitando así el sobrecalentamiento y reduciendo el consumo de energía de estos.

El lapso de transmisión de información no es continuo, en la tecnología BLE, solamente un periodo pequeño de tiempo se hace la radiodifusión, ejemplo: cada 100 ms solo 10 ms está transmitiendo, con esto se logra reducir considerablemente el tiempo de ejecución y así la energía necesaria es mucho menor, extendiendo cuantiosamente la vida útil de la batería.

Memoria de dispositivo: la posesión de una alta capacidad implica no solo a mayores precios, sino que a un consumo elevado de energía, debido al refrescamiento dinámico, lo que conlleva a una menor expectativa de vida de los dispositivos BLE. Por esto los elemento BLE y todas las capas funcionales están diseñadas para tener niveles de almacenamiento bajo; se puede observar esta limitante en la generación de paquetes cortos en la capa de conexión, reduciendo entonces la capacidad necesaria, además ya que los paquetes son basados en el protocolo ATT, se vuelve menos complejo el manejo de datos al no requerir ningún tipo de información de estado entre transacciones, a diferencia de Bluetooth clásico, donde se tiene varios protocolos con el objetivo de realizar diferentes acciones, así se disminuye el número de protocolos y la memoria requerida a fin de contenerlos, obviando cabeceras y por defecto disminuyendo el dispendio de potencia.

Plataformas soportadas: BLE está disponible en las plataformas de las versiones listadas a continuación[17].

1. iOS5+ (iOS7+ preferred)
2. Android 4.3+ (numerous bug fixes in 4.4+)
3. Apple OS X 10.6+
4. Windows 8 (XP, Vista and 7 only support Bluetooth 2.1)
5. GNU/Linux Vanilla BlueZ 4.93+

Arquitectura de BLE: la arquitectura BLE está dividida en tres partes, conocidas como controller, host y applications; controller, se encuentra en la capa física, es capaz de recibir y transmitir información vía radiodifusión, con las característica de entender los datos a nivel eléctrico en las señales captadas; el host es una pila software encargada del manejo de servicios sobre radio y la comunicación entre dos o más dispositivos BLE; finalmente application se encarga del uso del host y controller para crear casos de uso; la arquitectura se puede observar en la figura 2. 3:

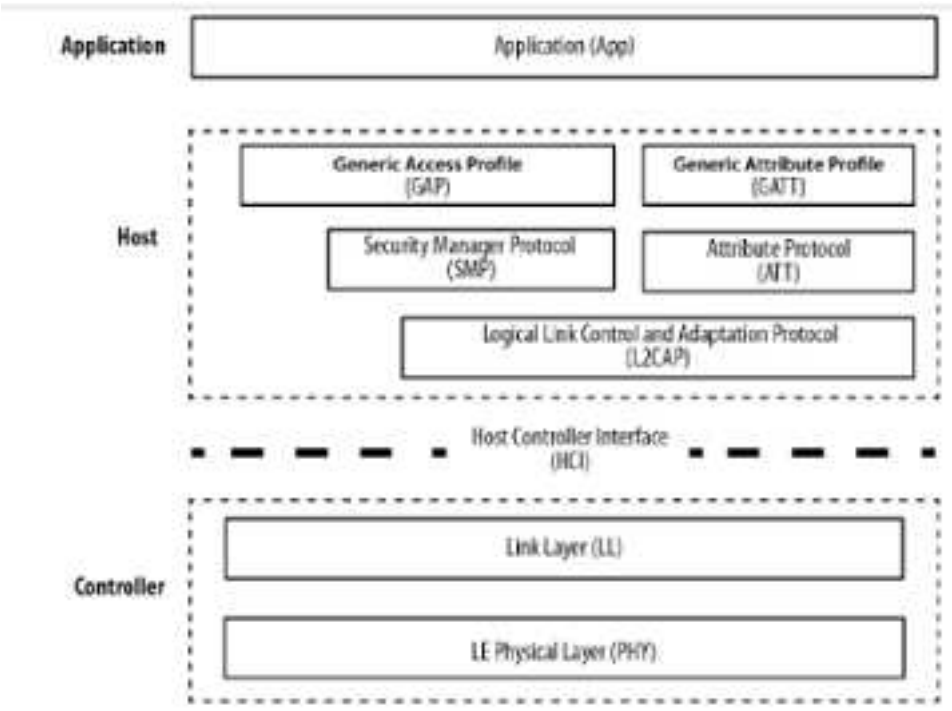


Figura 2. *3* Arquitectura de BLE *[18]*

ATT: Protocolo el cual define un conjunto de reglas para poder acceder y guardar la información de los dispositivos periféricos, estos datos se deben almacenar en un servidor de atributos llamado attributes, accesible mediante un attribute client, con opciones de escritura y lectura, los cuales permiten al cliente conocer las características; en ATT se definen seis tipos diferentes de mensajes los cuales son:

1. Petición desde el cliente al servidor.
2. Respuesta del servidor para responder la petición del cliente.
3. Comandos del cliente al servidor sin respuesta.
4. Notificaciones desde el servidor al cliente sin necesidad de confirmación.
5. Indicaciones mandadas desde el servidor al cliente.
6. Confirmaciones enviadas desde el cliente al servidor para afirmar una indicación

Los atributos son bits de información, los cuales poseen su respectiva handle[2], el cual se encarga de la identificación de la característica específica type[3] y value, donde la información será recolectada o insertada, ejemplo: un servicio de temperatura, donde la palabra temperatura es type, el valor en Celsius se almacena en value, y el handle sería la dirección de memoria. Todo esto es definido por el diseñador cuando se crean los servicios, o en

[2]Handle: dirección y denominación del atributo
[3]Type: determina el tipo de información que se ha guardado

caso de ser un servicio preestablecido en BLE es establecido por SIG.

Este protocolo permite envío de mensajes y manejo de información con condiciones obligatorias para cada atributo, ejemplo: algunos atributos pueden estar en modo lectura, o incluso permisos para dejar a clientes específicos con diferentes niveles de acceso de datos.

El uso del protocolo ATT es debido a su bajo nivel de requisito de memoria. Ya que funciona sin la necesidad de estados.

GAP: acrónimo que representa GENERIC ACCESS PROFILE, encargado del control de conexiones en advertising para BLE, se encarga de que los elementos BLE sean visibles a cualquier artilugio que esté escaneando, además de determinar cómo se crea la conexión y como se relacionan estos dispositivos.

Ubicado superiormente al protocolo ATT, define como el dispositivo descubre, conecta y presenta información útil al usuario, ejemplo: nombre del elemento wearable, empresa desarrolladora y el paquete de envió de datos; donde se pueden almacenar hasta treinta un bytes de información, además de introducir relaciones permanentes entre elementos, denominadas como "bonding".

