



Documento de diseño

PROYECTO DE APLICACIONES DE BIOMETRÍA Y MEDIO AMBIENTE

Sprint #0

Autor: Santiago Fuenmayor Ruiz

26 de septiembre de 2025

Tabla de contenido

ARDUINO.....3

 INGENIERÍA INVERSA.....4

TELÉFONO7

 INGENIERIA INVERSA.....8

ARDUINO

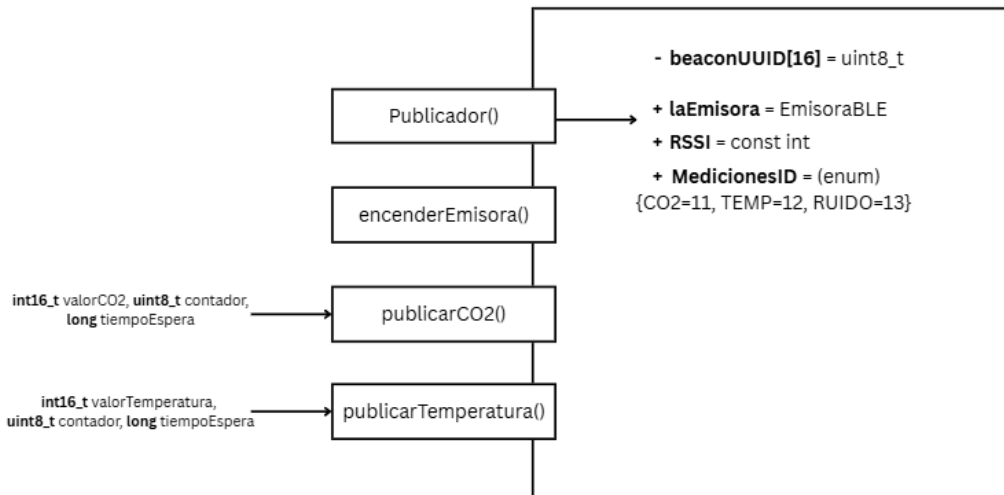
En la parte de Arduino, el código implementa el nodo sensor encargado de simular la lectura de parámetros ambientales y publicarlos mediante Bluetooth Low Energy (BLE) en formato iBeacon. Para ello se estructura en distintas clases con funciones bien definidas:

Medidor se encarga de obtener valores de CO₂ y temperatura (actualmente simulados), **Publicador** gestiona la creación de los paquetes iBeacon y su emisión, y **EmisoraBLE** abstrae la configuración del módulo BLE para encenderlo, detenerlo o iniciar la transmisión. Además, se incluyen clases de apoyo como **PuertoSerie**, utilizada para depuración a través del monitor serie, **LED**, que actúa como indicador visual del envío de datos, y **ServicioEnEmisora**, pensada para manejar servicios y características BLE en futuras ampliaciones.

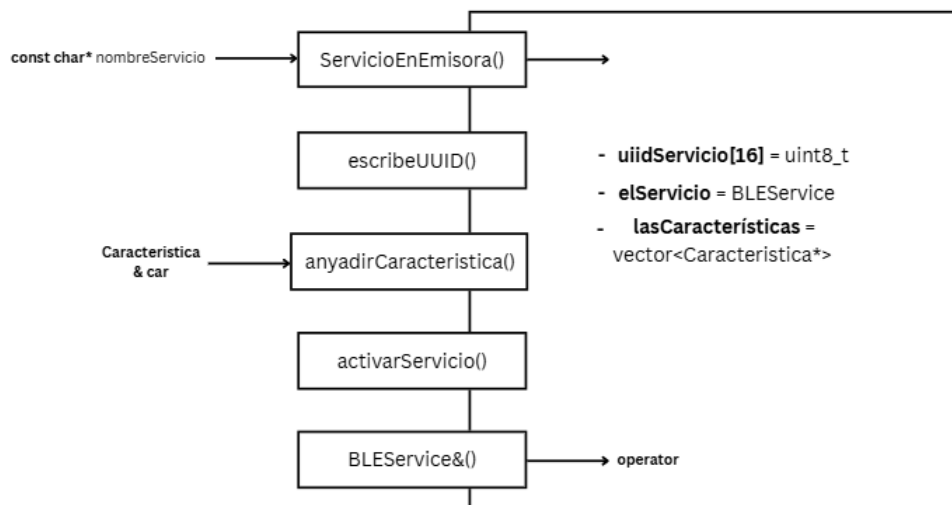
El flujo básico del sistema consiste en inicializar la emisora y el puerto serie, simular las medidas, empaquetarlas en un anuncio iBeacon y transmitirlos durante un tiempo definido, con la posibilidad de indicar el proceso mediante un parpadeo del LED. De este modo, el Arduino actúa como un beacon emisor de datos ambientales, que posteriormente son detectados por la aplicación móvil y enviados al servidor para su almacenamiento y consulta.

