



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Estudio de la tecnología de iBeacons utilizando una Waspote

TITULACIÓN: Grado en Ingeniería de Sistemas de Telecomunicación

AUTOR: Eric Murillo López

DIRECTOR: David Pérez Díaz de Cerio

FECHA: 5 de julio del 2016

Título: Estudio de la tecnología de iBeacons utilizando una Wasmote

Autor: Eric Murillo López

Director: David Pérez Díaz de Cerio

Fecha: 5 de julio de 2016

Resumen

Este documento describe el estudio y uso de dispositivos “beacon”, descritos en Bluetooth Low Energy, utilizando la plataforma Wasmote. El objetivo del trabajo es diseñar y desarrollar un conjunto de pruebas y mediciones que permitan evaluar la calidad de un sistema basado en Wasmote y el módulo BLE, fabricados por Libelium.

El sistema se ha configurado y programado para gestionar mensajes del tipo “advertisement” con el menor consumo posible. Este estudio intenta determinar la viabilidad de uso del sistema planteado, en un entorno de circulación viaria. En este entorno se emiten eventos “advertisement”, con información de la señalización, para los vehículos receptores, que actúan en función de los avisos obtenidos. Este escenario debe ser lo más eficiente energéticamente posible.

El proyecto se ha realizado en varias fases:

- Estudio de la arquitectura Wasmote y del módulo BLE. Estudio del entorno de desarrollo Wasmote y del conjunto de librerías que forma la API.
- Pruebas de concepto para determinar el funcionamiento de bajo consumo de Wasmote y del módulo BLE.
- Preparación del entorno de mediciones y desarrollo de los programas para Wasmote actuando como emisor y como receptor de “advertisements”.
- Ejecución sistemática del entorno para obtener las mediciones (datos cuantitativos).
- Análisis de resultados y conclusiones que nos indicarán la viabilidad del sistema.

El resultado del estudio permite concluir que el sistema Wasmote con el módulo BLE puede enviar “advertisements” con un consumo eficiente. En las conclusiones, se plantean posibles mejoras y vías de investigación, así como el impacto medioambiental y social que esta solución puede suponer.

Title: Estudio de la tecnología de iBeacons utilizando una Wasmote

Author: Eric Murillo López

Director: David Pérez Díaz de Cerio

Date: July, 5th 2016

Overview

This document describes the study and use of the beacon devices, described in Bluetooth Low Energy, using the Wasmote platform. The aim of this work is to design and develop a group of tests and measurements that allow to evaluate the quality of a system based in Wasmote with a BLE module, made by Libelium.

The system has been configured and programmed in order to manage advertisement-type messages with the lowest possible consumption. This study tries to determine the viability in the use of the system presented, in a road traffic environment. In this environment advertisement events are emitted with the signalling information for the receiver vehicles, which act depending on the obtained announcements. This scenery has to be the most energetically efficient possible.

The project has been developed in several phases:

- Study of the Wasmote architecture and of the BLE module. Study of the Wasmote development environment and of the libraries of the API.
- Concept tests to determine the low consumption performance of Wasmote and of the BLE module.
- Set up of the measurements environments and development of programs for Wasmote acting as transmitter and receiver of advertisements.
- Systematic execution of the environment in order to obtain the measurements (quantitative data).
- Analysis of the results and conclusions which will indicate the viability of the system.

The result of this study allows to conclude that the Wasmote system with the BLE module can send advertisements with an efficient consumption. In the conclusions, several improvement and several guidelines towards a further research on the topic are presented, as well as the environmental and social impact that this solution can suppose.

ÍNDICE

INTRODUCCIÓN	1
CAPÍTULO 1. DESCRIPCIÓN DE BLUETOOTH LOW ENERGY – ADVERTISING. APLICACIONES Y SOLUCIONES COMERCIALES	4
CAPÍTULO 2. DESCRIPCIÓN DE WASPMOTE Y DEL MÓDULO BLE	7
2.1 Plataforma Waspote	7
2.1.1 Gestión de energía	8
2.1.2 Gestión de interrupciones.....	9
2.2 Módulo BLE.....	10
2.3 Herramientas de desarrollo	13
2.4 Implementación del bajo consumo e interrupciones en Waspote.....	13
2.5 Implementación de BLE Advertising en Waspote	15
2.5.1 Configuración del modo emisión	15
2.5.2 Tipos de “advertising” y estructura de los paquetes de datos.....	19
2.6 Implementación del modo “scanning” en Waspote	22
2.6.1 Configuración del modo “scanning”.....	22
2.6.2 Funciones de escaneo	23
CAPÍTULO 3. CONFIGURACIÓN DEL ENTORNO	25
3.1 Configuración del adaptador Bluetooth.....	25
3.2 Configuración del analizador de paquetes Wireshark.....	26
3.3 Configuración de Waspote para la generación de logs	26
CAPÍTULO 4. IMPLEMENTACIÓN DEL MODO EMISIÓN	28
CAPÍTULO 5. IMPLEMENTACIÓN DEL MODO RECEPCIÓN.....	34
CAPÍTULO 6. MEDIDAS	38
6.1 Consumo en emisión	39
6.1.1 Configuración de los canales	40
6.1.2 Configuración de los datos	41
6.1.3 Configuración del intervalo de emisión	42
6.1.4 Configuración del modo sleep de Waspote.....	43
6.2 Consumo en recepción	45
6.2.1 Scan continuo.....	45
6.2.2 Scan interval 500 ms, Scan window 250 ms.....	46

6.3 Comparación “log” Waspote vs “log” Wireshark.....	48
CAPÍTULO 7. IMPLEMENTACIÓN RETRANSMISIÓN NODO A NODO	49
CAPÍTULO 8. CONCLUSIONES E IMPACTO SOCIAL.....	51
BIBLIOGRAFÍA	53
ANEXOS	55
Anexo 1. Descripción detallada de Waspote.....	56
A1.1 Datos generales	56
A1.2 Diagrama de bloques de Waspote.....	57
A1.3 Input/Output.....	58
A1.4 Entradas analógicas.....	59
A1.5 Entradas digitales.....	59
A1.6 PWM.....	60
A1.7 UART.....	60
A1.8 I ² C.....	61
A1.9 SPI.....	62
A1.10 USB	62
A1.11 Microcontrolador ATmega1281	62
A1.12 Tarjeta SD	63
A1.13 Real Time Clock – RTC	63
A1.14 LEDs.....	64
Anexo 2. Descripción detallada de Waspote IDE.....	65
A2.1 Descripción del entorno de desarrollo	65
A2.2 Arquitectura del sistema.....	67
A2.3 Estructura de un programa Waspote	68
A2.4 Estilo de programación Waspote y buenas prácticas	68
A2.5 Descripción de las librerías API	70
A2.6 Tipos de memoria de Waspote	70
A2.7 Gestión de la energía en Waspote	72
A2.8 Gestión de interrupciones en Waspote.....	74
Anexo 3. Código fuente de los programas	79
A3.1 Programa sendAdv.pde	79
A3.2 Programa scanAdv.pde.....	83
A3.3 Implementación de flooding	87
A3.4 Modificación de la API para la generación del log	89
Anexo 4. Diagramas de flujo	91
A4.1 Diagrama sendAdv.pde.....	91
A4.2 Diagrama scanAdv.pde.....	92

ÍNDICE DE FIGURAS

Fig. 2.1 Wasmote	7
Fig. 2.2 Relación entre interrupciones y estados de bajo de consumo	9
Fig. 2.3 Módulo BLE	10
Fig. 2.4 Distribución de canales	11
Fig. 2.5 Consumo de corriente medio durante un advertisement	12
Fig. 2.6 Relación potencia de emisión real y configuración del hardware	18
Fig. 2.7 Formato de los paquetes de “advertising”	20
Fig. 3.1 Dispositivos Bluetooth	25
Fig. 3.2 Formato del “log”	27
Fig. 4.1 Esquema del escenario utilizado	28
Fig. 4.2 Ejemplo del programa de emisión	31
Fig. 5.1 Ejemplo del programa de recepción	35
Fig. 5.2 Ventana del serial monitor con información de los datos guardados en la EEPROM	37
Fig. 6.1 Función transformada V_{OUT} vs I_{IN} del sensor de corriente	38
Fig. 6.2 Montaje del sensor de corriente	38
Fig. 6.3 Consumo durante la emisión de “advertisements”: Arriba (Wasmote + módulo BLE), Abajo módulo BLE	39
Fig. 6.4 Consumo de un “advertisement” emitido por dos canales	41
Fig. 6.5 Consumo de un advertisement con 3 bytes de <i>AdvData</i>	42
Fig. 6.6 Consumo de un advertisement con 31 bytes de <i>AdvData</i>	42
Fig. 6.7 “advertisements” con intervalo de 100 ms, consumo general	43
Fig. 6.8 “advertisements” con intervalo de 100 ms, consumo módulo BLE	43
Fig. 6.9 Consumo de la Wasmote durante la emisión de “advertisements” cada 100 ms en ciclos de cambio de modo de 2 s	44
Fig. 6.10 Consumo de la Wasmote durante la emisión de “advertisements” cada 100 ms en ciclos de cambio de modo de 500 ms	45
Fig. 6.11 Consumo de la Wasmote en modo “scan” continuo con tiempo de escucha de 500 ms	46
Fig. 6.12 Consumo de la Wasmote con scan interval 500 ms y scan window 250 ms, cambio de ciclo cada 2 s sin <i>BLE.sleep()</i>	47
Fig. 6.13 Consumo de la Wasmote con <i>scan interval</i> 500 ms y <i>scan window</i> 250 ms, cambio de ciclo cada 2 s con <i>BLE.sleep()</i>	47
Fig. 7.1 Recepción y emisión simultánea	49
Fig. 7.2 Emisión de un advertisement por dos canales, con recepción simultánea	50

ÍNDICE DE TABLAS

Tabla 2.1 Consumo del módulo BLE	11
Tabla 2.2 Opciones de visibilidad.....	16
Tabla 2.3 Opciones de conectividad	16
Tabla 2.4 Opciones de política de recepción y emisión	17
Tabla 2.5 Tipos de “advertising”	19
Tabla 2.6 Parámetros asociados al tipo de “advertising”	20
Tabla 2.7 Estructura de “payload”	21
Tabla 2.8 Modos de escaneo	22
Tabla 4.1 Estructura de datos del “advertisement”	29

ÍNDICE DE ACRÓNIMOS

AD	Advertising Data
ADV_IND	Advertising Indicator Packet
ADV_NONCONN	Advertising Non Connectable Packet
API	Application Programming Interface
BD ADDR	Bluetooth Device Address
BLE	Bluetooth Low Energy
CRC	Cyclic Redundancy Check
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GFSK	Gaussian Frequency Shift Keying
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile communications
HCI	Host Controller Interface
I ² C	Inter-Integrated Circuit
IDE	Integrated Development Environment
LE	Low Energy
LED	Light-Emitting Diode
LTV	Length Tag Value
MAC	Media Access Control
NFC	Near-field Communication
PDU	Protocol Data Unit
PWM	Pulse Width Modulation
RFID	Radio-frequency Identification
RSSI	Received Signal Strength Indicator
SD	Secure Digital
RTC	Real Time Clock
SIG	Special Interest Group
SPI	Serial Peripheral Interface
TFG	Trabajo Final de Grado
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
WTD	Watchdog

Quiero agradecer la ayuda, soporte y comprensión a los profesores David Pérez Díaz de Cerio y José Luis Valenzuela.

Quiero agradecer el apoyo continuo a lo largo de este trabajo a mis padres, novia, hermana y amigos.

INTRODUCCIÓN

El estudio realizado tiene como objetivo principal determinar la viabilidad de un sistema de emisión de “advertisements” que puedan ser recibidos por un dispositivo en movimiento como, por ejemplo, un vehículo en una vía de circulación. En la búsqueda de una circulación viaria más inteligente se han desarrollado diferentes soluciones para que un vehículo conozca información del entorno por donde circula, de manera que pueda utilizar dicha información para circular de manera más segura o para responder a acciones automáticamente.

Un ejemplo, sería el reconocimiento de las señales de limitación de velocidad con una cámara, para acomodar la velocidad a las necesidades de la vía de circulación. Esta tecnología tiene una ventaja frente a la solución que se plantea estudiar, ya que no requiere de un gran despliegue económico, pero no es una solución eficaz. La lluvia, suciedad y otros elementos, como otros vehículos que generen poca visibilidad, dificultan el reconocimiento automático de imágenes. Además, el reconocimiento de imágenes se realiza si el vehículo se encuentra a poca distancia de la señal, de manera que ofrece muy poco tiempo de reacción al conductor en caso de velocidades elevadas.

En escenarios de este tipo se requieren dos condiciones para que la solución sea viable. La primera, que el emisor tenga muy bajo consumo, ya que en una vía de comunicación son muchos los dispositivos emisores que deben tener el mínimo mantenimiento posible (reducir el número de cambios de batería o reducir el coste energético en general). La segunda condición, que el dispositivo receptor pueda recibir un mensaje, a una distancia del emisor que le permita tener tiempo de reacción, en una vía de circulación.

Emitir avisos destinados a cualquier receptor (lo que podríamos llamar “balizas”) es factible mediante Bluetooth a partir de la versión 4.0, que a su vez establece un nuevo protocolo que le permite consumir muy poca energía. Es lo que llamamos Bluetooth Low Energy (BLE) y BLE Advertising.

BLE Advertising es una manera de comunicar un periférico compatible con BLE, con múltiples dispositivos compatibles en un área cercana y en un sentido unidireccional. El dispositivo principal emite un evento para cualquier dispositivo cercano que lo pueda recibir y éste actúa a partir de la información recibida o se conecta en el modo clásico (emparejamiento) para recibir más información. Lo primero que debe definirse al utilizar la tecnología Bluetooth Advertising son los datos y el formato de los paquetes que se emiten. Por ejemplo, iBeacon [1] es una implementación de Bluetooth Advertising realizada por Apple que define el formato de avisos esperado por sus aplicaciones.

La idea principal en la que se basa esta tecnología es mantener los dispositivos en emisión, en modo de bajo consumo, “sleep”, y sólo se despertarán para lanzar un mensaje de tipo baliza con una estructura esperada por el receptor.

Este documento estudia la tecnología de emisión de avisos mediante el protocolo Bluetooth 4.0 con una implementación realizada a partir del producto Wasmote, comercializado por Libelium, al que se le ha añadido un módulo de comunicación compatible con BLE. Hubieran sido posibles otras implementaciones como, por ejemplo, una placa Arduino con una extensión de un módulo que incorpore un procesador de comunicaciones Bluetooth 4.0.

Se ha optado por Wasmote por las características de bajo consumo que asegura el fabricante. Éstas, combinadas con el bajo consumo establecido en las especificaciones de BLE, nos permiten partir de un sistema que, “a priori”, satisface los requerimientos de consumo deseados.

La realización de este proyecto ha requerido estudiar en detalle la tecnología BLE Advertising, la arquitectura de la Wasmote y su integración con el módulo BLE. También ha requerido el aprendizaje del entorno de desarrollo entregado por Libelium, en especial, el conocimiento de las librerías incluidas en su entorno de desarrollo Wasmote IDE.

A lo largo del proyecto se ha seguido una metodología basada en las siguientes fases, en orden cronológico:

- Estudio de la arquitectura Wasmote y del módulo BLE. Estudio del entorno de desarrollo Wasmote y del conjunto de librerías que forma la API, interfaz de programación de aplicaciones.
- Pruebas de concepto para determinar el funcionamiento de bajo consumo de Wasmote y del módulo BLE de “advertisements”.
- Documentación de la Wasmote a nivel teórico (funcionamiento, gestión de interrupciones, gestión de la energía, etc.) y de elaboración de programas (modificaciones de la API).
- Preparación del entorno de mediciones y desarrollo de los programas para que el sistema actúe como emisor y como receptor.
- Ejecución sistemática del entorno para obtener las mediciones (datos cuantitativos)
- Análisis de resultados y conclusiones que nos indicará la viabilidad del sistema.