Roles de dispositivos GAP: Los roles principales establecidos en GAP y configurados en el "ADVERTISING" son:

Dispositivos periféricos: debe ser pequeño, de bajo consumo de energía, poseer escasos recursos, que sea capaz de conectarse a un dispositivo central mucho más poderoso; ejemplo de dispositivos periféricos: monitores cardiacos, medidores de actividades físicas, con sensores de proximidad o inclusive un beacon como se observa en la figura 2.4.

Figura 2.4. Dispositivo periférico

Dispositivos centrales: Son por lo general dispositivos móviles, tabletas o computadores; con el requisito principal de poseer mucho más energía y memoria disponible que los periféricos, como visto en la figura 2.5.

Figura 2.5. Dispositivo Central

Proceso de advertising: comienza con el GAP periférico estableciendo un periodo de advertising, cada vez que este pase, el dispositivo mandara la señal encapsulada en paquetes de advertising, entre más largo sea el intervalo mayor energía será ahorrada, pero se perderá eficiencia en términos de velocidad al contestar. El paquete de advertising tiene un máximo de treinta un bytes, en caso de que el GAP central esté interesado en datos extra, este puede pedir otros treinta un bytes de información conocido como respuesta de escaneo, en la figura 2. 6 se observa el proceso.

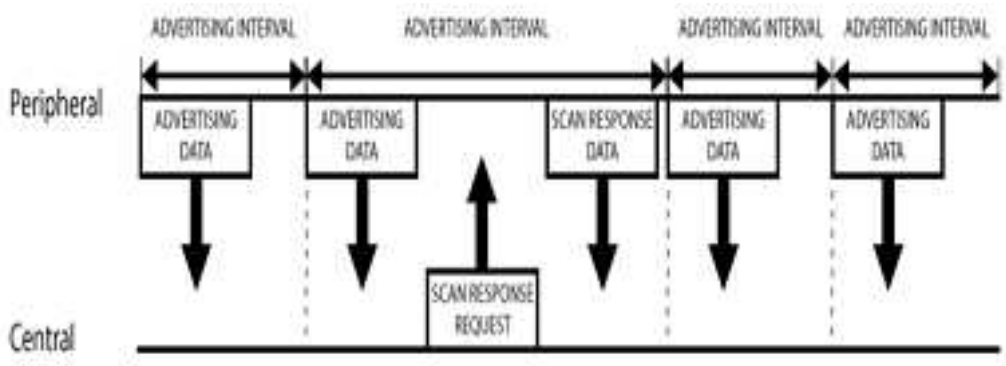


Figura 2.6. Proceso de "advertising" con "scan response request"

Topología de red radiodifusión: las propiedades de advertising proporcionadas por BLE, ofrecen ventajas con respecto al enlace entre elementos Bluetooth, estas características permiten mandar información de un elemento GAP periférico a varios GAP centrales, lo cual no es posible en el BLE tradicional en modo conexión ni en el Bluetooth clásico; utilizando el scan response data, es plausible enviar paquetes de datos a diversos dispositivos centrales, una similar arquitectura es utilizada comúnmente por los Beacons y puede ser observada en la figura 2.7.

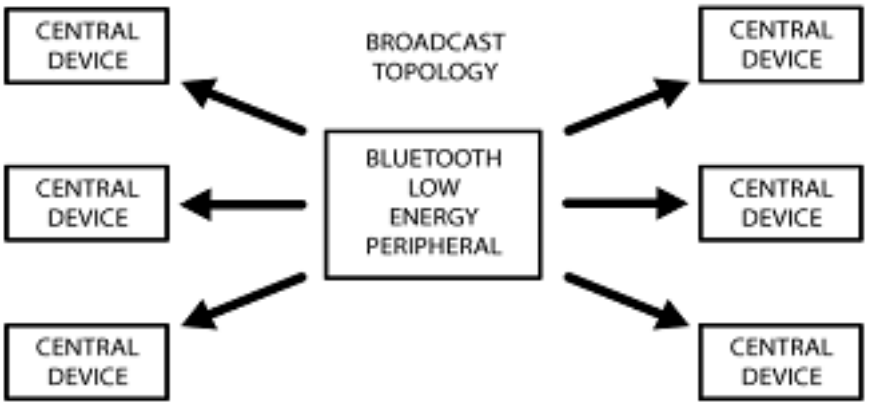


Figura 2.7. Topología de multiconexión en advertising

GATT: Acrónimo utilizado para "GENERIC ATTRIBUTE PROFILE", define como dos dispositivos BLE, pueden establecer una comunicación con transferencia de información bidireccional, mediante la utilización de conceptos de servicios y características, junto al protocolo ATT, el cual se encarga del almacenamiento de características y servicios en tablas, con números de identificación de dieciséis bytes, para servicios predeterminados y ciento veintiocho bytes para servicios personalizados, aunque al realizar las pruebas se ha determinado que utilizar una UUID de dieciséis bytes para un servicio personalizado no afecta, en nada el rendimiento del sistema, en GATT, el modo de conexión de GATT puede ser visto en la figura 2.8:

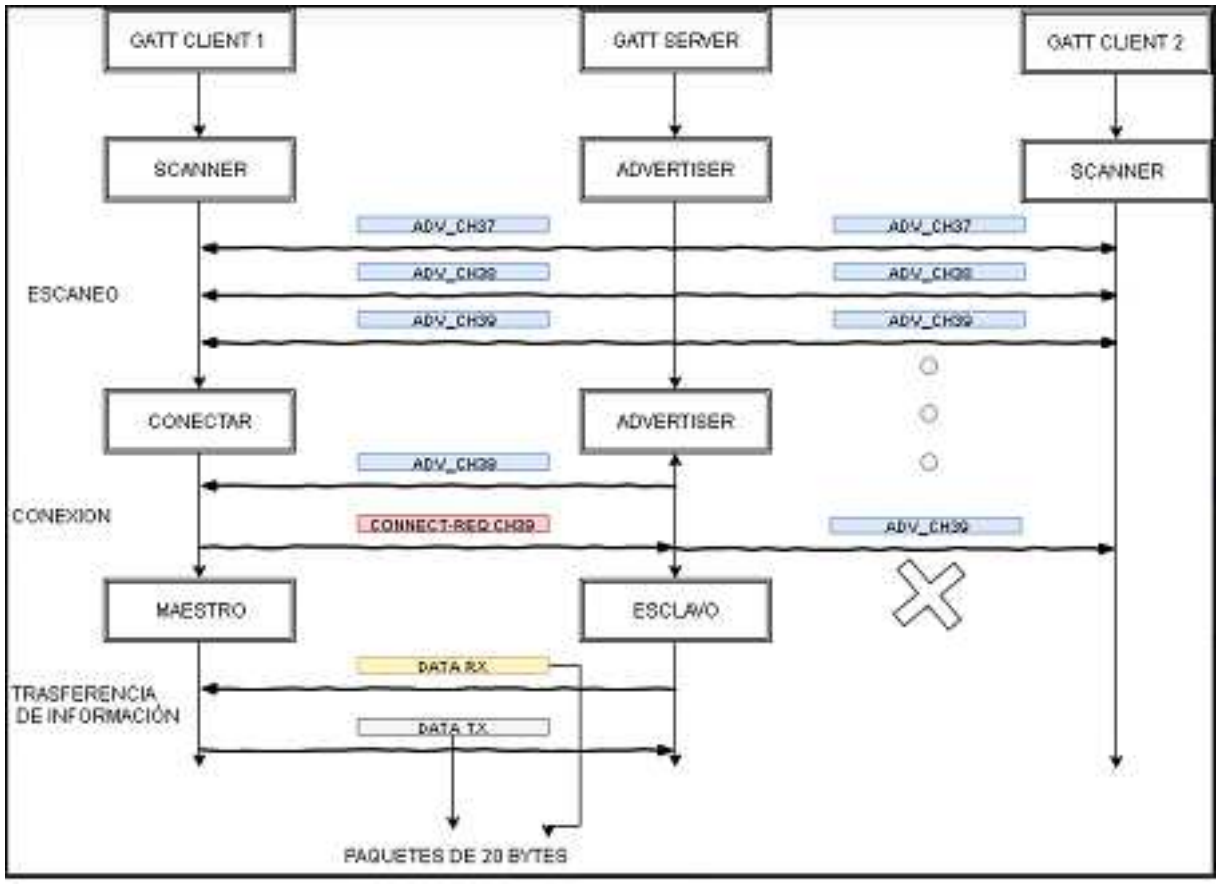


Figura 2.8. Diagrama de conexión GATT

Topología de conexión de red: a continuación se presenta la figura 2.9 el cual demuestra cómo funciona el enlace de BLE en GATT. Mientras un periférico solo puede ser conectado a un elemento central, un wearable central es capaz de ser vinculado a varios dispositivos periféricos; una vez creada la conexión entre el dispositivo central y periférico se podrá mandar información bidireccional.

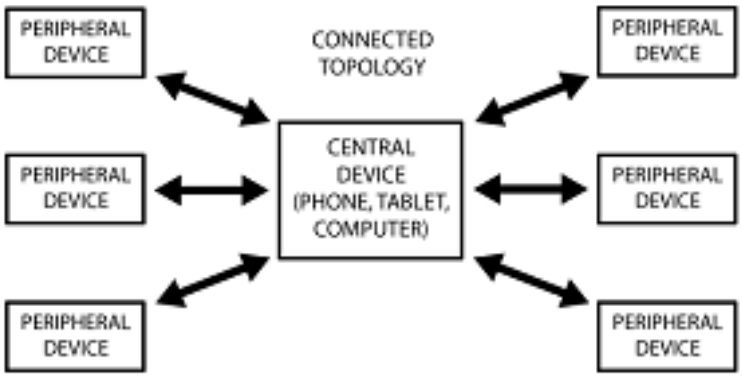


Figura 2.9. Diagrama de topología de conexión en GATT

Transacciones GATT: en GATT las comunicaciones son realizadas entre el cliente y el servidor, todo elemento periférico es conocido como servidor GATT, el cual contiene

una tabla de servicios y características definidas con el protocolo ATT, mientras que el dispositivo central, es llamado cliente GATT, ejemplo: celulares, computadores, tabletas; el cual envía una serie de peticiones de enlace al servidor GATT, de esta forma se debe entender que todas las conexiones serán iniciadas por el cliente GATT, cada vez que se inicie una vinculación se propondrá un intervalo de conexión desde el servidor GATT y el cliente intentara reconectarse cuando se cumpla ese periodo, comprobando si existe información disponible; se observa el intercambio de información GATT en la figura 2.10.

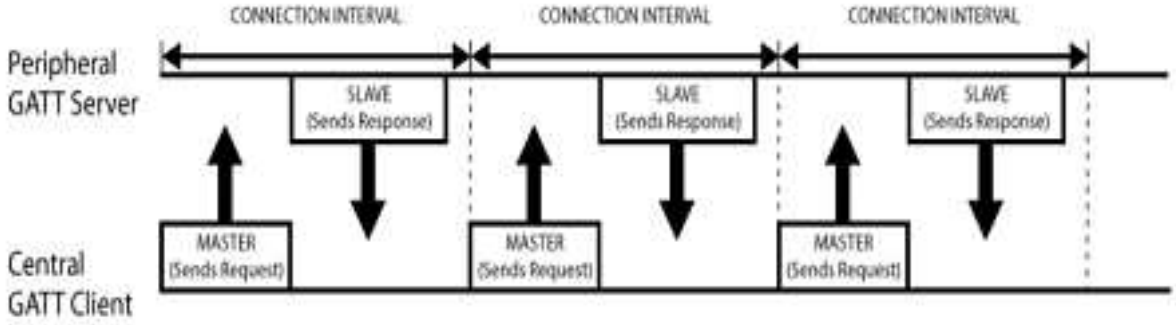


Figura 2.10. Diagrama de intercambio de conexión GATT

Servicios y características: Todas las transacciones GATT, en BLE, son basadas en perfiles, servicios y características; la forma en que estos son compuestos se ve en la figura 2. 11.

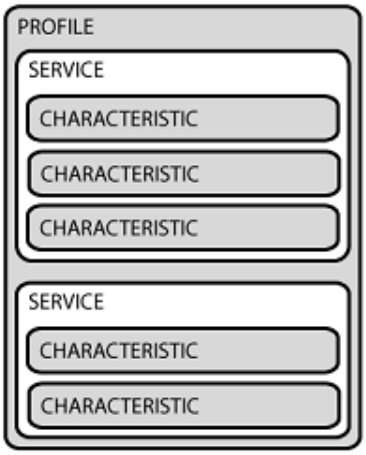


Figura 2.11. Diagrama de perfiles

Perfiles: Existe de manera abstracta en BLE, definido como la colección predefinida de servicios y características, la cual ha sido diseñada por SIG como perfil predefinido, o por el diseñador como perfil personalizado, un ejemplo de un servicio predeterminado es el Heart Rate Profile; el perfil de aceleración, no ha sido diseñado por SIG; su implementación fue hecha como un perfil personalizado; una lista completa de los servicios y características ofrecidas para BLE, puede ser encontrada en su documentación.

Servicios: cada servicio es utilizado para descomponer los datos en entidades lógicas, la cual contiene, varios trozos de información conocidos como características; cada servicio puede tener uno o más características, la forma de identificar cada servicio es mediante el uso de UUIDS, los cuales sirven como identificadores únicos compuestos de dieciséis bytes para servicios predefinidos y 128 bytes para servicios personalizados.