INGENIERÍA INVERSA

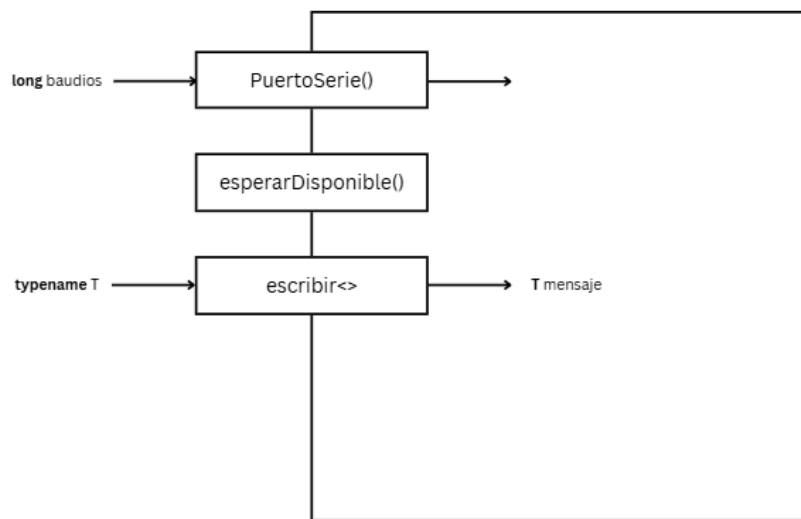
Publicador



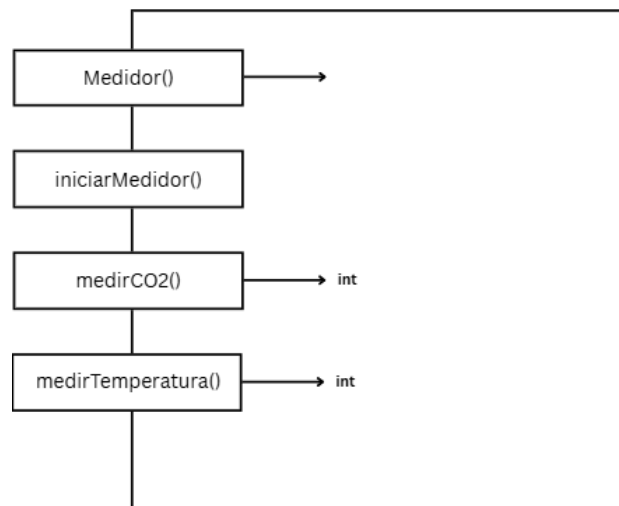
ServicioEnEmisora



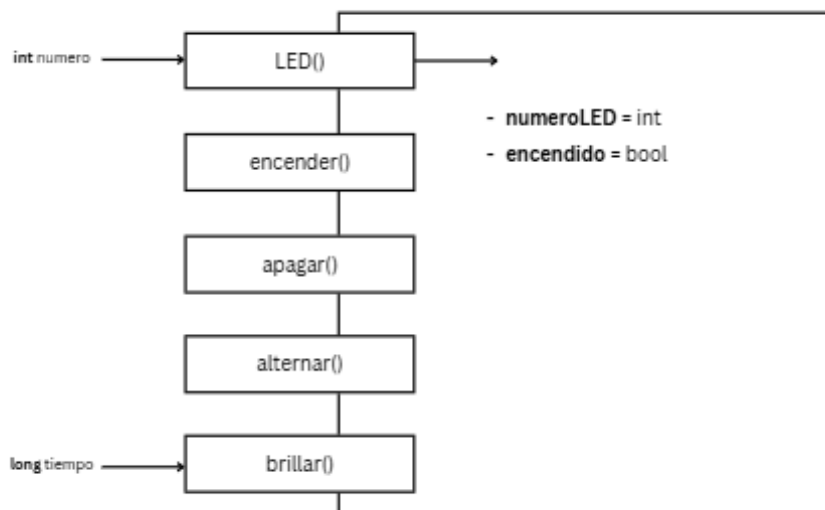
PuertoSerie



Medidor



LED



```

graph TD
    subgraph "Funciones de la biblioteca"
        A[EmisoraBLE()] --> B[encenderEmisora()]
        B --> C[encenderEmisora()]
        C --> D[detenerAnuncio()]
        D --> E[estaAnunciando()]
        E --> F[emitirAnuncionBeacon()]
        F --> G[emitirAnuncioBeaconLibre()]
    end

    subgraph "Parámetros de entrada y salida"
        A_in["const char*, uint16_t, int8_t"] --> A
        B_in["exionEstablecida, exionTerminada"] --> C
        F_in["id, int16_t major, minor, uint8_t rssi"] --> F
        G_in["const char* carga, uint8_t tamanyoCarga"] --> G
    end

    subgraph "Funciones de servicio"
        H[anyadirServicio()] --> I[anyadirServicioConSusCaracteristicas()]
        I --> J[anyadirServicioConSusCaracteristicasYActivar()]
        J --> K[instalarCallbackConexionEstablecida()]
        K --> L[instalarCallbackConexionTerminada()]
        L --> M[getConexion()]
    end

    subgraph "Parámetros de servicio"
        H_in["ServicioEnEmisora & servicio"] --> H
        I_in["ServicioEnEmisora & servicio"] --> I