Como resultado de las fases del proyecto se presentarán los resultados en varios apartados. La estructura de este proyecto está formada por una relación de figuras y tablas, una introducción, siete capítulos y unas conclusiones. Se han añadido 4 anexos que detallan aspectos señalados en los distintos capítulos.

En el primer capítulo se describe la tecnología BLE y los procedimientos de “Advertising”.

En el segundo capítulo se describen los aspectos más importantes de la plataforma Wasmote y de su entorno de desarrollo. Para tener un conocimiento más detallado se han elaborado dos anexos que contienen:

- Anexo 1: Descripción detallada de Wasmote.
- Anexo 2: Descripción detallada de Wasmote IDE.

El tercer capítulo muestra cómo se ha configurado el entorno del proyecto. Se define cómo instalar y configurar los elementos necesarios para que Wasmote y el módulo BLE puedan funcionar en modo emisión o recepción.

Una vez se han descrito los antecedentes necesarios, en el cuarto capítulo se muestra detalladamente la configuración y programación de Wasmote para que funcione en modo de emisión.

En el quinto capítulo se muestra la configuración y programación para Wasmote pero funcionando en modo recepción. El código y diagrama de flujo de los programas se encuentran en el anexo 3 y 4.

- Anexo 3: Código desarrollado.
- Anexo 4: Diagramas de flujo de los programas desarrollados.

La evaluación del rendimiento y del funcionamiento de la plataforma se describe en el sexto capítulo.

En el séptimo capítulo se explica la implementación y análisis de la retransmisión nodo a nodo.

Finalmente, se describen las conclusiones obtenidas durante la realización del proyecto. Además, se incluyen una serie de propuestas, que podrían ser objeto de estudio posterior sobre Wasmote, para mejorar su interacción con el módulo BLE, así como el impacto social y medioambiental que esta solución puede ocasionar.

CAPÍTULO 1. DESCRIPCIÓN DE BLUETOOTH LOW ENERGY – ADVERTISING. APLICACIONES Y SOLUCIONES COMERCIALES

Bluetooth 4.0 es un protocolo de comunicación definido por el grupo SIG (Bluetooth Special Interest Group). En esta versión se define Bluetooth Low Energy (BLE), un subconjunto con una pila de protocolo completamente nueva para desarrollar rápidamente enlaces sencillos entre dispositivos. Está dirigido a aplicaciones con dispositivos de muy baja potencia alimentados con una pila de botón, como alternativa a los protocolos estándar de Bluetooth que se introdujeron desde Bluetooth v1.0 a v4.0.

Se define también la posibilidad de comunicar un dispositivo emisor y receptor sin establecer un emparejamiento previo. Un dispositivo puede emitir un paquete con una estructura predeterminada que cualquier otro dispositivo puede reconocer. Se le llama Bluetooth Advertising. Además, se puede obtener de manera aproximada la distancia que hay entre un emisor y un receptor en función del valor RSSI (Received Signal Strength Indication) que se haya establecido en la comunicación.

Desde un punto de vista comercial se están utilizando tres tipos de dispositivos Bluetooth: Bluetooth clásico, compatible con versiones anteriores, Bluetooth Smart, para dispositivos que sólo soportan el modo Low Energy (LE) y finalmente, Bluetooth Smart Ready, que soporta ambos modos, el clásico y el modo LE.

El protocolo BLE tiene bajas tasas de transferencia ya que no transmite grandes cantidades de información. Básicamente, descubre un dispositivo compatible y establece una comunicación muy simple. Los dispositivos llamados “beacons”, cuya traducción en español es “baliza”, son dispositivos Bluetooth que sólo soportan el modo LE y que sólo utilizan los tres canales de “advertisement”. Un “beacon” transmite paquetes de datos en intervalos regulares con un volumen de datos transmitidos muy pequeño.

La comunicación de un dispositivo BLE utiliza un mecanismo de descubrimiento en una única dirección. El transmisor puede emitir paquetes de datos en intervalos de 20 ms mínimo y 10,24 s como máximo, en pasos de 0,625 μ s. El paquete “advertisement” tiene una longitud máxima de 47 bytes. Se define 1 byte de preámbulo, 4 bytes para la dirección de acceso, los datos y 3 bytes de CRC. Los datos que se transmiten en el paquete (PDU) tienen 2 bytes de cabecera, 6 bytes de la dirección MAC del emisor y de 0 a 31 bytes con los datos propiamente dichos.

En este estudio, Waspote operará en modo “advertisement” no conectable, de manera que toda la información que se transmite está contenida en los 31 bytes de datos, aunque el receptor puede pedir una conexión al dispositivo descubierto. El intervalo mínimo entre paquetes para este tipo de “advertisement” es de 100 ms.

Un ejemplo comercial de “beacon” es iBeacon [1], el cual, es simplemente un uso específico de la tecnología BLE Advertising especialmente soportada en dispositivos con sistema operativo iOS de Apple.

Un iBeacon es un “advertiser” que utiliza 30 bytes de los 31 bytes de datos que se transmiten en un paquete y que tiene un formato preestablecido por Apple. La estructura de los datos de un iBeacon consiste en:

- Prefijo: 6 bytes con el valor “02 01 1A 1A FF 4C 00 02 15” que identifica el modo de transmisión, el fabricante (4C 00 para Apple) y otros datos.
- UUID proximidad: identificador de 16 bytes del tipo de iBeacon, lo que permite que las aplicaciones puedan distinguirlos. Un receptor puede escanear un iBeacon con un UUID concreto.
- Major number: código de 2 bytes que identifica un grupo de iBeacons lo que permite tener un agrupador útil para las aplicaciones.
- Minor number: código de 2 bytes que identifica un iBeacon concreto del grupo del Major number.
- TX Power: un byte que indica la potencia de transmisión y que podemos convertir en una medida de distancia entre receptor y emisor.

Pero no sólo Apple ha implementado una solución utilizando BLE Advertising. Google ha creado un proyecto de código abierto llamado Eddystone [2] soportado en sistemas operativos iOS y Android. En sus especificaciones también se incluye información sobre el estado de la batería. Radius Networks también ha creado unas especificaciones para mensajes de tipo broadcast abierto y gratuito llamado AltBeacon [3].

Pero sea cual sea el formato del paquete de datos que se transmite en un “advertisement”, lo que hace tan prometedor esta tecnología son sus cualidades de geolocalización en espacios muy cercanos y el bajo consumo que se requiere. El receptor puede ser un teléfono inteligente compatible BLE con una aplicación que capture “advertisements”. Por ello son muchos los ámbitos de actuación en los que se podría utilizar un “beacon” y muchos los dispositivos que podrían recibir los paquetes de datos.

En particular, en una infraestructura viaria puede tener una gran utilidad. Por ejemplo, es posible instalar un dispositivo emisor en una señal de limitación de velocidad, de manera que, un vehículo obtenga información sobre cuál es la velocidad máxima permitida y actúe en consecuencia. La obtención de la información debe hacerse con cierta antelación para permitir la reacción del conductor. La distancia máxima considerada habitualmente para Bluetooth son 100 metros. Esta distancia máxima es pequeña para este escenario, ya que debido a la velocidad de circulación media podríamos perder paquetes u obtener la información con muy poco tiempo de maniobra. Ahora bien, se pueden obtener distancias de hasta un kilómetro como se concluye en este

artículo, [4] por lo que en nuestro caso la obtención de información con suficiente tiempo de antelación es factible. El volumen de datos transmitidos es pequeño y admisible por el formato de un paquete y el emisor tiene bajo consumo lo que facilita la reducción de las necesidades de mantenimiento de las señales, ya que cada señal puede disponer de una pila de botón con una duración de varios meses o años. El sistema, por tanto, parece factible y mejora el actual reconocimiento visual de imágenes, ya que no se ve afectado por la lluvia, suciedad y otros elementos que dificultan el reconocimiento automático de imágenes.

El uso de emisores de “advertisements” puede ser muy útil en entornos comerciales, turísticos o en eventos. En un entorno comercial se puede guiar al usuario que disponga de un receptor (un móvil con Bluetooth 4.0 y una aplicación adecuada) utilizando la cercanía a una tienda. Es lo que llamaríamos “geomarketing”, con un gran potencial para las empresas comerciales.

En un museo podemos emitir información relacionada con un objeto cercano que está expuesto. Se podría disponer de una guía interactiva que mostrase información en el itinerario elegido por el propio usuario y no por el predeterminado por el propio museo. En un evento con un volumen importante de personas sería una manera de enviar información útil para el usuario, en tiempo real.

Otras aplicaciones pueden llegar a ser sorprendentes. En la ciudad de Busan [5], en Corea del Sur, se está desarrollando una experiencia piloto para garantizar que las mujeres embarazadas o personas con necesidades especiales, dispongan de un asiento reservado en el transporte público. Para ello, se han entregado dispositivos “beacon” en forma de llavero que emiten un “advertisement” que será detectado en la zona de asientos reservados para que, ante la recepción del aviso, se encienda una luz que indica que uno de los asientos reservados debe ser desocupado. El indicador luminoso se apaga cuando el llavero se encuentra junto al asiento, es decir, cuando la persona con necesidad especial ya se ha sentado en el asiento destinado a tal efecto.

Otro posible uso de un llavero “beacon” podría ser, que el propietario del llavero “informase” a un autobús público que necesita la apertura de una rampa para entrar en el autobús con una silla de ruedas. Una vez dentro del autobús, la proximidad permitiría también comunicar que la rampa ya puede ser recogida.

En centros hospitalarios, podría servir para guiar a los pacientes a las diferentes dependencias a las que debe dirigirse para hacerse pruebas así como para que el personal hospitalario pudiera localizar el material hospitalario compartido, por ejemplo, en urgencias.

En resumen, estamos delante de una tecnología con grandes posibilidades y en la medida en que se generalice el uso de los dispositivos emisores se desarrollaran aplicaciones útiles, generando una demanda que reducirá los costes de fabricación. Una reducción de los costes de fabricación de los emisores facilitará una extensión de su uso, lo que también redundará en una mejora para los usuarios.

CAPÍTULO 2. DESCRIPCIÓN DE WASPMOTE Y DEL MÓDULO BLE

2.1 Plataforma Wasmote

Wasmote es una plataforma de sensores inalámbricos de código abierto basada en Arduino. Tiene una arquitectura modular, especialmente centrada en la aplicación de modos de bajo consumo, para permitir que los nodos sensores ("motes") puedan ser completamente autónomos y alimentados por batería. Ofrece una vida útil variable entre 1 y 5 años dependiendo del ciclo de trabajo y la radio utilizada.

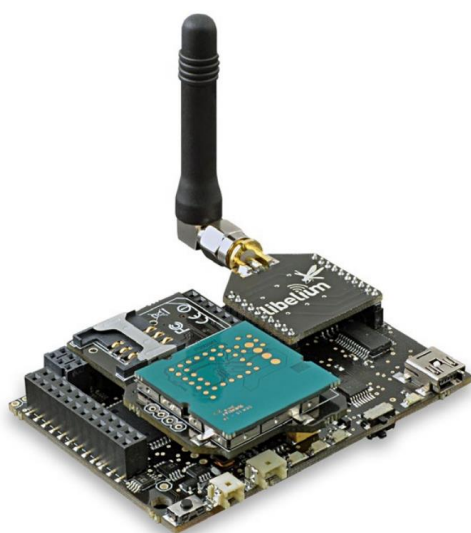


Fig. 2.1 Wasmote

La plataforma se basa en una placa con un microcontrolador ATmega1281 con opciones de funcionamiento de bajo consumo y con varios sensores incorporados en la placa base como un sensor de temperatura, un acelerómetro y un detector de nivel de batería.

La placa base incorpora varios conectores para entradas y salidas. Dispone de 7 entradas analógicas, 8 entradas/salidas digitales, una PWM, dos UART, una I²C, un puerto mini USB, un conector para GPS y una SPI.

El fabricante tiene una gama muy variada de sensores compatibles con la placa base que permiten monitorizar concentraciones de gases, análisis de agua y de ruido, entre otros.

Wasmote dispone de varios módulos de radio conectables a partir de dos sockets de la placa base. Esto permite a la plataforma ser muy versátil y de esa manera elegir el módulo que más se adapte al proyecto. En particular se disponen de módulos de radio ZigBee, Wifi, GSM/GPRS, GPRS + GPS, 3G + GPS, RFID/NFC, Bluetooth. En nuestro caso se ha elegido un módulo Bluetooth Low Energy suministrado por Libelium, por sus cualidades de bajo consumo y por la posibilidad de utilizar el protocolo BLE Advertising mediante una API desarrollada por Libelium.

El componente UART es clave en el diseño de Waspote. La placa dispone de dos UART y además varios puertos pueden ser conectados al mismo UART mediante el uso de multiplexores. Esta disposición puede verse esquematizada en la figura “Diagrama de bloques de las señales de datos” que figura en el Anexo 1. El módulo BLE comparte el UART0 junto con el puerto mini USB utilizando un multiplexor de Texas Instruments.

El puerto mini USB se utiliza para comunicarse con un ordenador o con un dispositivo compatible. A través del puerto se puede alimentar la plataforma y se carga el programa en el microcontrolador. Por defecto UART0 controla la señal de datos del Socket0 donde se conecta el módulo BLE y atiende a la emisión/recepción de datos a través del USB sólo momentáneamente, cuando así es requerido.

En los próximos apartados se destacan dos aspectos de Waspote que son clave en el proyecto: la gestión de la energía y la gestión de las interrupciones. El Anexo 1 contiene una descripción detallada de la plataforma Waspote.

2.1.1 Gestión de energía

Waspote dispone de tres modos de operación de bajo consumo, llamados “Sleep”, “Deep Sleep” e “Hibernate”.

“Deep Sleep” y “Sleep” consumen $55\mu\text{A}$, mientras que “Hibernate” consume $0,06\mu\text{A}$. En estos modos de operación, el consumo es muy inferior a los 15mA del modo de operación normal (modo “ON”). El microprocesador puede permanecer en modo “Sleep” entre 32 milisegundos y 8 segundos. En el modo “Deep Sleep” e “Hibernate” puede permanecer entre 8 segundos y minutos, horas o días. Cuando el procesador entra en un modo de bajo consumo sólo puede ser despertado mediante una interrupción de hardware, proporcionada por un sensor, un módulo de comunicaciones, o el reloj interno RTC.

Por último, el microprocesador puede entrar en modo hibernación, “Hibernate”. En dicho estado el programa principal se para y el microcontrolador y los módulos están totalmente desconectados. Sólo puede despertar con una alarma previamente programada en el reloj interno RTC que es el único dispositivo alimentado en ese modo. Se puede permanecer en este estado desde segundos hasta minutos, días u horas.

El procedimiento que se debe seguir para que Waspote entre en el modo hibernación requiere de una acción manual sobre la placa, que consiste en que la primera vez que se ejecuta el programa, se debe situar el interruptor de hibernación en la posición OFF (se dispone de 3 segundos para hacerlo mientras el LED rojo se ilumina intermitentemente). Una vez cambiada la posición, cada vez que la Waspote se reinicie comprobará el interruptor, e hibernará o no, en función de éste.

Se decidió utilizar Wasmote por sus cualidades en cuanto a consumo de energía. En los capítulos de implementación de emisión y de recepción se configurará Wasmote para conseguir el mínimo consumo posible y será objeto de estudio. Parte fundamental del proyecto recae en el hecho de que debe ser un sistema que consuma muy poca energía.

2.1.2 Gestión de interrupciones

Como ya se ha indicado en el apartado de “Gestión de la energía”, el microcontrolador de Wasmote permanece en un estado de bajo consumo esperando cualquier interrupción. Si un sensor supera un valor determinado (umbral), advierte a Wasmote generando una interrupción y de esa manera Wasmote recupera el control saliendo de su estado de bajo consumo.