Características: las características son el nivel más bajo de información de BLE GATT, definidas, como tipos de atributos que contienen un único valor lógico.

Cada característica contiene un UUID diferente, que puede ser de dieciséis o ciento veintiocho bytes, las características personalizadas tienen ciento veintiocho bytes de información, mientras que las predefinidas solo dieciséis bytes; cuando se trabaja con los periféricos de BLE, se tiene que programar los perfiles con sus respectivos servicios y características; toda característica tiene un descriptor, en donde se definirá, que tipo de acciones se realizara; el máximo número de bytes que puede ser enviados en value son veinte bytes y el mínimo son un byte, las acciones en la característica pueden ser leer, escribir, notificar o unir dispositivos, una característica personalizada se ve en la figura 2.12.

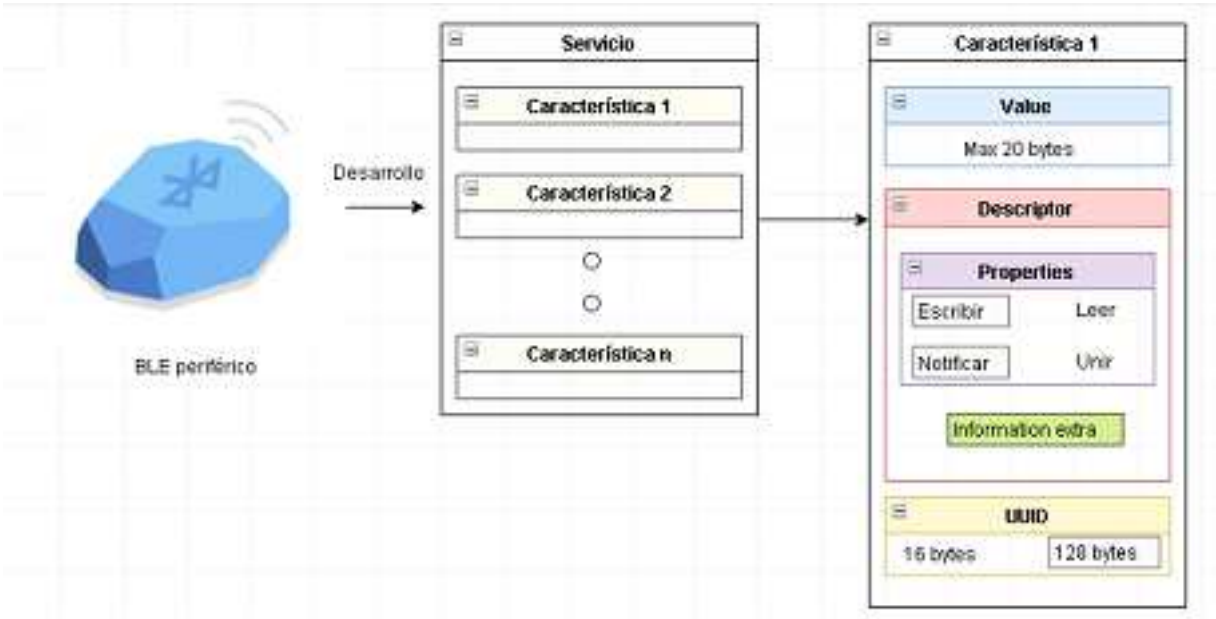


Figura 2.12. Característica personalizada con propiedades de escritura y lectura contínua

3.1.4. Funcionamiento del acelerómetro

Los acelerómetros, ver figura 2.13, son dispositivos electromecánicos los cuales censan tanto fuerzas dinámicas como estáticas, ejemplo de fuerzas estáticas: la gravedad como constante en la tierra, mientras que en las fuerzas dinámicas entran las vibraciones y movimientos humanos.

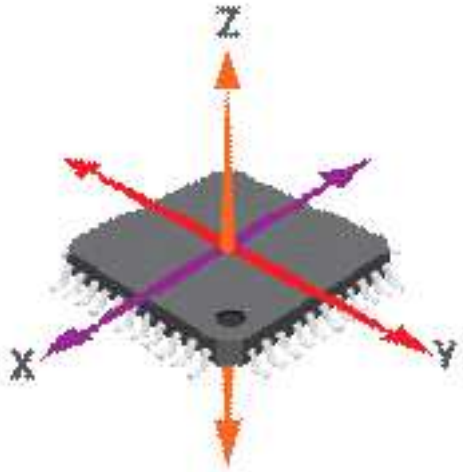


Figura 2.13. Ejes de un acelerómetro

Los acelerómetros son capaces de medir aceleración en uno, dos o tres ejes; debido al bajo costo de producción los acelerómetros de 3 axis son más comunes.

En general un acelerómetro contienen placas internas capacitivas, algunas de las cuales se hallan fijadas, mientras que otras se encuentran amarradas a resortes minúsculos que se pueden mover internamente en el momento que la fuerza de la aceleración ejerce efecto sobre ellas, cuando las placas se muevan la capacitancia entre ellas cambia, de estos cambios en la capacitancia es que se determina la aceleración, otros acelerómetros son construidos a base de materiales piezoeléctricos, ver figura 2. 14, los cuales son pequeñas estructuras de canales que sacan corrientes eléctricas dependiendo de la fuerza mecánica ejercida sobre ellos.

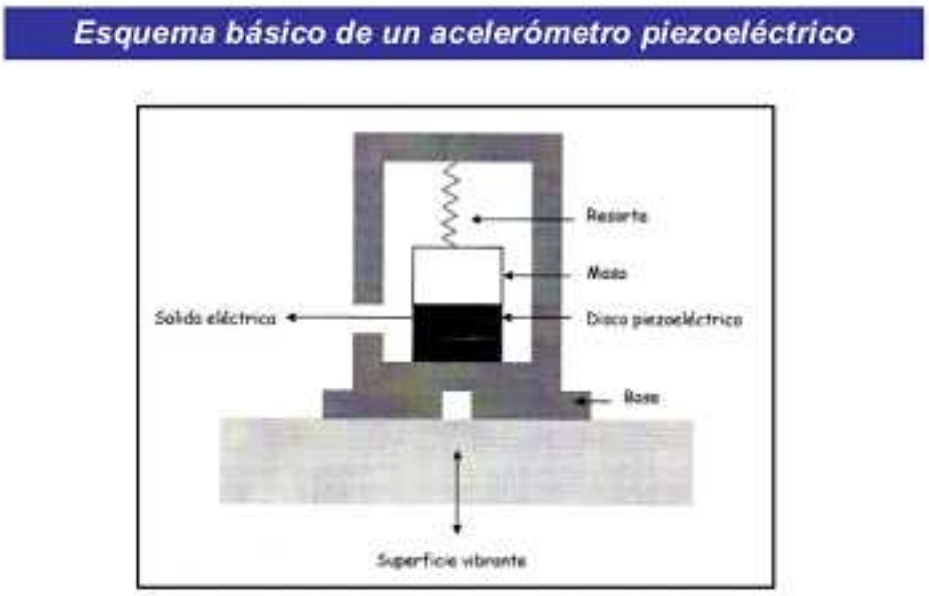


Figura 2.14. Acelerómetro piezoeléctrico

3.1.5. Conexión de un acelerómetro

La mayoría de los acelerómetros necesitan una conexión básica con una batería y enlace de transferencia de información, ver figura 2. 15, se recomienda leer el DataSheet para verificar las especificaciones del acelerómetro, también es importante en caso de ser digital, revisar los drivers proporcionados por el fabricante.