        J_in["ServicioEnEmisora & servicio, T & restoCaracteristicas"] --> J
        K_in["CallbackConexionEstablecida cb"] --> K
        L_in["CallbackConexionEstablecida cb"] --> L
        M_in["uint16_t coonHandle"] --> M
    end

    subgraph "Parámetros de retorno"
        H_out["bool"] --> H
        I_out["bool"] --> I
        J_out["bool"] --> J
        E_out["bool"] --> E
    end

    subgraph "Listado de parámetros"
        N["- nombreEmisora = const char*"]
        O["- fabricanteID= const uint16_t"]
        P["- txPower= const int8_t"]
    end

```

```
graph TD
    A["const char* nombreCaracteristica"] --> B["Caracteristica()"]
    B --> C["Caracteristica()"]
    C --> D["asignarPropiedadesPermisosYTamanyoDatos()"]
    E["const char*, uint8_t props, SecureMode_t read, SecureMode_t write, uint8_t tam"] --> D
    F["uint8_t props, SecureMode_t read, SecureMode_t write, uint8_t tam"] --> D
    D --> G["escribirDatos()"]
    G --> H["notificarDatos()"]
    I["const char* str"] --> G
    J["const char* str"] --> H
    G --> K["instalarCallbackCaracteristicaEscrita()"]
    H --> K
    K --> L["activar()"]
```

Diagrama de flujo de la función `activar()` en el archivo de código fuente de la librería de ejemplo:

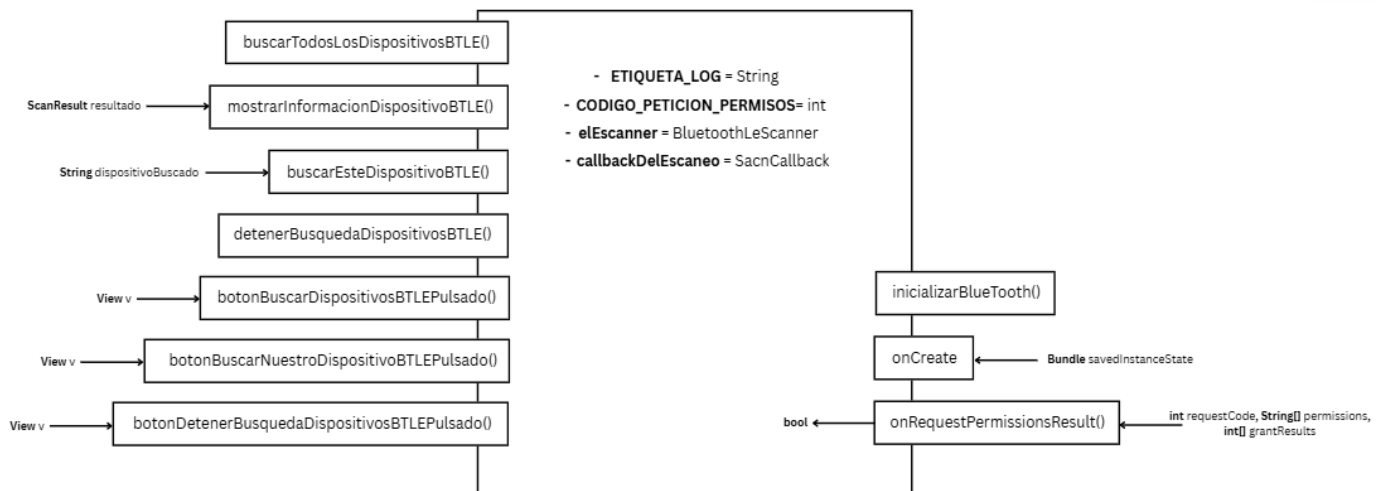
- Entrada: `const char* nombreCaracteristica` → `Caracteristica()`
- Procesamiento: `Caracteristica()` → `asignarPropiedadesPermisosYTamanyoDatos()`
- Entradas adicionales para `asignarPropiedadesPermisosYTamanyoDatos()`:
 - `const char*, uint8_t props, SecureMode_t read, SecureMode_t write, uint8_t tam`
 - `uint8_t props, SecureMode_t read, SecureMode_t write, uint8_t tam`
- Salidas de `asignarPropiedadesPermisosYTamanyoDatos()`:
 - `uint8_t[16]` (asignado a `uuidCaracteristica[16]`)
 - `uint8_t` (asignado a `laCaracteristica`)
- Procesamiento: `asignarPropiedadesPermisosYTamanyoDatos()` → `escribirDatos()` → `notificarDatos()` → `instalarCallbackCaracteristicaEscrita()`
- Entradas para `escribirDatos()` y `notificarDatos()`:
 - `const char* str`
- Salida de `instalarCallbackCaracteristicaEscrita()`: `void`
- Salida final: `activar()`

TELÉFONO

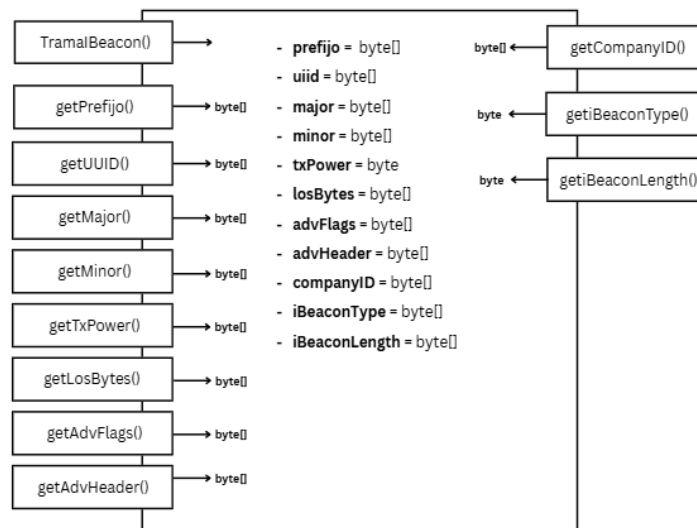
En la parte de **Android**, la aplicación funciona como receptor de los anuncios iBeacon enviados por el Arduino, interpretando las tramas y mostrando los valores medidos. Se apoya en las clases MainActivity para la gestión de la interfaz y detección, TramaBeacon para decodificar los paquetes recibidos y Utilidades para funciones auxiliares, mientras que el AndroidManifest.xml define los permisos de Bluetooth y localización. En este Sprint 0 su funcionamiento es básico o simulado, mostrando cómo se prepararían los datos para su posterior envío al servidor.

INGENIERIA INVERSA

MainActivity



TramalBeacon



Utilidades