La relación entre los estados de bajo consumo y las interrupciones se resume en el siguiente gráfico:

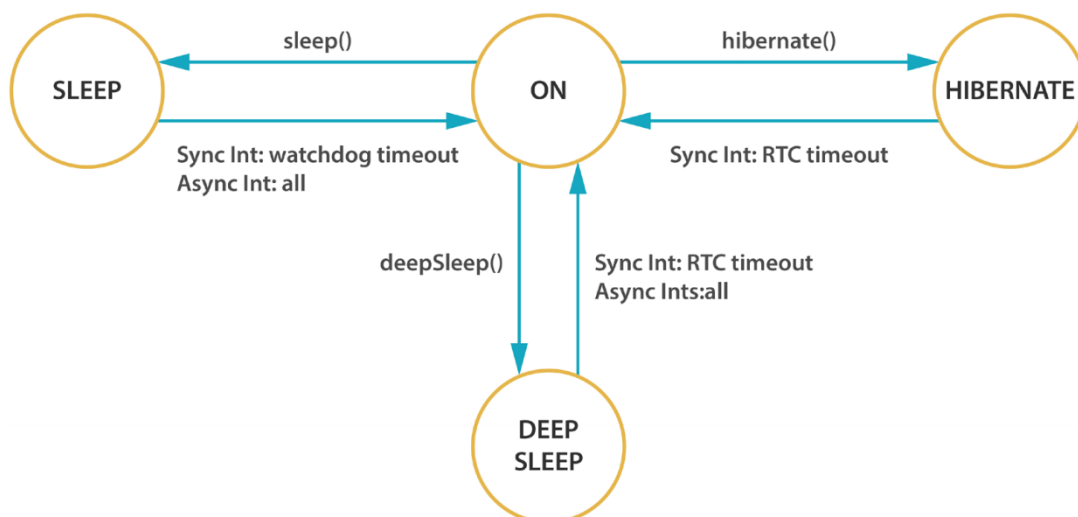


Fig. 2.2 Relación entre interrupciones y estados de bajo de consumo

Las interrupciones síncronas son programadas por los relojes internos y pueden ser de dos tipos:

- Alarmas periódicas que son gestionadas por el reloj interno RTC y que permiten generar una interrupción en un momento concreto del futuro. El RTC que incorpora Wasmote es un oscilador de cristal de cuarzo Maxim DS3231SN que corre a 32 kHz y es muy preciso, teniendo una variación de 1 minuto por año. En el modo de hibernación, Wasmote sólo consume lo necesario para alimentar al RTC, 0,06µA. El

microcontrolador y el RTC utilizan el bus I²C actuando como maestro y esclavo respectivamente.

- Alarmas relativas que son programadas teniendo en cuenta el momento actual y son controladas por RTC y el microcontrolador interno "Watchdog", abreviado WTD, que corre a 128 kHz. El WTD genera una señal de interrupción cuando el reloj llega a un determinado valor que puede despertar al microcontrolador del estado de bajo consumo "Sleep".

Las interrupciones asíncronas, que se ejecutan cuando se cumple una condición, en el caso del acelerómetro, por ejemplo, cuando detecta una caída libre, son las siguientes:

- Sensores
- Acelerómetro
- Módulo XBee (sólo protocolo Digimesh)
- Watchdog
- RTC

Como se puede observar, no hay ninguna interrupción establecida para el módulo BLE. Esto es una mejora o posible futura línea de investigación, ya que sería muy interesante tener la posibilidad de despertar el procesador de la Waspote, en función de las interrupciones emitidas por el módulo BLE.

2.2 Módulo BLE

El módulo utilizado para este trabajo es un módulo Bluetooth compatible con Bluetooth Low Energy.



Fig. 2.3 Módulo BLE

El módulo BLE puede enviar anuncios de tipo "broadcast", permite escanear otros dispositivos, permite conectarse a otros dispositivos BLE en modo maestro o esclavo, puede conectarse con teléfonos inteligentes, permite alternar funcionamiento de bajo consumo con funcionamiento en modo transmisión y es capaz de determinar la distancia con otro dispositivo.

Los módulos BLE utilizan la banda 2,4GHz (2402MHz – 2480MHz) dividida en 37 canales de datos y 3 de advertisement, con una separación entre ellos de 2 MHz y modulación GFSK.

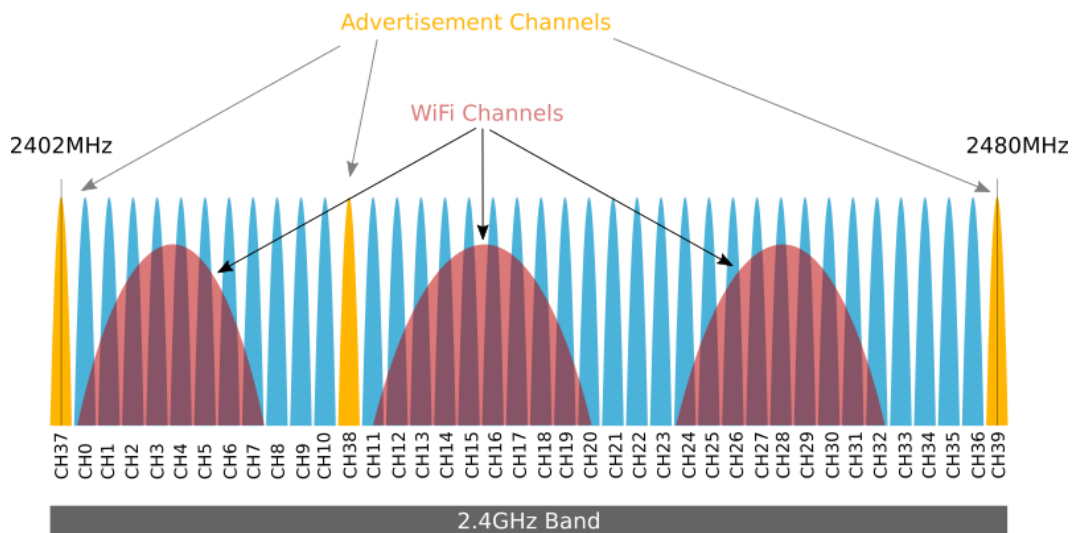


Fig. 2.4 Distribución de canales

A continuación, tenemos sus especificaciones:

Protocolo: Bluetooth v4.0 / Bluetooth Smart

Chipset: BLE112 (Bluegiga)

Sensibilidad RX: -103dBm

TX Power: [-23dBm, +3dBm]

Antena: 5dBi

Seguridad: AES 128

Rango: 100 metros (a máxima velocidad de transmisión). Este rango puede ser aumentado considerablemente [4].

El módulo BLE está gestionado por el UART0 y está conectado a la Waspote por el Socket0 alimentándose por 3,3V. La siguiente tabla muestra el consumo medio en diferentes estados del módulo.

Tabla 2.1 Consumo del módulo BLE

Estado	Consumo
OFF	0mA
Sleep	0,4µA
ON	8mA
Emitiendo (“advertising” / conexión)	36mA

El estándar Bluetooth Low Energy ha sido diseñado para un bajo consumo de energía. A pesar de ello, durante la transmisión, podemos alcanzar valores relativamente altos durante pequeños períodos de tiempo.

El siguiente gráfico muestra una curva media de consumo de corriente (mA) en función del tiempo (ms), mientras que un advertisement se envía por el módulo BLE. Esta gráfica está proporcionada por Libelium. Se analiza y compara con las gráficas obtenidas en laboratorio en el capítulo de mediciones.

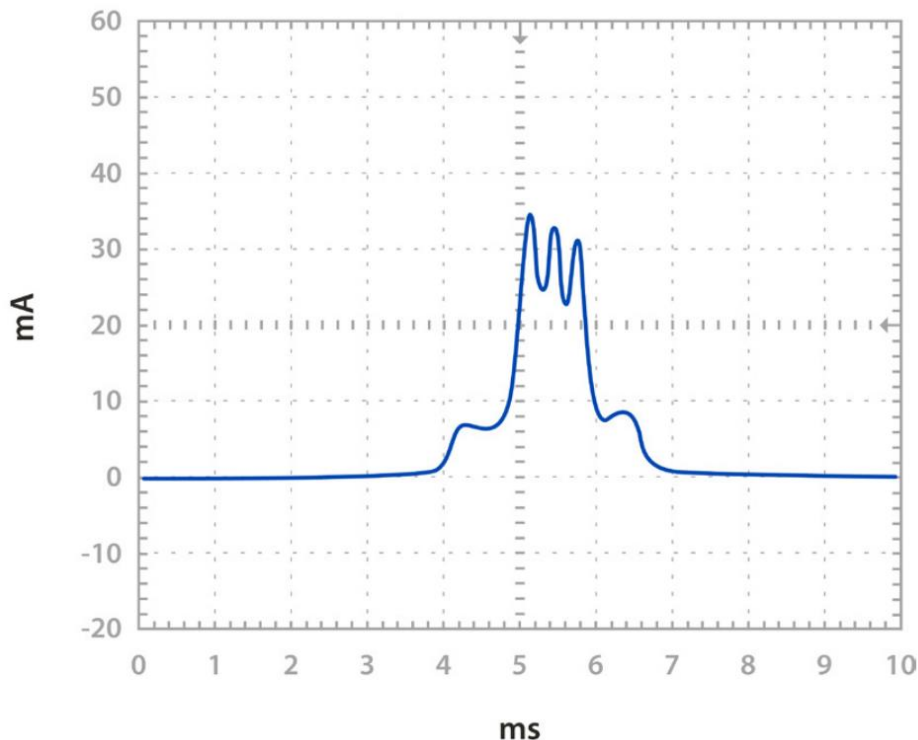


Fig. 2.5 Consumo de corriente medio durante un advertisement

De la misma manera que Waspote tiene varios modos de funcionamiento de bajo consumo, el módulo BLE utiliza los estándares Bluetooth Low Energy que han sido diseñados para optimizar el consumo del dispositivo. Los estados posibles son:

- "Sleep" con el mejor consumo posible de $0,4\mu\text{A}$.
- Encendido sin comunicación, "ON", con un consumo de 8mA.
- Encendido con emisión de "advertisements", con un consumo de 36mA.

Si consideramos, que los momentos de transmisión se producen en periodos cortos de tiempo, el promedio de consumo es muy reducido.

El módulo utiliza tres modos de bajo consumo que son gestionados de manera automática por el chipset y que oscilan entre $235\mu\text{A}$ y $0,4\mu\text{A}$ de consumo [14].

2.3 Herramientas de desarrollo

Libelium, fabricante de Wasmote, proporciona el entorno de desarrollo Wasmote IDE para programar la plataforma, incluyendo, un compilador y un conjunto de librerías de código abierto que constituyen la interfaz de programación de la plataforma (librerías API desarrolladas en C y C++). Al ser de código abierto, las librerías pueden ser revisadas e incluso mejoradas por el usuario. Sin embargo, Libelium no lo recomienda ya que pueden darse situaciones indeseables como colisión de datos, conflictos de memoria o comportamientos inesperados.

Wasmote IDE es el entorno de desarrollo integrado para poder programar Wasmote. El compilador está basado en el compilador de código abierto de la plataforma Arduino, si bien, Libelium no recomienda utilizar el IDE de Arduino ya que únicamente la versión descargable desde la web de Libelium está garantizada para que funcione de manera correcta con la plataforma Wasmote.

En el proceso de instalación del entorno se crean varias carpetas donde se encuentran los ejemplos suministrados por el fabricante, las librerías “core” y las librerías específicas de cada módulo. Además, se instala el compilador. A través de la conexión mini USB de Wasmote, ésta se puede conectar al ordenador, de manera que desde el entorno de desarrollo podemos ejecutar un programa de monitorización de la Wasmote donde recibiremos el “output” de los programas que se ejecuten y también podremos enviar comandos al programa cargado. Previamente, se deben instalar en el ordenador, los drivers que crearán un puerto serie virtual encaminado a través del puerto mini USB de Wasmote. El puerto mini USB, además, sirve para alimentar a la placa y permite descargar los programas desarrollados en Wasmote IDE.

En el Anexo 2 se describe con detalle el entorno de desarrollo: el compilador, la API en general, buenas prácticas, las librerías del módulo BLE, las librerías de gestión de energía, de gestión de interrupciones y de gestión de la memoria, así como la generación de logs en la tarjeta SD. A lo largo del proyecto se ha requerido aprender con detalle las librerías del entorno de desarrollo a partir de los ejemplos incluidos en la instalación y de la documentación en línea mantenida por Libelium.

2.4 Implementación del bajo consumo e interrupciones en Wasmote

Las funciones para gestionar el bajo consumo se incluyen en la clase WaspPWR. En dicha clase, se encuentran las funciones para elegir el modo de bajo consumo así como todo lo relacionado con la gestión de la batería. A continuación, se describen las funciones para implementar los diferentes modos de consumo.

- “Sleep”: En este modo, Wasmote quedará a la espera de una interrupción síncrona generada por el WTD o una interrupción asíncrona. Con la función *sleep(uint8_t timer, uint8_t option)* se define el tiempo que se quiere esperar hasta la generación de la interrupción por parte del WTD y qué componentes de la Wasmote se quieren dormir.
 - *timer*: WTD_16MS, WTD_32MS, WTD_64MS, WTD_128MS, WTD_250MS, WTD_500MS, WTD_1S, WTD_2S, WTD_4S, WTD_8S.
 - *option*: Estas opciones son comunes para el modo “Deep Sleep”. ALL_OFF, duerme toda la Wasmote a excepción del WTD; SENS_OFF, duerme toda la placa a excepción del WTD y el Socket0; SOCKET0_OFF, duerme toda la placa a excepción de WTD y los pines para los sensores.

- “Deep Sleep”: En este modo, Wasmote quedará a la espera de una interrupción síncrona generada por el RTC o por una interrupción asíncrona. Se configura mediante la siguiente función: *deepSleep(const char* time2wake, uint8_t offset, uint8_t mode, uint8_t option)*.
 - *time2wake*: Define el tiempo de espera hasta la generación de la interrupción. El formato es DD:HH:MM:SS (Día, Hora, Mes, Segundos).
 - *Offset*: Define si el tiempo de espera es relativo o absoluto. Si es relativo (RTC_OFFSET) pasará un tiempo *time2wake* desde que se ejecuta la función hasta que se genera la interrupción, mientras que con un tiempo absoluto (RTC_ABSOLUTE), la interrupción se generará a modo de alarma en un tiempo concreto. Se puede configurar el día, mes, año y hora del RTC para indicarle que genere una interrupción un día a una hora en concreto.
 - *mode*: Especifica el modo de la alarma RTC.
 - *option*: Al igual que en el modo Sleep, se define qué partes de la placa se dormirán.

- “Hibernate”: En este modo, Wasmote quedará a la espera de una interrupción síncrona generada por el RTC. Se configura mediante la función *hibernate(const char* time2wake, uint8_t offset, uint8_t mode)* donde los parámetros y funcionamiento coinciden con los de “Deep Sleep”, con la salvedad de que no es posible elegir qué parte de la placa dormir, ya que para conseguir el mínimo consumo posible, sólo el RTC es alimentado.

Los puntos 7 y 8 del Anexo 2 describen con detalle la gestión del consumo y las interrupciones tanto a nivel de software como de hardware. Se explica cómo Wasmote trata estas interrupciones asociadas a unos pines concretos, cómo trata la información el multiplexor, así como cuál debe ser el esquema de programación de interrupciones.

2.5 Implementación de BLE Advertising en Wasmote

Libelium permite interactuar con el módulo BLE a través de la API del módulo incluida en las librerías del entorno de desarrollo. Todas las funciones para gestionar el módulo se incluyen en la clase WaspBLE. El objeto creado se llama BLE y se crea de manera automática. La librería se compone de dos archivos WaspBLE.cpp y WaspBLE.h.