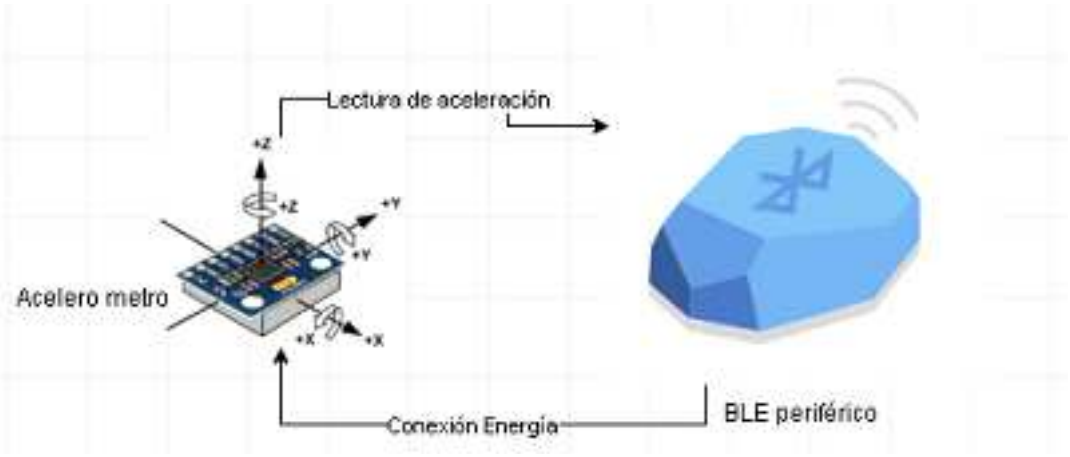


Figura 2.15. Conexión de acelerómetro

3.2. ACELERÓMETRO

3.2.1. Interfaz de conexión

Los acelerómetros se pueden comunicar por comunicaciones análogas, digitales o por interfaz de modulación de pulsos:

1. Los acelerómetros con comunicación de interfaz análoga muestran los cambios de la aceleración mediante variaciones de voltaje, estos valores por lo general varían entre tierra y la fuente provista, un ADC dentro del micro controlador después puede ser usado para leer estos valores, su ventaja es que son más baratos comparados con su versión digital.

2. Acelerómetros con una interfaz digital pueden comunicarse mediante los protocolos de comunicación SPI o I2C, estos tienden a ser más funcionales y tienen menor sensibilidad al ruido, de las 3 maneras de comunicación de acelerómetro esta fue la escogida.

3. Los acelerómetros que traspasan información mediante modulación de pulso PWM, generan ondas cuadradas, con un periodo conocido, pero tienen un ciclo de respuesta que varía con la aceleración.

3.3. FILTROS DIGITALES

3.3.1. Valor promedio y desviación estándar de señales digitales

El valor medio o DC en la electrónica, denominado como u, determina el promedio total de la señal, calculado al sumar todas las muestras de la señal y dividiéndola por el tamaño completo con la fórmula:

$$u = \frac{1}{N} * \sum_{i=0}^{N-1} Xi$$

u: valor medio
N: número de muestras
Xi: Señal

El valor de la desviación estándar es una forma más compleja de análisis de señales, útil cuando la señal no sigue un patrón específico como una onda seno o coseno, sino que tiene una naturaleza aleatoria denotado por σ, la cual se puede hallar sumando los valores absolutos de las desviaciones de cada una de las muestras por separado con el valor medio al cuadrado y dividiendo por N-1.

$$\sigma^2 = \frac{1}{N-1} * \sum_{i=0}^{N-1} 1*(Xi - u)^2$$

La desviación estándar describe que tanto varía la señal con respecto a su valor medio y en electrónica se lo llama valor AC.

En algunos casos el valor medio puede representar la señal original, mientras que la desviación estándar significa la señal de ruido, estos parámetros son representados por SNR y CV.

$$SNR = u \ / \ \sigma$$

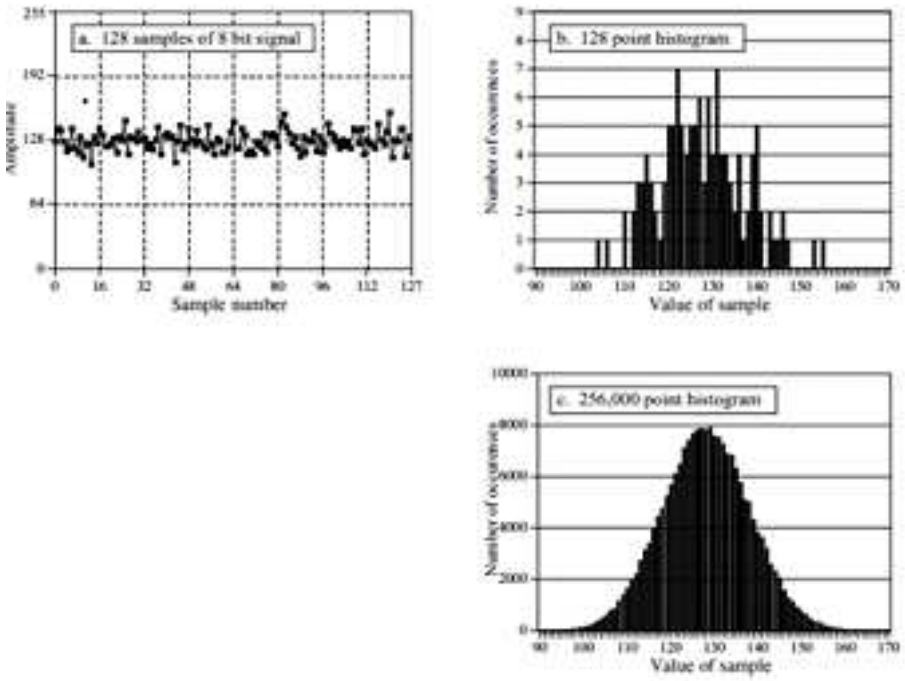
$$CV = (\sigma \ / \ u)$$

Según las ecuaciones anteriores se observa que entre mejor sea la señal más alto sea el SNR y menor sea el cv.

ESTADÍSTICA: ciencia de interpretar información numérica.
PROBABILIDAD: herramienta para entender los procesos que generan de señales.

3.4. HISTOGRAMA

Los histogramas son una representación gráfica de las muestras obtenidas y el número de ellas, ejemplo: un convertidor ADC de ocho bits, donde se capturan doscientos sesenta mil valores, para el cálculo de este histogramas, no es necesario analizar todas los datos, se puede tomar una ventana de ciento veintiocho muestras y analizarla, en el cual cada valor tiene una probabilidad de ser una de las doscientos cincuenta seis posibilidades, con valores desde cero hasta doscientos cincuentaicinco, en esta forma el histograma representa gráficamente las muestras estudiadas; lo anterior se ve reflejado en la figura 2.20.



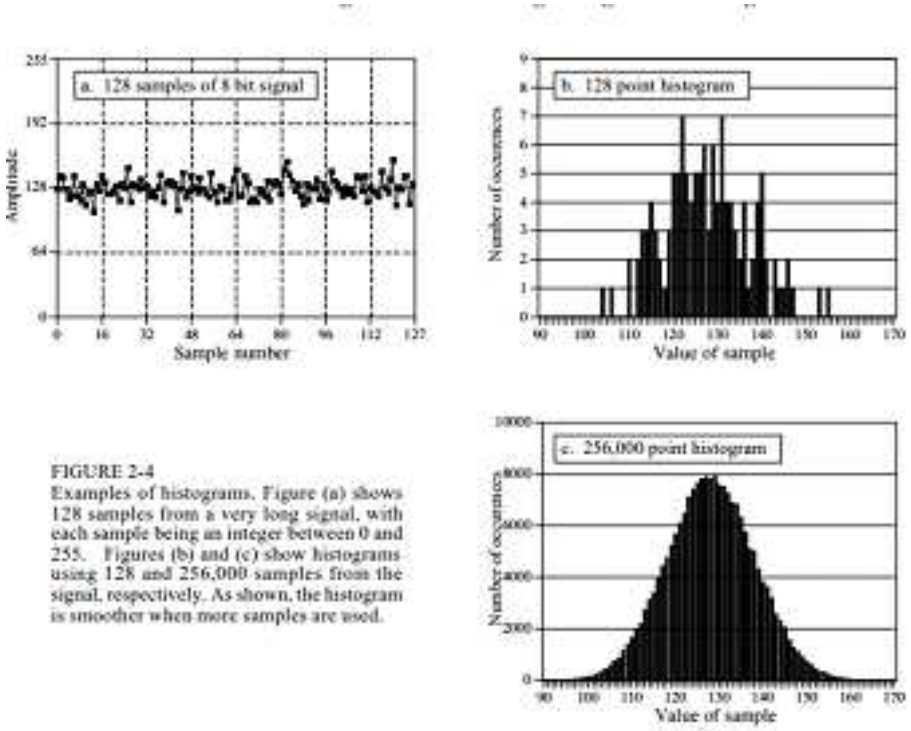


FIGURE 2-4
Examples of histograms. Figure (a) shows 128 samples from a very long signal, with each sample being an integer between 0 and 255. Figures (b) and (c) show histograms using 128 and 256,000 samples from the signal, respectively. As shown, the histogram is smoother when more samples are used.

Figura 2.20. Histogramas obtenidos de la guía de la ingeniería para DSP, Figura a. Muestra 128 muestras de una señal larga con valores entre 0 y 255. Figura b y c. Muestran histogramas usando 128 y 256.000 muestras de la señal.

Se observa que entre más muestras se utilicen mejor será el histograma.

Si se contempla la figura final se observa la imagen al usar todas las muestras, demostrando que entre más valores mejor será el histograma de la señal, para poder calcular la el histograma, denotado por H, se debe crear un arreglo de datos con cada posible valor de la señal, el cual será graficado contra el número de veces que ha ocurrido, de esto se puede concluir que la suma de todos los valores del histograma tienen que ser igual a los puntos en la señal.

$$N = \sum_{i=0}^{M-1} Hi$$

N: número de muestras en la señal.
M: número de puntos en el histograma
Hi: histograma

La función del histograma y la razón de ser utilizada es que permite calcular el valor medio y desviación estándar para grandes cantidades de datos, utilizando las formulas:

$$u = \frac{1}{N} * \sum_{i=0}^{M-1} i * Hi$$

$$\sigma^2 = \frac{1}{N-1} * \sum_{i=0}^{M-1} 1*(i-u)^2 * Hi$$

Las anteriores ecuaciones han sido diseñadas para trabajar con un numero de datos finitos del histograma, en caso de querer trabajar con parámetros infinitos no se llamará más un histograma sino se conocerá como PMF (Probability Mass Function), que sería equivalente a la señal sin ruido. En caso de que las muestras de la señal sean decimales se tendrá que aplicar BINNING para evitar así una saturación por exceso de valores.

Las anteriores ecuaciones han sido diseñadas para trabajar con un numero de datos finitos del histograma, en caso de querer trabajar con parámetros infinitos no se llamarallamará más un histograma sino se conocerá como PMF (Probability Mass Function), que sería equivalente a la señal sin ruido.

511749804 La mayor debilidad de los histogramas, aparece cuando el número de posibles valores que pueda tomar una muestra sea mucho mayor que el número de muestras, en donde se encontrara un error; el anterior caso pasara siempre para valores decimales; para solucionar este problema se llama se empleara el método.

3.4.1. Binned Signals

Un problema común al utilizar histogramas es que se requiere de muestras enteras, lo cual no suele cumplirse en la realidad, es común encontrarse números de punto flotante, lo que hace que el histograma tenga demasiados valores posibles, llegando a calcular millones de posibles posiciones de la señal muestreada, lo que implica una pesada carga computacional, la solución 511749805BINNED SIGNALS, (ver figura 2.21), plantea una serie de rangos en la cualdonde deben estar contenidoas todoas loas posibles valores que tome la señal capturada, cada uno de estos valores entonces será posicionado en el rango en que caiga, y eseéste rango se tomaráa como una cuenta, logrando así, solucionarndo el problema de un exceso de valores.511749805aadmin511749805173461648 4Trate de organizar un poco la redacción de esta parte, pero igual revisen por favor si no se perdió la idea que traían previamente.