Wasmote se comunica con el módulo BLE enviándole comandos. El módulo BLE responderá al comando o enviará un evento. La API del módulo nos permite configurarlo en uno de los cuatro modos de funcionamiento previsto en el estándar Bluetooth 4.0. Estos modos son:

- “Advertiser”: Emisor de mensajes “advertisement”.
- “Scanner”: Modo escucha de emisores del tipo “advertiser”.
- “Master”: Establece comunicación con otro dispositivo, emitiendo y marcando los tiempos de emisión.
- “Slave”: Dispositivo conectado a un único “master”.

El trabajo se plantea en dos escenarios en los que el módulo BLE actúa como “Advertiser” o como “Scanner”.

Para poder utilizar el módulo BLE es necesario inicializarlo. Durante ese proceso el módulo es alimentado, se permite la comunicación entre Wasmote y el módulo mediante el UART y algunas variables se inicializan. Se utilizan dos métodos: *BLE.ON()* y *BLE.OFF()* para activar o desactivar el módulo BLE. También es posible reiniciar por programa el módulo mediante el método *BLE.reset()*.

El fabricante no permite seleccionar el modo de bajo consumo que será utilizado, sino que los establece de manera automática, utilizando el mejor modo en función de las necesidades del chipset en cada momento. Debemos confiar, por tanto, en que, durante la emisión de “advertisements”, el consumo por parte del módulo BLE será el mínimo posible. Para ello, utiliza el método *BLE.sleep()* que deshabilitará la conexión UART con Wasmote, lo que le impide recibir comandos de Wasmote, y seleccionará el mejor modo posible.

El método para despertar el módulo y permitir la conexión mediante UART con Wasmote es *BLE.wakeup()*.

2.5.1 Configuración del modo emisión

Un “advertisement” es un dispositivo BLE que emite un conjunto de datos, en un formato preestablecido, sin establecer primero una conexión con otro dispositivo BLE.

Existen diferentes posibilidades de emisión de un “advertisement”. Para establecer el modo de emisión se utilizan diferentes métodos con una serie de

parámetros definidos en la API. Las características que se pueden configurar son:

- a) **Visibilidad:** Determina qué módulos pueden ver “advertisements” emitidos por el módulo BLE. La función que se utiliza es *setDiscoverableMode(Parámetro_Visibilidad)*.

Tabla 2.2 Opciones de visibilidad

Parámetro visibilidad	Funcionamiento del módulo BLE
BLE_GAP_NON_DISCOVERABLE	El módulo BLE no está enviando paquetes
BLE_GAP_LIMITED_DISCOVERABLE	El módulo BLE emite “advertisements” que pueden ser visibles por un dispositivo en modo de escaneo limitado
BLE_GAP_GENERAL_DISCOVERABLE	El módulo BLE es visible para los dispositivos en modo de escaneo general
BLE_GAP_BROADCAST	Igual que BLE_GAP_NON_DISCOVERABLE, pero emite “advertisements”
BLE_GAP_USER_DATA	El módulo envía un “advertisement” definido por el usuario

- b) **Conectividad:** El aviso incluye un “flag” que indica si el módulo BLE admite emparejamientos o no. Se utiliza llamando al método *setConnectableMode(Parámetro_conectividad)*. El parámetro determina la característica de conectividad del módulo BLE:

Tabla 2.3 Opciones de conectividad

Parámetro conectividad	Funcionamiento del módulo BLE
BLE_GAP_NON_CONNECTABLE	El módulo BLE no podrá conectarse con ningún otro dispositivo.
BLE_GAP_DIRECTED_CONNECTABLE	El módulo BLE podrá ser conectado por sólo un nodo
BLE_GAP_UNDIRECTED_CONNECTABLE	El módulo BLE podrá ser conectado por cualquier nodo

- c) **Filtros en la comunicación:** Es posible definir una política de envío de avisos para configurar si el módulo BLE responderá a las peticiones de

escaneo y si permitirá conexiones con otros dispositivos. El método utilizado es *setFiltering(scan_pol, adv_pol, scan_duplicate)* y el segundo parámetro, *adv_pol*, determina el comportamiento de las peticiones que pueda recibir el módulo BLE:

Tabla 2.4 Opciones de política de recepción y emisión

adv_pol	Funcionamiento del módulo BLE
BLE_GAP_ADV_POLICY_ALL	El módulo BLE responde a cualquier petición de conexión de un “master” y permite la conexión.
BLE_GAP_ADV_POLICY_WHITELIST_SCAN	Responde sólo a las peticiones de escaneo a partir de una “lista blanca” y permite la conexión de cualquier “master”
BLE_GAP_ADV_POLICY_WHITELIST_CONNECT	Responde sólo a las peticiones de escaneo de cualquier master y sólo permite la conexión con los dispositivos de una “lista blanca”
BLE_GAP_ADV_POLICY_WHITELIST_ALL	Responde sólo a las peticiones de escaneo a partir de una “lista blanca” y sólo permite la conexión con uno de ellos

Una “lista blanca” es una lista de dispositivos con permisos especiales para una determinada acción. *whiteListAppend(mac)* permite añadir el dispositivo con su dirección MAC. *whiteListRemove(mac)* permite eliminar un dispositivo de la lista y *whiteListClear()* elimina la lista de manera completa.

- d) **Potencia de transmisión:** La potencia de transmisión afecta a la distancia máxima a la que se puede recibir un “advertisement”. Utiliza el método *setTXPower(integer)* que admite como parámetro un entero entre 0 y 15, lo que equivale a una potencia real de -23 a +3 dBm. A continuación, la figura **Fig. 2.6** muestra la relación entre la potencia real y la configuración del hardware [6].

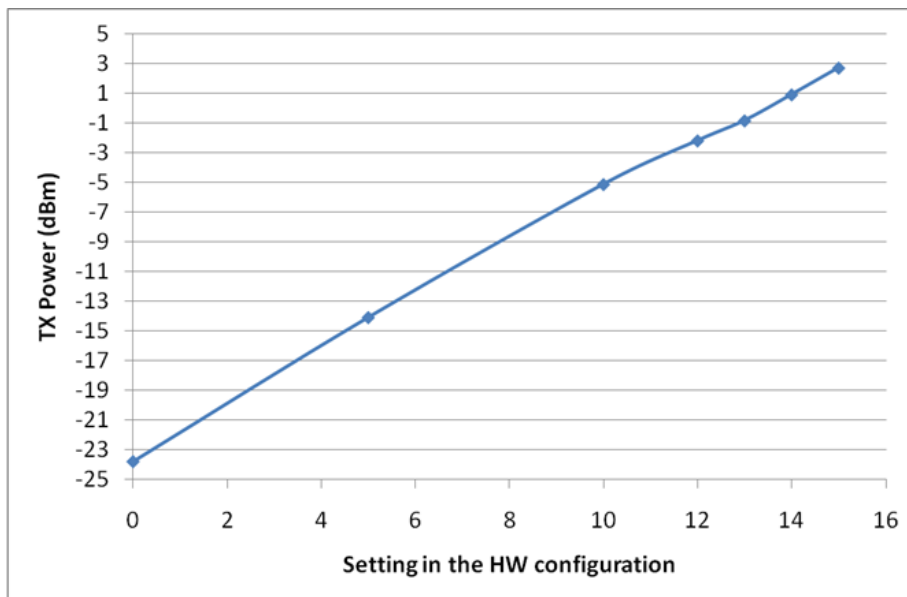


Fig. 2.6 Relación potencia de emisión real y configuración del hardware

- e) **Intervalo entre “advertisements” y canales utilizados:** Se puede determinar el mínimo y máximo tiempo entre envíos de avisos y los canales para avisos que pueden ser usados. Para ello se utiliza el método `setADVParameters(valor_mínimo, valor_máximo, máscara_canales)`. Para aplicar los cambios se debe dejar de emitir, cambiar los parámetros y volver a emitir.

El valor mínimo y el valor máximo admiten un valor entre 32 y 16384 unidades de tiempo, donde cada unidad de tiempo equivale a 0,625 μ s. De manera que los valores mínimos y máximos entre un envío y otro oscilan entre 20 ms y 10,24 s. Por defecto, el módulo envía “advertisements” cada 320 ms. Para fijar una emisión a un determinado valor se debe indicar el mismo valor para el valor mínimo y para el valor máximo. Se permite un valor mínimo y máximo diferente entre sí para que el chipset decida entre uno y otro en función de las necesidades del momento.

La máscara de canales determina qué canales están activos para enviar “advertisements”. La especificación determina que se emite por los canales 37, 38 y 39 de manera secuencial. El parámetro es un entero de 8 bits, donde los tres bits menos significativos representan cada canal. Por ejemplo el número decimal 7, en formato binario 0111, indica que los tres canales serán utilizados. El número decimal 3, en formato binario 0011, indica que los canales 38 y 37 serán utilizados, y de la misma manera para cualquier otro valor entre 1 y 7.

2.5.2 Tipos de “advertising” y estructura de los paquetes de datos

Según los parámetros utilizados en los métodos *setDiscoverableMode()* y *setConnectableMode()* tendremos diferentes tipos de “advertisements” y para cada tipo, la especificación Bluetooth 4.0 determina un formato del paquete de datos que se envía.

Los diferentes tipos de “advertising” se muestran en la siguiente tabla. En la primera columna se indica el nombre con el que identificamos el tipo de “advertising”, en la segunda columna se describe brevemente y en la tercera columna se detalla el procedimiento que se sigue entre el módulo BLE y un dispositivo que esté “escaneando” avisos.

Tabla 2.5 Tipos de “advertising”

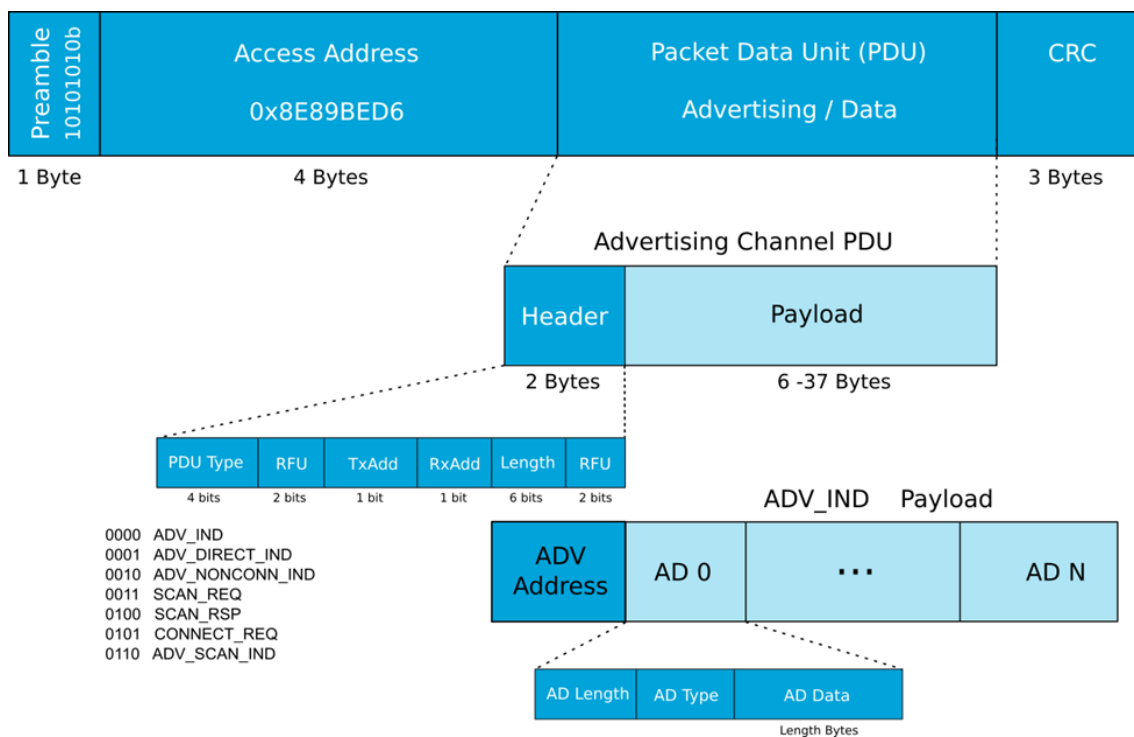
Tipo de Advertising	Características	Procedimiento seguido
Conectable indirecto	El módulo BLE puede conectarse con cualquier dispositivo	El módulo BLE envía avisos que son identificados por otros dispositivos. Una vez es encontrado, recibe una solicitud de conexión que se contesta enviando la información necesaria para establecer un emparejamiento.
Conectable directo	El módulo BLE se podrá conectar con sólo un dispositivo	El módulo BLE envía un aviso con los datos del dispositivo con el que se espera conectar. El dispositivo contesta solicitando una conexión para establecer el emparejamiento.
No conectable	El módulo BLE no se conectará con ningún dispositivo	El módulo BLE emite un aviso en modo “broadcast” sin ninguna posibilidad de que el resto de dispositivos pueda conectarse o solicitar más información.
Descubrible	El módulo BLE no se conectará con ningún dispositivo, pero es escaneable.	El módulo BLE emite como en el caso “No conectable”, pero puede recibir peticiones y contestarlas sin llegar a establecer comunicación, sin embargo puede responder con información adicional.

Los parámetros utilizados para cada tipo de “advertising” se definen en la tabla (Tabla 2.6).

Tabla 2.6 Parámetros asociados al tipo de “advertising”

Tipo de Advertising	Parámetro de conectividad	de Parámetro de visibilidad	Estructura de datos del aviso
Conectable indirecto	BLE_GAP_UNDIRECTED_CONNECTABLE		ADV_IND
Conectable directo	BLE_GAP_DIRECTED_CONNECTABLE		ADV_DIRECT_IND
No conectable	BLE_GAP_NON_CONNECTABLE		ADV_NONCONN_IND
Descubrible	BLE_GAP_NON_CONNECTABLE	BLE_GAP_USER_DATA	ADV_SCAN_IND

El paquete de avisos establecido en BLE se define en la capa de enlace. Existe un único formato de paquete y tiene la siguiente estructura [7]:

**Fig. 2.7** Formato de los paquetes de “advertising”

El preámbulo es usado para la gestión del protocolo interno. Los paquetes de “advertising” tienen como preámbulo 10101010b.

La dirección de acceso es siempre la misma para los paquetes de “advertising”: 0x8E89BED6.

La información del “advertisement” emitida en un paquete tiene una longitud de entre 2 y 39 bytes. Estos dos primeros bytes son la cabecera mientras que los bytes restantes son considerados “payload” y su longitud es de 6 a 37 bytes siendo estos 6 primeros bytes, la MAC (ADV Address) del dispositivo emisor. Esto nos deja con 31 bytes, como máximo, de libre disposición.

Cada tipo de “advertisement” tiene un “Advertising Channel PDU” diferente. Los 4 primeros bits de la cabecera del “Advertising Channel PDU” identifican el tipo de aviso que se está enviando. Por cada tipo de aviso la estructura de “payload” es la siguiente:

Tabla 2.7 Estructura de “payload”

Tipo de aviso	Longitud de “payload”	Estructura de “payload”
ADV_IND	Máximo 31 bytes	<i>AdvA</i> Dirección del dispositivo emisor (6 bytes) <i>AdvData</i> Datos que se le envían
ADV_DIRECT_IND	12 bytes	<i>AdvA</i> Dirección del dispositivo emisor (6 bytes) <i>InitA</i> Dirección del dispositivo receptor e iniciador de la petición de conexión (6 bytes)
ADV_NONCONN_IND	Máximo 31 bytes	<i>AdvA</i> Dirección del dispositivo emisor (6 bytes) <i>AdvData</i> Datos que se le envían
ADV_SCAN_IND	Máximo 31 bytes	<i>AdvA</i> Dirección del dispositivo emisor (6 bytes) <i>AdvData</i> Datos que se le envían

Finalmente, los datos que se envían forman parte de la estructura *AdvData*, la cual es una concatenación de Advertising Data Types definidos en [8] y/o diferentes tipos definidos por el desarrollador de la aplicación. Cada dato AD0, AD1, ..., ADN se envía de la forma codificada LTV, es decir, primero su longitud en bytes, luego el tipo de datos y finalmente, el valor del dato.