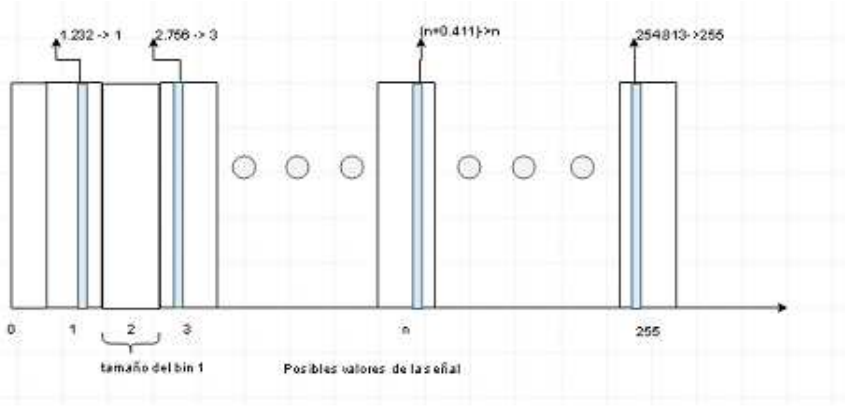


Figura 2.21. Señal con 256 posibles valores y 256 bins, con un tamaño de +-0.5 alrededor de cada entero.

3.4.2. Parámetros de calidad

Las señales digitales al ser procesadas, enviadas y transformadas son afectadas por una u otra forma de ruido probabilístico o digital, en este contexto los parámetros de precisión y certeza son de utilidad, ya que miden, estiman o predicen ciertos eventos, ejemplo: la consistencia de la medición de los sensores sobre la aceleración; la exactitud y la certeza son valores que permiten determinar la cercanía de la señal a los resultados reales mediante el uso de métodos determinísticos y cualitativos como los histogramas.

Los anteriores procesos, pueden dar como resultado dos tipos de errores comunes, en uno el primero de los casos, la media estáa movida del centro, a esto se le denominadice certeza. Yy la segunda es queen el segundo caso, las mediciones individuales no estén de acuerdo unas con otras, lo cual se verá reflejado en la anchura de la función, a esto se le llama precisión, que podrá ser obtenida con la desviación estándar, SNR o la CV.

Cuando los sistemas tienen una buena certeza pero una mala precisión, el histograma estará centrado en la media con una desviación estándar demasiado ancha, lo que implica que el valor total entero será correcto pero cada muestra individual tendrá una pobre credibilidad, en este tipo de casos sirve, la aplicación de filtros como Average Moving Filter para reducir la falta de precisión, de esa manera se garantiza la repetitividad del experimento; esto es producido por ruido aleatorio, mientras que errores que sean dados por la certeza, son generados por errores sistemáticos, que son repetidos igualmente en cada muestra del experimento, y son resueltos mediante mecanismos de calibración, lo que resultará siempre en resultados equivocados alejados del valor verdadero.

3.4.3. Filtros pasa baja

Los filtros pasan baja o filtros kernel, ver figura 2. 23, permiten el paso frecuencias por debajo de la frecuencia de corte conocida como Fc, en caso contrario se generara un aplacamiento que impedirá la contaminación por altas frecuencias, como se observa en

la figura siguiente, la típica forma de los filtros pasa baja, es compuesta de una serie de puntos discretos finitos, los cuales al aplicarles convolución con la señal, generan atenuación por encima de la frecuencia de corte especificada, estos filtros también son llamados suavizadores porque eliminan los altos picos que se encuentra en la información obtenida.

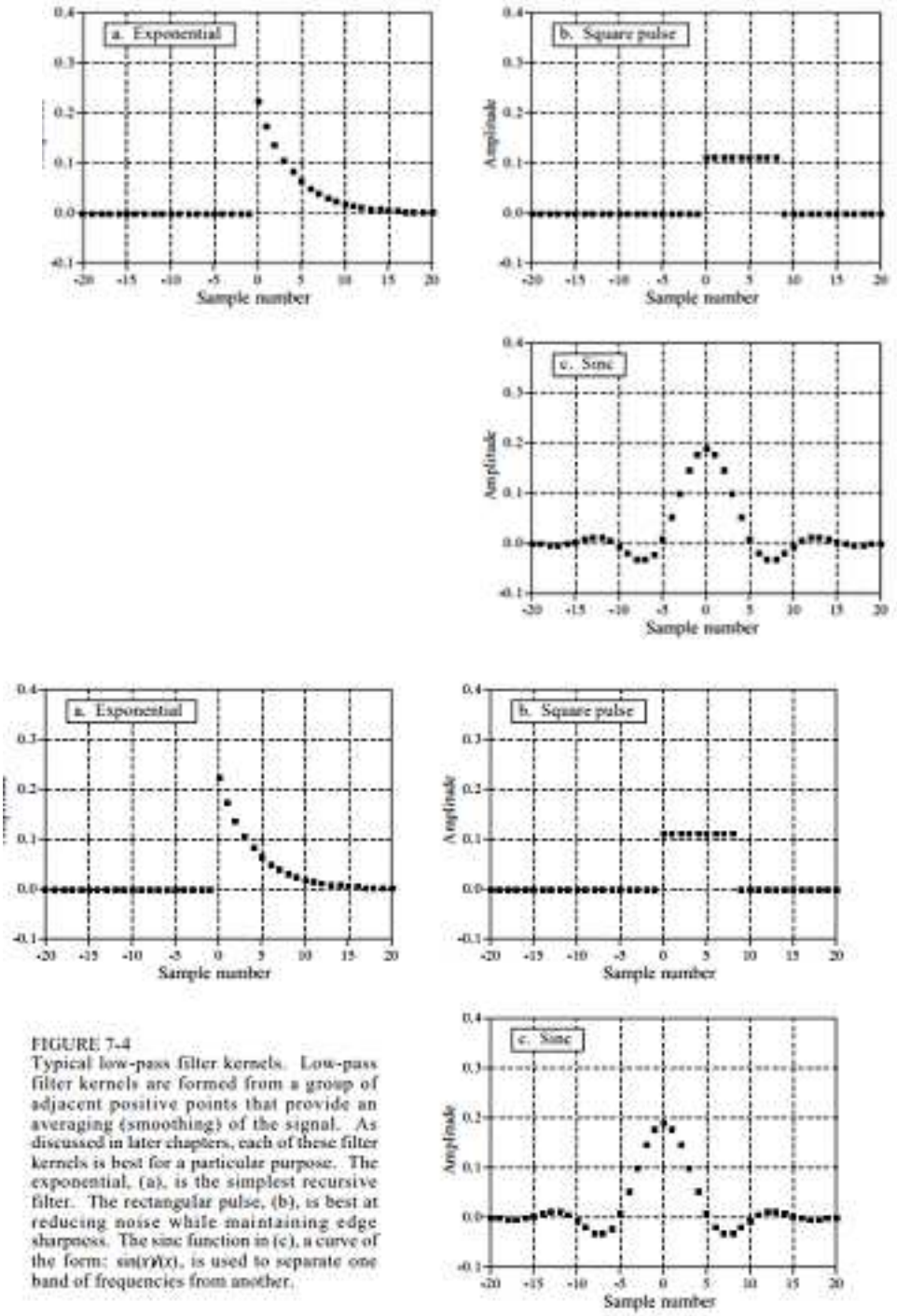


Figura 2.23. Típicos filtros pasa baja

El kernel de la Figura a es el más simple. El de la Figura b es el mejor para reducir ruido blanco y mantener el filo. El kernel de la Figura c es usado para separar una banda de frecuencia de otra.

Se propone utilizar tres diferentes tipos de filtros, el filtro *"Moving average filters", BUILD.IO, "Window-Sync filters"* utilizando el filtro *"BLACK MAN"* por su respuesta paso, en el dominio del tiempo:Los filtros pasa baja estudiados son:

MOVING AVERAGE FILTERS
BUILT.IO
BLACKMAN

Moving Average Filter El Moving aAverage Filter, es el filtro pasa baja más básico de los filtros digitales en dominio del tiempo, utilizado por su simpleza y baja carga operacional; su alta popularidad se debe a su efectividad a la hora de reducir el ruido gaussiano, manteniendo una buena respuesta de paso, la fórmula matemática para este tipo de filtro se ve en:

$$y\left[i\right] = \frac{1}{M} * \sum_{j=0}^{M-1} x[i+j]$$

y[i]: es la señal de salida
x[i]: es la señal de salida
M: es el número de puntos para promediar

El Moving Average Filter es una convolución de la señal de entrada con una ventana cuadrada del tamaño dado por los números de punto M.

BUILT.IO Es un filtro diseñado para videojuegos, el cual utiliza los sensores de los dispositivos android, está pensado para retirar el ruido indeseado obtenido por estos en frecuencias mayores a la fc; para evitar estos altos errores en aplicaciones android es necesario aplicar procesos de suavizado, la naturaleza de este filtro está enfocada a la estabilidad de sus lecturas, por eso al ser calculado sus valores promedios y desviación estándar, se observa que ambos valores fueron reducidos, en especial la desviación estándar.

$$y\left[i\right] = y\left[i-1\right] + fc * \left(x\left[i\right] - y[i-1]\right)$$

y[i]: Señal de salida

x[i]: Señal de salida

fc: Frecuencia de corte

i: Índice que va desde 0 hasta el número total de muestras

BlackMan Filter Filtro por ventanas tipo FIR, ver figura 2.24, con atenuación de banda de -74 dB, con un bandpass ripple de tan solo el 0.02 %, y transición de banda lenta comparada con otros filtros pasa baja.

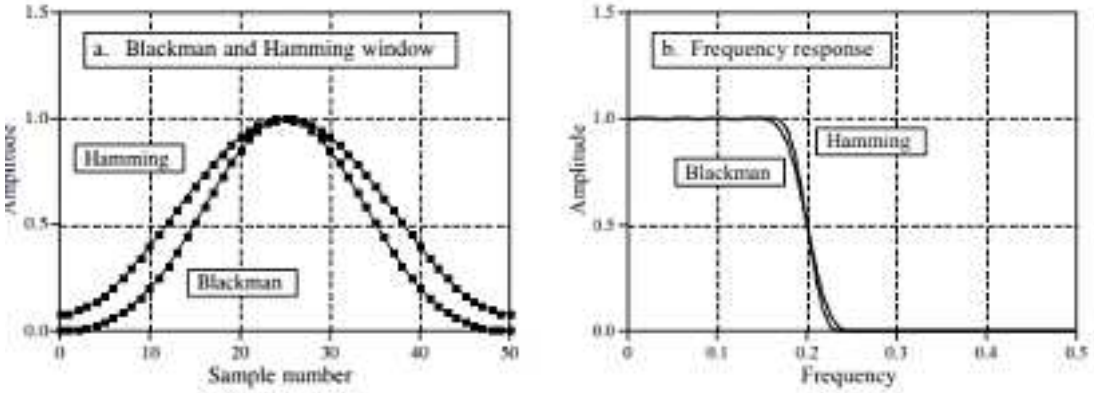


Figura 2.24. Filtro Blackman y su respuesta en frecuencia

Para diseñar los filtros, dos parámetros tienen que ser escogidos, la frecuencia de corte fc, la cual es una porción de la frecuencia de muestreo y el largo del filtro kernel M; debido a la simetría en las bandas de atenuación y no atenuación en este filtro, la fc será escogida entre 0 y 0.5, el valor del ROLL-OFF o el tiempo en que atenúa la señal BW es:

$$M = \frac{4}{BW}$$

M: Tamaño del filtro

BW: Banda de paso entre bandas de no atenuación y atenuación

El BW es afectado por el valor de M, entre más grande sea más rápido será el filtro, mientras que la frecuencia de corte no lo afectara en lo absoluto, esto se observa en la figura 2.25:

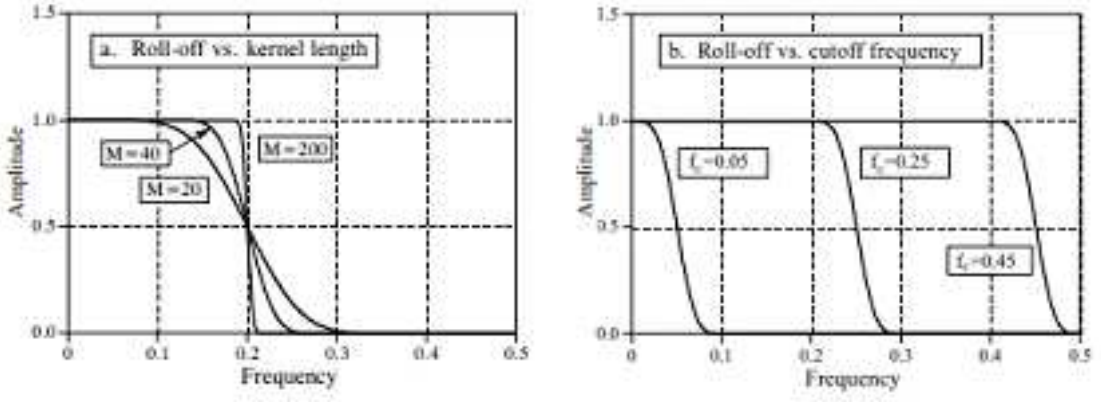


Figura 2.25. Variación de la respuesta en frecuencia con diferente M y Fc

Gracias a la simetría de este filtro se puede utilizar una simple inversión espectral para calcular su forma en pasa alta; la ecuación para calcular este filtro es:

$$h\,[i] = K\frac{\sin\left(2 * pi * fc * \left(i - \frac{M}{2}\right)\right)}{i - M/2} * \left[0{,}42 - 0{,}5 * \cos(\frac{2 * pi * i}{M}) + 0{,}08 * \cos(\frac{4 * pi * i}{M}))\right]$$

h[i]: filtro por ventanas BlackMan
K: Normalización
M: Tamaño del filtro
fc: Frecuencia de corte
i: Índex que recorre de 0 a M