En un programa de Wasmote y tras haber definido el tipo de aviso utilizado, para determinar cuáles son los datos enviados se utiliza el método `setAdvData(BLE_GAP_ADVERTISEMENT, data, 31)`, donde `data` será una variable de 31 bytes que debe contener los datos en la forma LTV descrita anteriormente.

2.6 Implementación del modo “scanning” en Wasmote

Para localizar “advertisements” emitidos por otros dispositivos Bluetooth, Wasmote debe hacer un “scan” de dispositivos. El módulo BLE permite configurar la manera en la que se hace el escaneo seleccionando el modo, el intervalo y la ventana de escaneo, si es pasivo o activo, filtrando por MAC y determinando la potencia de transmisión.

2.6.1 Configuración del modo “scanning”

Las características configurables son:

- a) **Modo de escaneo:** Determina qué tipo de dispositivos pueden ser escaneados. La función que se utiliza es *BLE.setDiscoverMode(modos)* donde el parámetro modo puede ser:

Tabla 2.8 Modos de escaneo

Tipo de escaneo	Parámetro	Dispositivos escaneables
Limited discover	BLE_GAP_DISCOVER_LIMITED	Limited discoverable
General discover	BLE_GAP_DISCOVER_GENERIC	Limited discoverable Generic discoverable
Observation discover	BLE_GAP_DISCOVER_OBSERVATION	“Beacon”

- b) **Intervalo de escaneo, ventana de escaneo y tipo:** se determina con el uso del método *BLE.setScanningParameters(interval, window, tipo)*.

El intervalo de escaneo mínimo es 2,5 ms y el máximo 10,24 segundos. El valor por defecto en Wasmote es de 46,875 ms. El rango para la ventana de escaneo es de 2,5 ms a 10,24 segundos y por defecto Wasmote utiliza 31,25 ms. Sin embargo, los parámetros *interval* y *window* se expresan en unidades de 0,625 ms, de manera que los parámetros están entre el valor 4 y 13684. El valor *window* no puede ser superior al valor *interval* e igualarlos supondría un escaneo continuo.

Se puede establecer el tipo de escaneo, es decir, si es pasivo o activo. En el caso de ser pasivo, el módulo BLE escucha “advertisements” e informa a Wasmote de los dispositivos detectados con la información contenida en el “advertisement” como información sobre la potencia de emisión, MAC del nodo u otros datos. Si se hace un escaneo activo una

vez detectado un dispositivo se le puede solicitar más información no incluida en el AdvData. Para configurarlo en un modo u otro se debe usar la constante `BLE_ACTIVE_SCANNING` o `BLE_PASSIVE_SCANNING` (opción por defecto) en el parámetro *tipo*.

- c) **Filtros en la comunicación:** El módulo BLE ofrece la posibilidad de escanear sólo los nodos especificados en una “lista blanca”, definiendo una política de escaneo. El método `BLE.setFiltering(scan_policy, adv_policy, mac_filter)` admite dos valores para el parámetro `scan_policy`: `BLE_GAP_SCAN_POLICY_ALL` para que el módulo BLE escanee todos los dispositivos y `BLE_GAP_SCAN_POLICY_WHITELIST` para que el módulo BLE escanee todos los dispositivos especificados en una “lista blanca”. `whiteListAppend(mac)` permite añadir el dispositivo del cual se conoce su dirección MAC. `whiteListRemove(mac)` permite eliminar un dispositivo de la lista y `whiteListClear()` elimina la lista de manera completa. El parámetro `mac_filter` tiene dos opciones, activado o desactivado. Cuando está activo se filtran los duplicados de MAC.
- d) **Potencia de transmisión:** Durante el escaneo, solo se detectan los dispositivos que emiten con una potencia superior o igual a la definida con el método `setTXPower(integer)`, que admite como parámetro, un entero entre 0 y 15, lo que equivale a una potencia real de -23 a +3 dBm.

2.6.2 Funciones de escaneo

El módulo BLE permite escanear la red o un dispositivo concreto, analizando el “advertisement” y formateando y almacenando los datos obtenidos, siguiendo una estructura de datos predeterminada por Libelium. Los datos obtenidos pueden guardarse en la EEPROM y más tarde pueden imprimirse por el puerto USB.

El escaneo hace uso de la memoria EEPROM, en una zona reservada entre un límite inferior y uno superior, para guardar el resultado del escaneo. La estructura de datos que guarda Wasmote contiene la siguiente información: MAC del dispositivo BLE detectado, RSSI intensidad de la señal recibida, y el nombre “amigable” del dispositivo si lo tiene. El método `BLE.printInquiry()` muestra en el terminal USB los datos del último dispositivo descubierto. Los datos del “advertisement” almacenados en AdvData no son guardados en la memoria EEPROM.

En la implementación que se ha hecho de Wasmote escaneando “advertisements” se ha optado por guardar la información de los eventos detectados en la tarjeta de memoria SD, para poder así guardar el evento completo y poder realizar un estudio a partir de los datos del fichero “log.txt” generado. Para evitar el almacenamiento de datos en la EEPROM se debe

actualizar la etiqueta `ENABLE_EEPROM_SAVING` con el valor 0. Existe la posibilidad de que al guardar en la tarjeta SD generemos un retardo y se pierdan paquetes. En el capítulo de medidas se comparará el “log” generado por Waspote con el “log” generado por Wireshark para poder ver el impacto que esto supone en el escáner.

El método principal que incorpora Waspote para escanear dispositivos BLE es el método `BLE.scanNetwork(time, txpower)` donde `time` se expresa en segundos y `txpower` es un entero de las mismas características que el parámetro que se utiliza en `setTXPower(integer)`.

El tiempo máximo de escaneo es de 5 minutos, limitado por la API de Waspote para evitar un problema de “overflow” del RTC.

El método que permite escanear un dispositivo concreto es `BLE.scanDevice(mac_address, time, txpower)` y es usado para saber si un dispositivo concreto está en el área de escaneo del módulo BLE. También utiliza los parámetros `time` y `txpower` en el mismo sentido que el método `BLE.scanNetwork(time, txpower)`.

El módulo BLE permite hacer un escaneo de la red para encontrar otros módulos BLE, incluyendo el nombre “amigable” de cada módulo y por ello utiliza el modo de escaneo activo. En este caso los dispositivos localizados deben incluir en el `AdvData` el nombre “amigable” del dispositivo y si no lo hacen, el nombre “amigable” quedará vacío. El método utilizado en este caso es `BLE.scanNetworkName(time, txpower)` con los parámetros definidos de la misma manera que en los métodos de escaneo anteriores y así el nombre “amigable” se guardará en la estructura de la memoria EEPROM.

Y finalmente, una última opción de escaneo es la que permite escanear la red hasta un número determinado de dispositivos descubiertos o hasta que el límite de tiempo de escaneo ha sido alcanzado. El escaneo finaliza cuando una de las dos condiciones se cumple. Se requiere utilizar el método `BLE.scanNetworkLimited(num_max)` que a su vez llama a `BLE.scanNetworkName(time, txpower)`, donde `num_max` es el número de dispositivos descubiertos que una vez alcanzado hará que finalice el escaneo.

CAPÍTULO 3. CONFIGURACIÓN DEL ENTORNO

El entorno de trabajo que se ha diseñado se compone de los siguientes elementos:

- Waspnote con un módulo BLE instalado en el Socket0.
- Un adaptador Bluetooth USB compatible con la versión 4.0, de la marca Trust.
- Un ordenador con sistema operativo Linux con el analizador de paquetes Wireshark instalado.
- Un cable USB para conectar Waspnote al ordenador.
- El entorno de desarrollo Waspnote IDE instalado en el ordenador.

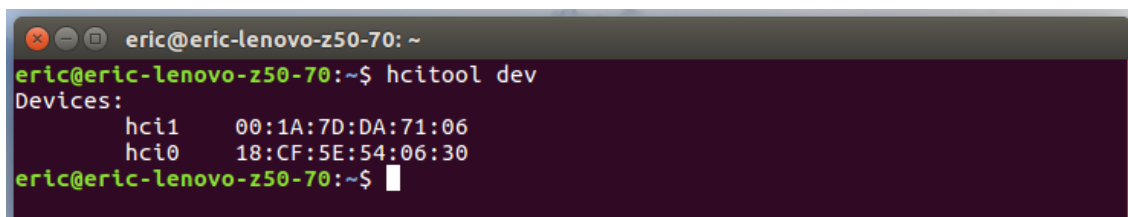
En este capítulo se explican los procedimientos seguidos para instalar y configurar los elementos necesarios del entorno de trabajo. El resto de procedimientos de configuración necesarios para cada escenario del trabajo se detallarán en el capítulo correspondiente.

3.1 Configuración del adaptador Bluetooth

El sistema operativo Linux Ubuntu requiere de la instalación de la pila del protocolo Bluetooth BLE. Una vez instalada, el ordenador podrá configurar las conexiones Bluetooth y enviar comandos al dispositivo Bluetooth mediante una utilidad del Terminal, llamada *hcitool*. Se ha de instalar la pila ofrecida por BlueZ, la cual soporta los protocolos Bluetooth y sus capas.

El proceso seguido para la instalación de la pila es el que se describe en [9].

Una vez conectado el adaptador al ordenador se ejecuta el comando *hcitool dev* para comprobar si el sistema operativo detecta el dispositivo. En la figura (Fig. 3.1) siguiente se observan dos dispositivos: el primero, hci0, corresponde al chip Bluetooth incorporado en el ordenador, el segundo, hci1, corresponde al adaptador Bluetooth Trust que se va a utilizar, ya que tiene mejores prestaciones que el chip incorporado.



```
eric@eric-lenovo-z50-70: ~  
eric@eric-lenovo-z50-70:~$ hcitool dev  
Devices:  
    hci1    00:1A:7D:DA:71:06  
    hci0    18:CF:5E:54:06:30  
eric@eric-lenovo-z50-70:~$
```

Fig. 3.1 Dispositivos Bluetooth

Se puede observar también, la dirección BD ADDR (Bluetooth Device Address) de cada dispositivo.

Desde el Terminal de Linux se ejecutan una serie de comandos para configurar el dispositivo Bluetooth en modo scan o en modo emisión, dependiendo del escenario que se describirá en los capítulos siguientes. Esta configuración se obtiene tras la ejecución de una serie de comandos HCI. Para facilitar su ejecución se han creado varios archivos script de Linux descritos en [9] para configurar el adaptador USB en el modo de trabajo deseado.

3.2 Configuración del analizador de paquetes Wireshark

Para visualizar el tráfico de paquetes en tiempo real se utiliza el programa analizador de tráfico de paquetes, Wireshark, que se debe instalar siguiendo las instrucciones en la página web [10] del desarrollador.

Una vez ejecutado el programa, para poder analizar el tráfico se debe seleccionar el dispositivo a analizar, en el entorno de trabajo corresponde al dispositivo Bluetooth hci1, que se muestra en Wireshark como dispositivo Bluetooth1. Para que los dispositivos Bluetooth sean accesibles por Wireshark se requiere configurar los permisos de acceso mediante Wireshark a los dispositivos Bluetooth. La secuencia de comandos necesaria para ello se describe en [9].

3.3 Configuración de Waspote para la generación de logs

Para poder medir las prestaciones de la Waspote en modo escáner se necesita la generación de un “log” con los “advertisements” recibidos. Así es posible estimar si hay una pérdida de paquetes.

Como se ha explicado anteriormente, por defecto, Waspote guarda información del emisor de “advertisements” en la EEPROM. El sistema almacena una lista de dispositivos y conforme recibe “advertisements” va actualizando la información relacionada con cada dispositivo, como es la MAC, el RSSI y el nombre del dispositivo.

Este método no almacena los datos del advertisement ni guarda los paquetes recibidos ni el tiempo en el que han sido descubiertos. Por esta razón se ha modificado la API para generar un “log” guardado en la tarjeta de memoria SD.

La información que llega a Waspote viene dada por el chipset del módulo BLE, de manera que no podemos recibir el paquete Bluetooth y ver sus 47 bytes en plano. El chipset recibe este paquete Bluetooth y envía la información mediante un evento al procesador de la Waspote. Este evento es el definido en la API como BLE_EVENT_GAP_SCAN_RESPONSE.

CAPÍTULO 4. IMPLEMENTACIÓN DEL MODO EMISIÓN

En este escenario Wasmote emitirá avisos que serán analizados con varias herramientas según el entorno instalado. Los elementos requeridos para implementar la plataforma en modo de emisión son los mostrados en la figura (Fig. 4.1).

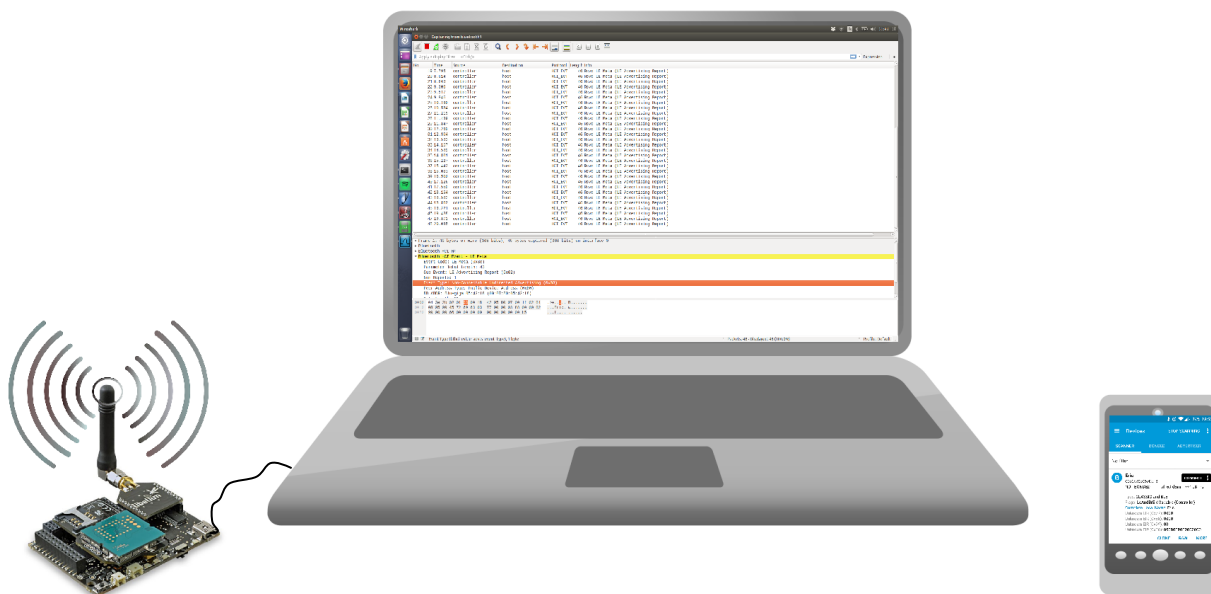


Fig. 4.1 Esquema del escenario utilizado

Wasmote está conectada al ordenador a través del puerto mini USB, sobre el que se tiene un puerto serie virtual configurado a 115.200 baudios, 8 bits, 1 bit stop y sin paridad. El puerto USB alimenta a Wasmote y sirve para interactuar desde el entorno de desarrollo Wasmote IDE.

El ordenador utilizado tiene sistema operativo Ubuntu 16.04 LTS y tiene instalada la pila Bluetooth de BlueZ [11]. Además, está conectado un adaptador Bluetooth de la marca Trust que hará las funciones de receptor de Bluetooth en este caso.

En el ordenador está instalado el programa Wireshark para analizar los datos recibidos por el adaptador Trust, mostrarlos en pantalla y generar un “log” de la comunicación establecida en un periodo de tiempo.

También se dispone de un teléfono Android con la aplicación nRF Master Control que funciona como receptor de los avisos y muestra el estado de la conexión y los datos recibidos.

Se pretende que Wasmote se comporte como un dispositivo “beacon” para emitir paquetes de “advertisement” siguiendo el protocolo BLE Advertising.

El programa desarrollado permite:

- Configurar Wasmote en modo emisión de “advertisement”.
- Configurar Wasmote para reducir el consumo al máximo.
- Enviar comandos por el serial monitor para cambiar diferentes parámetros de configuración de los advertisement, como el tiempo entre “advertisements” o la información contenida en él.

Tal y como se describe en el Anexo 2, un programa de Wasmote tiene extensión .pde y se compone de dos funciones: *setup()* que se ejecuta al iniciar o reiniciar Wasmote y *loop()* que se ejecuta de manera indefinida. Las configuraciones generales se incluyen en la función *setup()* y la emisión de paquetes en la función *loop()*.

El programa permite la interacción con el usuario, de manera que pueda seleccionar diferentes parámetros de configuración de la emisión. Una vez cargado el programa a través del entorno de desarrollo, el programa comienza su ejecución y queda a la espera de la interacción del usuario.

Los datos enviados siguen la estructura de datos propuesta para el entorno de circulación definido en [12]. La siguiente tabla muestra, con valores de ejemplo, la estructura definida:

Tabla 4.1 Estructura de datos del “advertisement”


Length	Type	Value
2 bytes	0x01 – Flags	0x08
5 bytes	0x09 – Complete name	0x45, 0x72, 0x69, 0x63 (Eric en ASCII)
3 bytes	0x77 – Señal de tráfico	0x00, 0x00
3 bytes	0x88 – Vía	0x00, 0x00
2 bytes	0x99 – Estado de la vía	0x00
9 bytes	0x66 – Ubicación	0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

A través del cuadro de texto y del botón “Enviar” de la ventana del “Serial Monitor” del entorno Wasmote IDE, podemos enviar comandos a Wasmote. Se dispone de las siguientes opciones:

- “Set interval advertisement”: determina el tiempo en milisegundos que transcurre entre envíos del aviso y para ello se fija el intervalo mínimo y el intervalo máximo con el mismo valor. Para “advertisement” del tipo no

conectable (ADV_NONCONN_IND) se debe usar un valor entre 100 [13] ms y 10240 ms.

- "Select channels": permite seleccionar los canales de emisión.
- "Set flags": permite asignar el valor de la etiqueta "Flags" [8] incluida en el "Advertisement Data".



```
E#
Configure advertisement:

    1 - Set interval advertisement
    2 - Select channels
    3 - Set flags

Press any other key to start
Set advertisement interval in ms, e.g: 100/ = 100 ms --- 200 ms 1
Configure advertisement:

    1 - Set interval advertisement
    2 - Select channels
    3 - Set flags

Press any other key to start
Set advertisement channels, e.g: 7 = 39,98,37 2
0 0 0 0 0 1 0 0 - 39
0 0 0 0 0 0 1 0 - 38
0 0 0 0 0 0 0 1 - 37

Set advertisement channels: 5
Channel 37 selected
Channel 39 selected

Configure advertisement:

    1 - Set interval advertisement
    2 - Select channels
    3 - Set flags

Press any other key to start
Set advertisement flags, e.g: 08 = Simultaneous LE and BR/EDR to Same Device Capable (Controller)
0 0 0 0 0 0 0 1 - LE Limited Discoverable Mode
0 0 0 0 0 0 1 0 - LE General Discoverable Mode 3
0 0 0 0 0 1 0 0 - BR/EDR Not Supported
0 0 0 0 1 0 0 0 - Simultaneous LE and BR/EDR to Same Device Capable (Controller)
0 0 0 1 0 0 0 0 - Simultaneous LE and BR/EDR to Same Device Capable (Host)

Set advertisement flags: 8
Simultaneous LE and BR/EDR to Same Device Capable (Controller)

Configure advertisement:

    1 - Set interval advertisement
    2 - Select channels
    3 - Set flags

Press any other key to start
Transmitting... 4

    A - Stop advertisements
    B - Setting advertisement interval
    C - Setting data
    D - Start advertisements

Send any key to stop
```

Desplazamiento automático

No hay fin de línea

115200 baud

Fig. 4.2 Ejemplo del programa de emisión

1. Se muestra la introducción de 200 ms como intervalo de “advertisements”.
2. Se muestra el resultado de la selección de los canales 37 y 39.
3. Se muestra el resultado de la modificación de la etiqueta “Flags”.
4. Se inicia la transmisión pulsando cualquier tecla y se queda a la espera de que se pulse cualquier tecla para que se pare la transmisión.

Al pulsar cualquier otra tecla diferente a 1, 2 o 3 se iniciará la ejecución de la transmisión. A partir de ese momento podremos analizar con Wireshark los diferentes paquetes de “advertisement” que se envían y con nRF Master Control podemos ver el intervalo de emisión y el contenido de los datos del “advertisement”.

El código fuente del programa `sendAdv.pde` se ha transcrito en el Anexo 3. De manera esquemática el flujo del programa es el siguiente:

- a) Se inicializan las variables internas del programa y se asigna el contenido de los datos del “advertisement” almacenados en una variable de tipo vector de 31 bytes llamada `data[31]`.
- b) En la función `setup()` se activa el USB con el método `USB.ON()` y se desactivan los sensores de Waspote para economizar energía con el método `PWR.switchesOFF(SENS_OFF)`.
- c) La función `loop()` se ejecuta indefinidamente. Consta de dos secciones:
 1. Sección de configuración de parámetros: codificada en la función `config()`. Se ejecuta la primera vez y siempre y cuando no se haya solicitado el inicio de la transmisión.
 2. Sección de transmisión de mensajes: Cuando se pulsa una tecla diferente a las opciones de configuración entonces se inicia la transmisión con el intervalo especificado y con el resto de parámetros seleccionados, mientras no se pulse una tecla para parar la transmisión. La función que transmite mensajes se llama `runAdv()`. Si se ha solicitado parar la transmisión se ejecuta `BLE.OFF()` pudiéndose transmitir de nuevo pulsando una tecla cualquiera.

Este programa no es apto para un entorno de explotación ya que lo ideal es que en la función `setup()` se asignen los parámetros de transmisión y la función `loop()` ejecute las instrucciones para transmitir de manera indefinida el “advertisement”. Este programa se obtendría con la eliminación de la sección que permite interactuar con el usuario. Sin embargo, se ha optado por permitir que el usuario actualice los parámetros de la transmisión porque de esa manera se podrán analizar y medir la transmisión en diferentes situaciones que en capítulos posteriores se expondrán.

El flujo de programa de manera gráfica está representado en el Anexo 4.

A continuación, se destacan algunas funciones que explican en detalle las particularidades que ofrece la API de Waspote para gestionar la alimentación del módulo BLE y para determinar el modo de transmisión y el tipo de “advertisement” que se envía.

La función *USB.ON()* abre UART0¹ para comunicarse con el conversor serie-USB del fabricante FTDI incluido en la placa de Waspote y de esa manera se puede utilizar la ventana “Serial Monitor” de Waspote IDE para que el usuario interactúe con el programa.

La función *config()* muestra en pantalla las opciones de configuración que se pueden seleccionar y se espera la introducción de una opción por parte del usuario. Para mostrar las opciones posibles se utiliza el método *USB.println(“texto”)*, para detectar si se ha pulsado una tecla se utiliza el método *USB.available()* y para leer el valor introducido se utiliza *USB.read()*. Se utiliza la clase USB que representa la ventana “Serial Monitor” de Waspote IDE. Según el valor introducido se ejecutarán las funciones que solicitan al usuario el intervalo de emisión, los canales de “Advertising” activados y los datos de la etiqueta “Flag”, mediante las funciones *setIntervals()*, *setChannels()* y *setFlags()* que almacenarán los valores introducidos en las variables *interval*, *chan* y *data[2]* respectivamente.

Cuando se ha seleccionado un valor diferente a las opciones de configuración posibles se ejecuta *runAdv()*, función que ejecuta lo necesario para poder transmitir mensajes de “advertisements”. Lo primero que se hace es ejecutar *BLE.ON(SOCKET0)* para alimentar el módulo BLE y abrir la conexión con Waspote a través de UART0. Para actualizar los parámetros de transmisión el dispositivo no debe emitir y por ello se le sitúa en un estado NO descubrible usando el método *BLE.setDiscoverableMode(BLE_GAP_NON_DISCOVERABLE)*. A continuación, se actualizan los parámetros de intervalo de transmisión y canales activos con el método *BLE.setAdvParameters(interval, interval, chan)* y se actualizan los valores de la etiqueta “Flags”. Con el método *BLE.setAdvData(BLE_GAP_ADVERTISEMENT, data, 31)* se indican los datos que se desean enviar. Y, por último, se le indica que emita “advertisement” con el método *BLE.setDiscoverableMode(BLE_GAP_USER_DATA)*.

Como el módulo BLE dispone de un modo de funcionamiento de bajo consumo, una vez iniciada la transmisión, se ejecuta el método *BLE.sleep()* para entrar en el mejor modo de bajo consumo disponible. Existen tres modos de bajo consumo, administrados automáticamente por el módulo, por lo que el usuario no tiene ningún control sobre ellos. El módulo ha sido optimizado por el fabricante para seleccionar el mejor modo de potencia para cada condición [14].

El modo de funcionamiento de bajo consumo es independiente de los modos de ahorro de energía de Waspote.

Finalmente, si se ha pulsado una tecla cualquiera durante la transmisión, el programa ejecuta el método *BLE.OFF()* con lo que apaga el módulo, cierra la conexión con UART y queda a la espera para volver a transmitir pulsando otra tecla cualquiera.

¹ Ver la figura del apartado 2.8.2 del Anexo 2

CAPÍTULO 5. IMPLEMENTACIÓN DEL MODO RECEPCIÓN

En este capítulo se aborda el escenario en que Waspote y el módulo BLE son configurados en modo escaneo para analizar los “advertisements” recibidos. Los elementos requeridos para implementar la plataforma en modo de recepción son los mismos que utilizamos en la implementación de la plataforma en modo emisión, con la salvedad, de que el adaptador Bluetooth conectado al ordenador hace las funciones de emisor y Waspote y el módulo BLE hacen las funciones de recepción.

En este escenario, el programa Wireshark no se utilizará, ya que el receptor es Waspote y se ha desarrollado una rutina dentro del código fuente que permite al programa guardar los datos recibidos en la memoria SD.

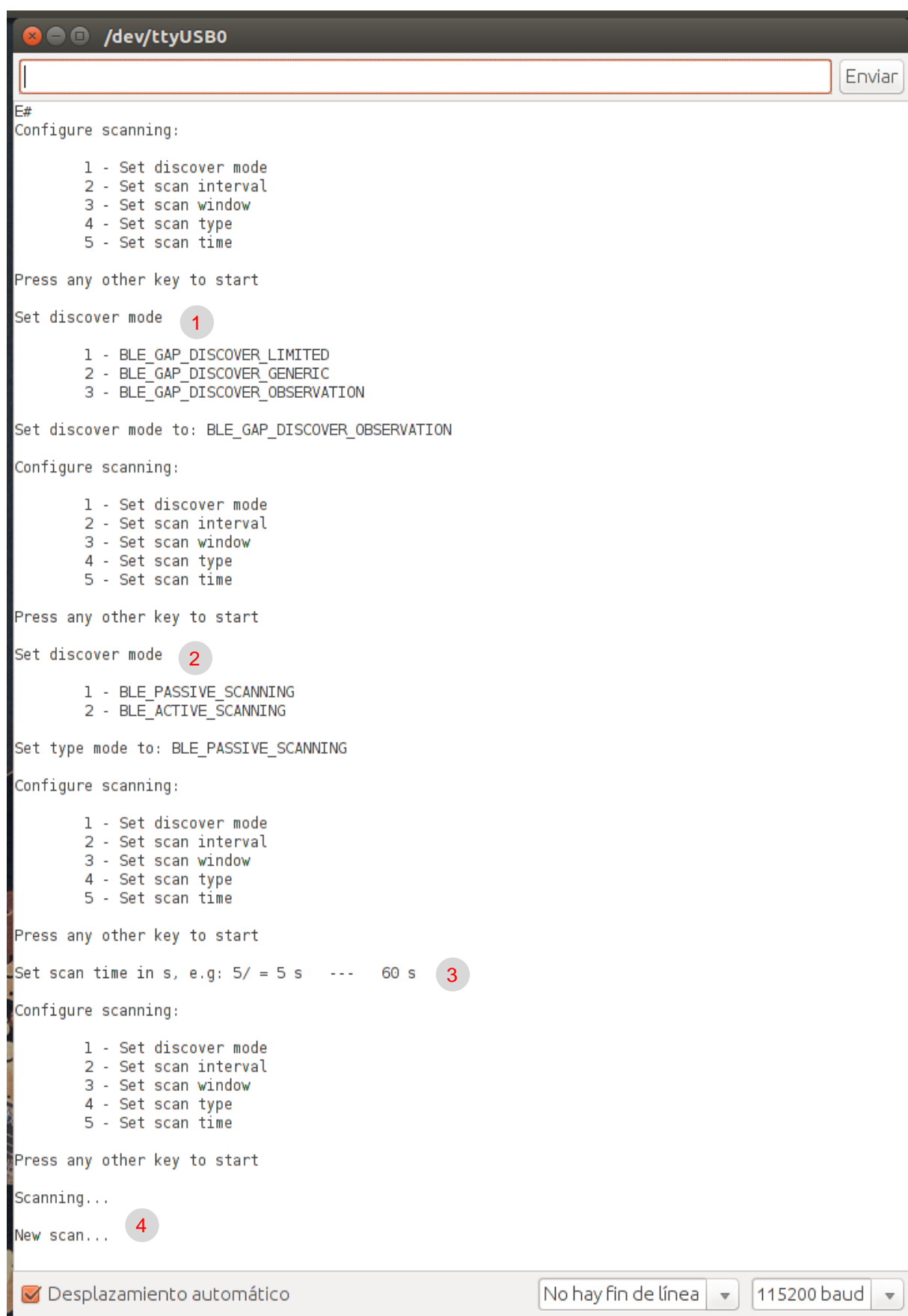
El escenario está planteado para que Waspote escanee dispositivos que emitan “advertisements” siguiendo el protocolo Bluetooth 4.0.

El programa desarrollado permite:

- Determinar el modo de escaneo, es decir, seleccionar el “discover mode”.
- Seleccionar el intervalo, la ventana y la duración de escaneo.
- Seleccionar el tipo de escaneo, activo o pasivo.
- Generar un “log” de los paquetes recibidos.

Se disponen de las siguientes opciones:

- “Set discover mode”: Permite modificar el modo de escaneo entre “discover limited”, “discover generic” y “discover observation”.
- “Set scan interval”: determina el tiempo en milisegundos para el intervalo de escaneo, que debe comprender un valor entre 2,5 ms y 10240 ms.
- “Set scan window”: determina el tiempo en milisegundos para la ventana de escaneo, debe comprender un valor entre los márgenes del “scan interval” y nunca puede ser superior a este.
- “Set scan type”: Permite cambiar el tipo de escáner. Emite un “scan request” pidiendo más información a los dispositivos que lo permitan en caso de ser escáner de tipo activo.
- “Set scan time”: Asignar el tiempo de escaneo en segundos.

**Fig. 5.1** Ejemplo del programa de recepción

En la figura (**Fig. 5.1**) se observa cómo se han modificado los parámetros de ejecución del escaneo. Una vez seleccionada la opción deseada podemos pulsar cualquier tecla para iniciar el escaneo. Si el periodo de tiempo de escaneo no se ha superado podemos pulsar cualquier otra tecla para finalizar anticipadamente el escaneo de Wasmote.

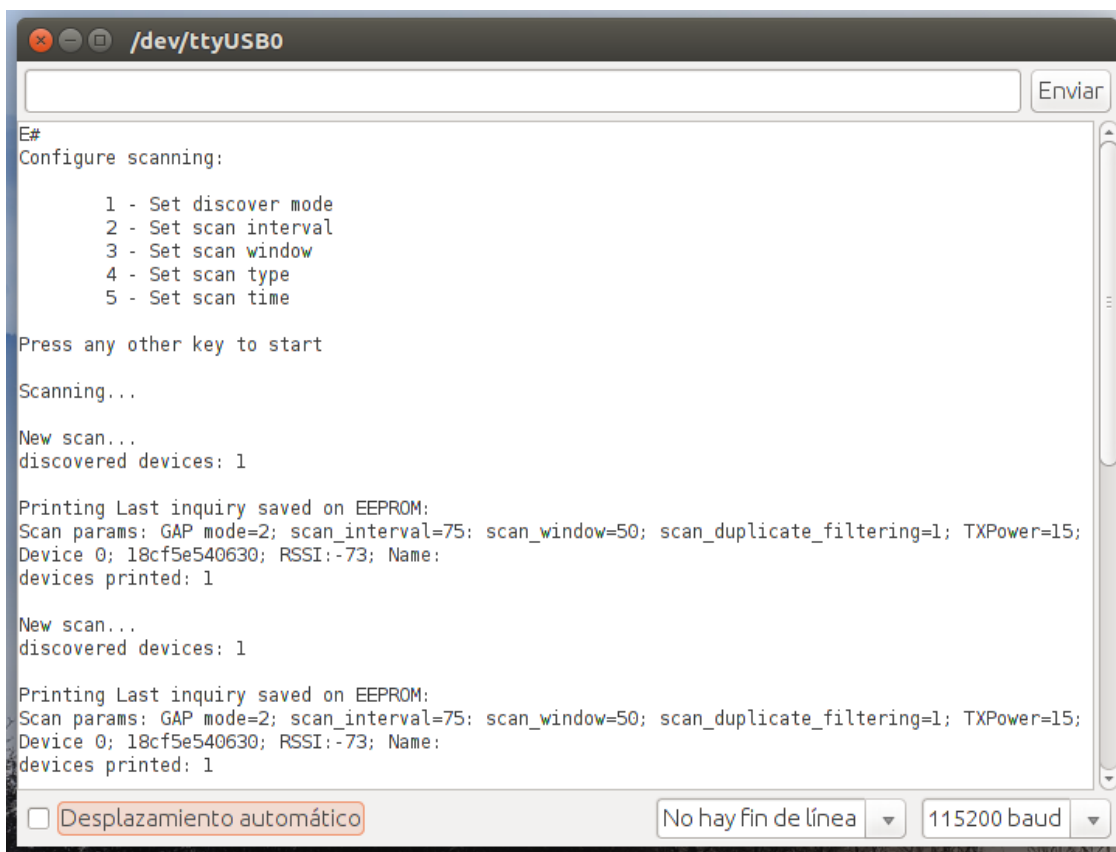
1. Se muestra como se ha seleccionado el modo de escaneo “observation” para poder descubrir los dispositivos que emiten “advertisements”
2. Se muestra la selección de un escaneo pasivo.
3. Se fija el tiempo máximo de escaneo del módulo BLE.
4. Se inicia el escaneo y se genera el “log” en la SD.

El programa verifica que el valor de la ventana de escaneo no sea superior al intervalo de escaneo. El valor por defecto del intervalo es de 75 ms y de la ventana de escaneo es de 50 ms.

El código fuente del programa scanAdv.pde se ha transcrito en el Anexo 3. De manera esquemática el flujo del programa es el siguiente:

- a) En la función *setup()* se activa el USB con el método *USB.ON()* y se desactivan los sensores de Wasmote para economizar energía con el método *PWR.switchesOFF(SENS_OFF)*.
- b) A partir de ese momento la función *loop()* se ejecuta indefinidamente y consta de varias secciones:
 1. Sección de configuración de parámetros, codificada en la función *config()*. Se ejecuta la primera vez y siempre y cuando no se haya solicitado el inicio de la transmisión. La función *config()* gestiona el menú por el que se solicitan los datos que se quieren configurar, los envía a Wasmote a través de la función *confScan()*. Se inicializan las variables internas del programa y los parámetros configurables con los siguientes valores por defecto:
 - i. Discover mode = BLE_GAP_DISCOVER_OBSERVATION.
 - ii. Interval scan = 75 unidades de 0,625 ms.
 - iii. Windows scan = 50 unidades de 0,625 ms.
 - iv. Scan Type = BLE_PASSIVE_SCANNING.
 2. Sección de escaneo de mensajes y generación de “log” incluida en la rutina *runScan()*. Esta sección se ejecuta en un bloque indefinido que es interrumpido cuando se ha superado el parámetro *time* utilizado al llamar al método *BLE.scanNetwork(time)*. Una vez finalizado el tiempo de escaneo se muestra en la consola el número de dispositivos detectados y se vuelca el contenido de la memoria EEPROM o bien se genera en la SD el “log” correspondiente.

A continuación, se muestra una figura con el resultado obtenido en la ventana del “serial monitor”. En este caso se guarda la información en la EEPROM y con la función *BLE.printInquiry()* se nos muestra la última información almacenada.



The screenshot shows a serial monitor window titled "/dev/ttyUSB0". The window contains the following text:

```
E#
Configure scanning:
    1 - Set discover mode
    2 - Set scan interval
    3 - Set scan window
    4 - Set scan type
    5 - Set scan time

Press any other key to start

Scanning...

New scan...
discovered devices: 1

Printing Last inquiry saved on EEPROM:
Scan params: GAP mode=2; scan_interval=75; scan_window=50; scan_duplicate_filtering=1; TXPower=15;
Device 0; 18cf5e540630; RSSI:-73; Name:
devices printed: 1

New scan...
discovered devices: 1

Printing Last inquiry saved on EEPROM:
Scan params: GAP mode=2; scan_interval=75; scan_window=50; scan_duplicate_filtering=1; TXPower=15;
Device 0; 18cf5e540630; RSSI:-73; Name:
devices printed: 1
```

At the bottom of the window, there is a checkbox labeled "Desplazamiento automático" which is unchecked. To its right are two dropdown menus: "No hay fin de línea" and "115200 baud".

Fig. 5.2 Ventana del serial monitor con información de los datos guardados en la EEPROM

El flujo de programa de manera gráfica está representado en el Anexo 4.

CAPÍTULO 6. MEDIDAS

En este capítulo se explica el método para realizar las medidas de corriente y así poder comprobar y analizar el consumo de la Waspote. Para ello se ha utilizado el diseño de un sensor de corriente de Texas Instrument [15]. La siguiente gráfica muestra la función V_{OUT} en función de una I_{IN} teórica.

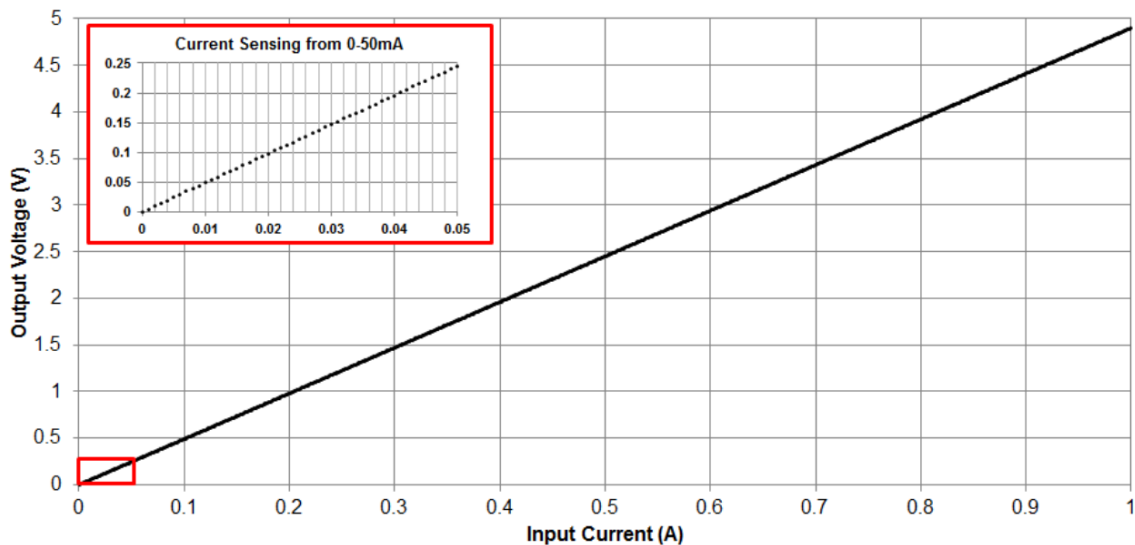


Fig. 6.1 Función transformada V_{OUT} vs I_{IN} del sensor de corriente

La salida del sensor de corriente se muestra en el osciloscopio y éste envía la información a un ordenador al cual está conectado vía IP. Un programa desarrollado en Matlab obtiene la información para permitir su posterior análisis. La imagen siguiente muestra el montaje del sensor de corriente:

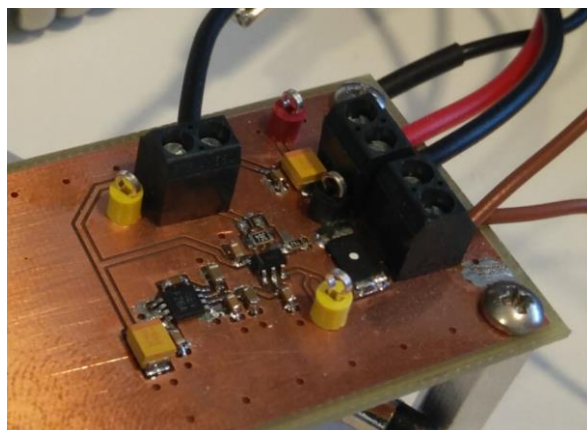


Fig. 6.2 Montaje del sensor de corriente

En las gráficas obtenidas con el osciloscopio se muestra el consumo durante un tiempo determinado pero no se muestra el valor de éste. Con Matlab se determina el valor del consumo en voltios y se convierte a amperios siguiendo la función transformada V_{OUT} vs I_{IN} del sensor de corriente (**Fig. 6.1**). El valor por el que se debe multiplicar el consumo en voltios es la pendiente, cuyo valor aproximado es de 0.2038.

6.1 Consumo en emisión

Se han medido los niveles de tensión durante la emisión de “advertisements”. Inicialmente se ha obtenido el consumo global, es decir, el de la Waspote junto con el módulo. En una segunda prueba se ha medido el consumo independiente del módulo BLE. En ambos casos el módulo BLE consume lo mínimo posible gracias a la función *BLE.sleep()*.

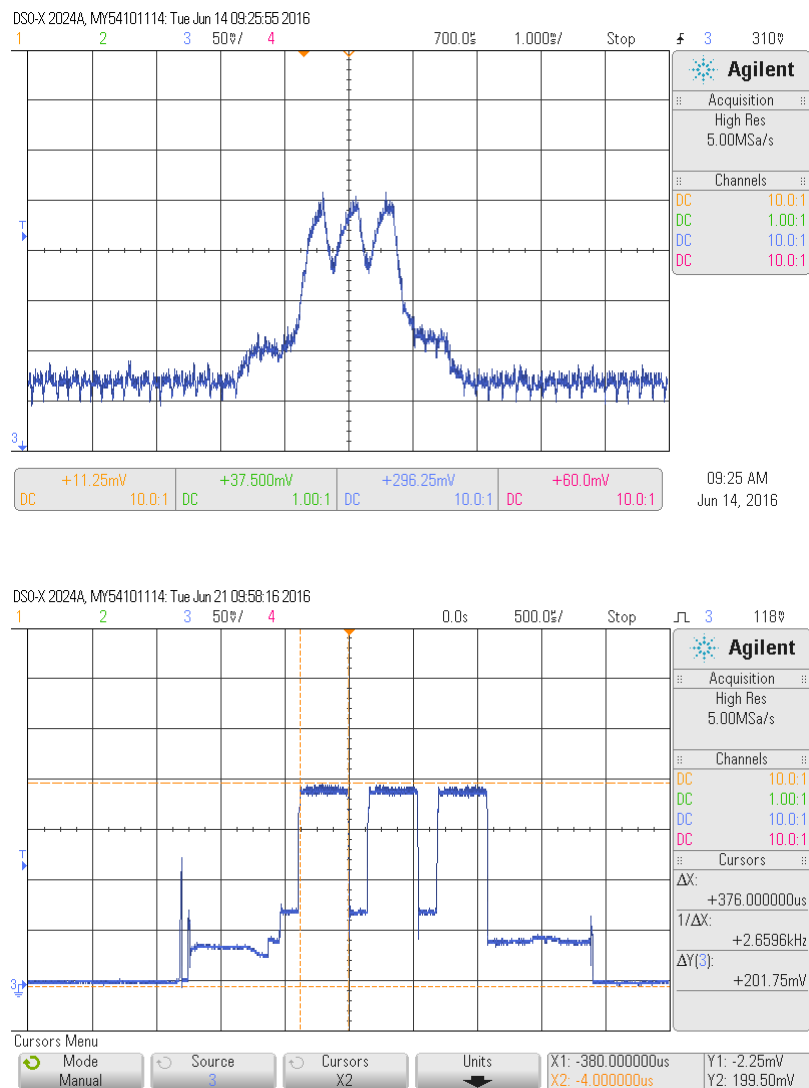


Fig. 6.3 Consumo durante la emisión de “advertisements”: Arriba (Waspote + módulo BLE), Abajo módulo BLE

Se puede observar como en la primera gráfica la forma del “advertisement” se enmascara debido al consumo de Wasmote. Tenemos un consumo máximo durante la emisión, de 70mA, algo superior, a lo que corresponde a los valores teóricos proporcionados por Libelium, teniendo Wasmote un consumo medio de 15mA más los 36mA que consume el módulo BLE durante la emisión.

Durante el periodo de no emisión, existe un consumo alrededor de los 35mA, valor superior a los 15mA de consumo esperados para la Wasmote en estado ON.

En cuanto a la segunda gráfica podemos apreciar un consumo muy limpio, donde se diferencia claramente el envío del advertisement por el canal 37, 38 y 39. Ambos pulsos duran lo mismo y tienen un consumo de 36mA. Coincide con el consumo medio en emisión del módulo BLE.

Podemos apreciar también, como en este caso durante el inicio y final de la emisión del advertisement se produce una bajada considerable de consumo hasta llegar a “ground”, es decir, a tierra, consumo 0mA o muy cercano a 0mA.

Si comparamos la gráfica del módulo BLE con la **Fig. 2.5** proporcionada por Libelium, podemos apreciar como la forma general es parecida y el consumo mínimo y máximo coincide. A pesar de ser una gráfica retocada es una buena referencia.

A continuación, se exponen las gráficas obtenidas después de diversas configuraciones del programa en modo emisión.

6.1.1 Configuración de los canales

Mediante el programa desarrollado para la emisión de “advertisements”, podemos elegir los canales por los que emitir. En la siguiente gráfica (**Fig. 6.4**) se muestra el comportamiento en cuanto a consumo cuando dos canales son seleccionados. Se aprecian los dos picos correspondientes a la emisión del “advertisement” por solo dos canales.

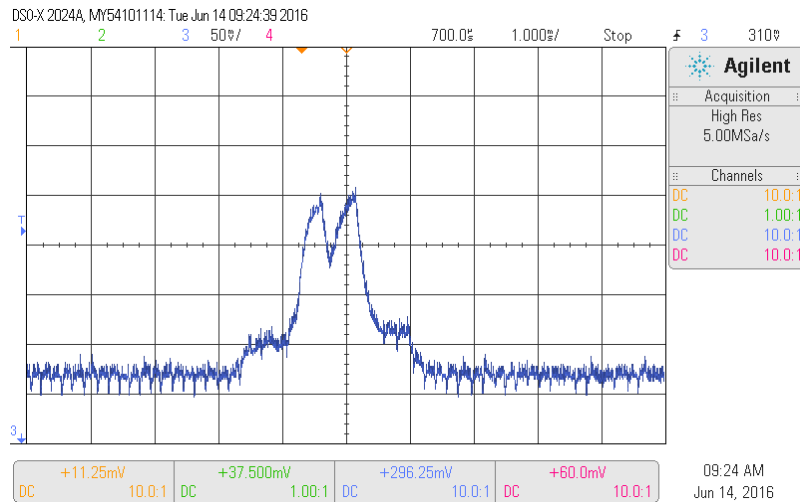


Fig. 6.4 Consumo de un “advertisement” emitido por dos canales

6.1.2 Configuración de los datos

El tiempo de emisión del “advertisement” variará en función de la longitud de los datos. En un escenario donde se pretende un consumo mínimo debemos conocer cómo afecta esta longitud en el tiempo que estamos emitiendo y por tanto en el consumo.

La longitud mínima para un advertisement es de 19 bytes (1 byte de preámbulo + 4 bytes de dirección de acceso + 2 bytes de cabecera + 6 bytes de MAC + 3 bytes de LTV + 3 de CRC). En un entorno donde un solo LTV contiene la información, el tiempo mínimo de emisión en cada canal es de 152 μ s. Teniendo en cuenta la tasa de velocidad Bluetooth, 1 Mbit/s.

Por otra parte, la longitud máxima es aquella donde todo el “payload” es utilizado, es decir, los 31 bytes libres. La longitud del paquete es por tanto de 47 bytes y esto equivale a 370 μ s de tiempo de emisión.

Las siguientes gráficas (**Fig. 6.5 y Fig. 6.6**) muestran ambos supuestos:

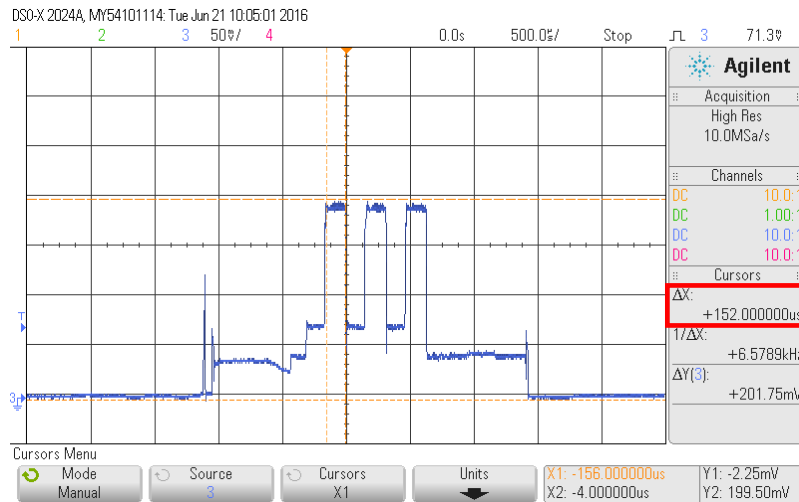


Fig. 6.5 Consumo de un advertisement con 3 bytes de *AdvData*

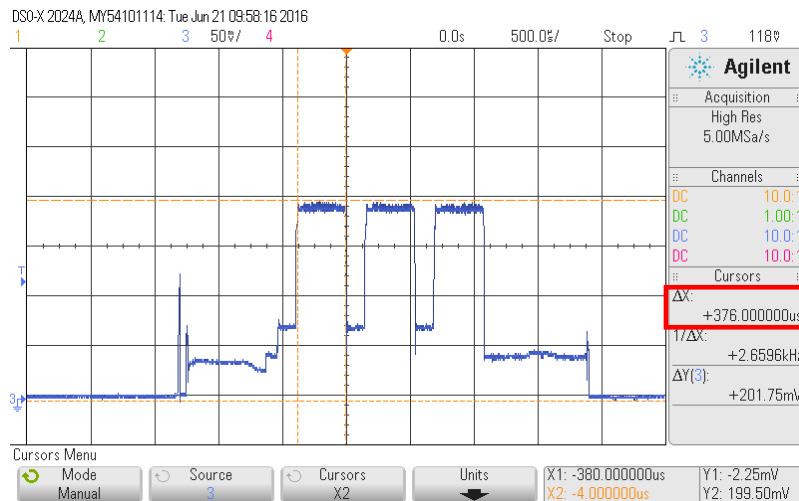


Fig. 6.6 Consumo de un advertisement con 31 bytes de *AdvData*

6.1.3 Configuración del intervalo de emisión

Las figuras **Fig. 6.7** y **Fig. 6.8** muestran una serie de “advertisements” emitidos con 100 ms de intervalo de emisión. En la primera, se muestra el consumo general, de manera, que nos aparecen pequeñas subidas de consumo periódicas debidas a la Waspote. Se puede apreciar como la distancia entre “advertisements” es la configurada, más 0 – 10 ms de tiempo aleatorio añadido para evitar colisiones.

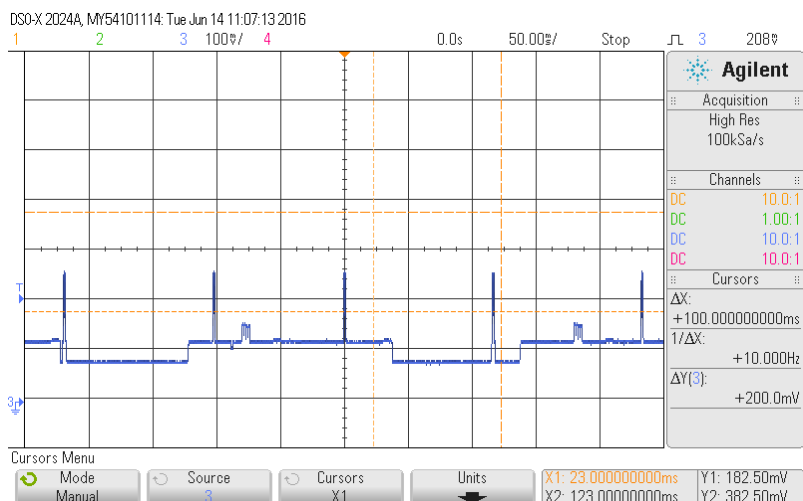


Fig. 6.7 “advertisements” con intervalo de 100 ms, consumo general

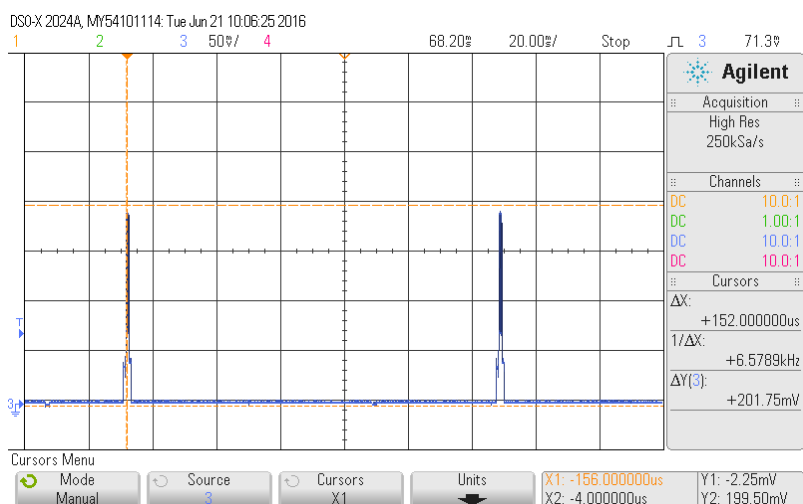


Fig. 6.8 “advertisements” con intervalo de 100 ms, consumo módulo BLE

En esta gráfica se puede apreciar como entre los momentos de emisión, el módulo BLE tiene un consumo muy cercano a 0mA.

6.1.4 Configuración del modo sleep de Waspote

Se requiere del mínimo consumo por parte de la plataforma, de manera que debemos configurar no solo el modo “sleep” del módulo BLE sino el modo de bajo consumo para Waspote. Debido a que ninguna de las interrupciones asíncronas resulta de utilidad en nuestro escenario real, utilizamos interrupciones síncronas, es decir, las interrupciones generadas por el RTC o el

WTD. Con estas interrupciones se puede dejar la Wasmote en un estado de bajo consumo durante un tiempo.

En una primera prueba, se ha configurado la Wasmote para que esté en modo “sleep” durante dos segundos, despierte y emita en modo normal durante otros dos segundos y así sucesivamente. Para no perder la emisión de “advertisements” durante el modo “sleep” se apaga toda la Wasmote, a excepción del Socket0, de manera que el módulo BLE sigue estando alimentado y como se ha configurado su chipset en la primera ejecución del programa, éste sigue emitiendo mientras tiene alimentación.

En la siguiente figura se aprecia lo anteriormente descrito.

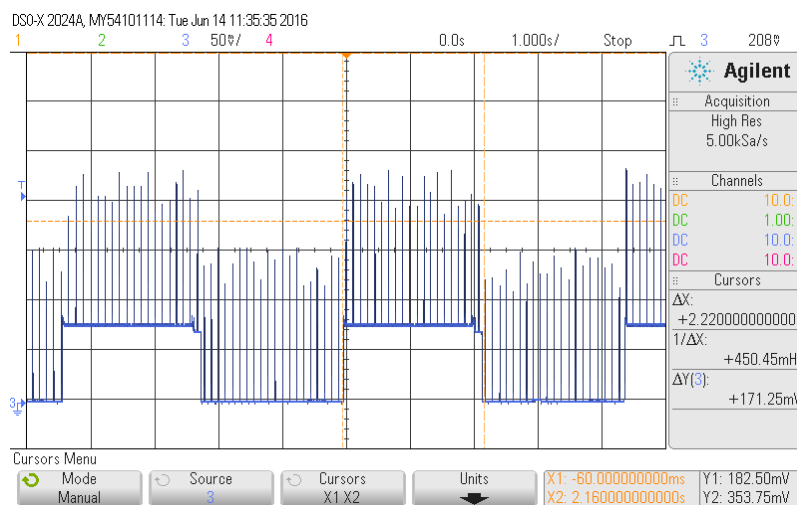


Fig. 6.9 Consumo de la Wasmote durante la emisión de “advertisements” cada 100 ms en ciclos de cambio de modo de 2 s

En la gráfica se puede apreciar como en los ciclos de “sleep” el consumo es cercano a los 0mA, a excepción de las emisiones de los “advertisements”. Es decir, en un “sleep” prácticamente infinito tendríamos un consumo muy parecido al del módulo BLE individualmente. Esto no es factible en un caso real ya que Wasmote debería despertar en función de ciertas interrupciones para, por ejemplo, poder cambiar los datos de emisión cuando el administrador del sistema así lo considere. En un entorno en el que el “beacon” debe estar escuchando y emitiendo a la vez dado que su información debe variar en función de lo que escuche, la gráfica no demuestra un caso de uso real, pero demuestra el bajo consumo de Wasmote.

En una segunda prueba, con un ciclo de cambio de modo inferior al anterior, concretamente 500 ms, y apagando y encendiendo la emisión en cada modo, obtenemos un resultado algo diferente.

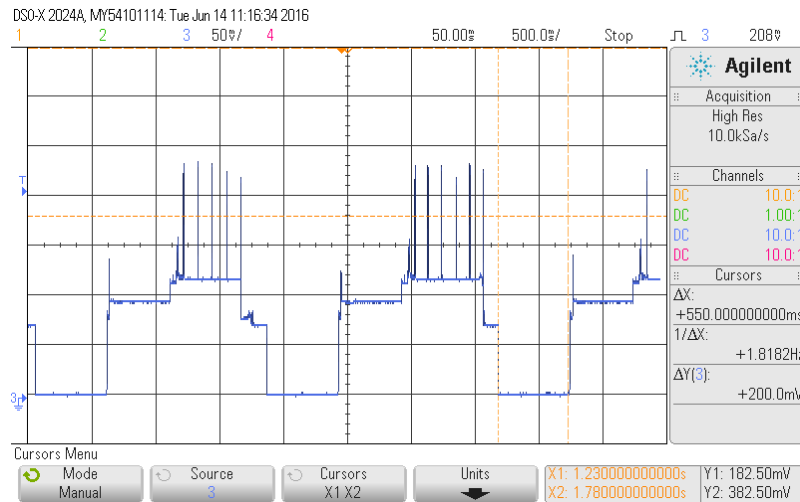


Fig. 6.10 Consumo de la Wasmote durante la emisión de “advertisements” cada 100 ms en ciclos de cambio de modo de 500 ms

En la gráfica se aprecia como la transición del modo sleep al modo normal no es instantánea, sino que existen 500 ms en los que Wasmote se inicializa e indica al módulo que vuelva a emitir. Esto no se producía anteriormente ya que no dejábamos de emitir en ningún momento.

6.2 Consumo en recepción

Se han medido los niveles de tensión para Wasmote en modo recepción. Con el programa desarrollado se han configurado los valores de *scan interval* y *window interval*. Esta configuración se ha combinado con la función *BLE.sleep()* para que el módulo BLE consuma lo mínimo posible.

6.2.1 Scan continuo

El módulo ha sido configurado para tener un *scan interval* de 500 ms y un *window interval* de 500 ms. Esto implica que, en teoría, estaremos escaneando el 100% del tiempo. La función de escaneo estará a la escucha durante un tiempo determinado. En este caso el tiempo de escucha es de 500 ms.

La siguiente gráfica (**Fig. 6.11**) muestra el consumo de Wasmote con esta configuración.

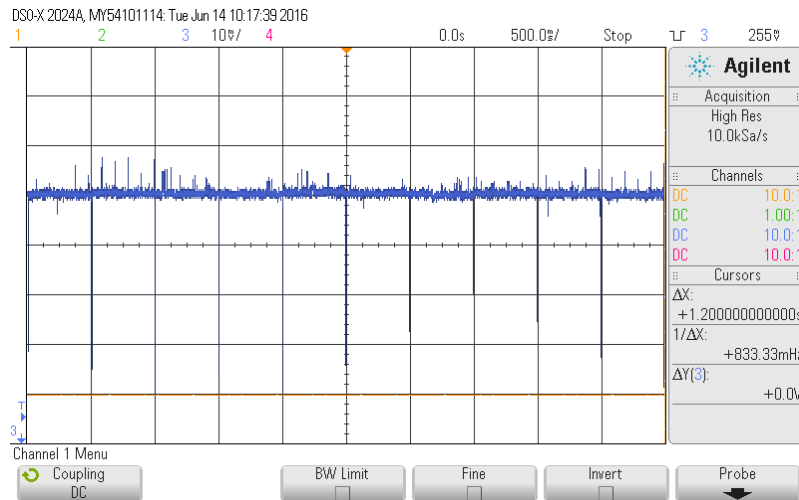


Fig. 6.11 Consumo de la Wasmote en modo “scan” continuo con tiempo de escucha de 500 ms

Como se puede apreciar, el consumo se mantiene, debido al “scan” continuo, a excepción de bajadas cada 500 ms, debido al tiempo de escucha de la función “scan”. El consumo máximo es de 62mA y el mínimo de 42mA aproximadamente.

6.2.2 Scan interval 500 ms, Scan window 250 ms

El módulo ha sido configurado para tener un *scan interval* de 500 ms y un *window interval* de 250 ms. Esto implica que estaremos escaneando el 50% del tiempo. Para ser lo más eficientemente posible, debemos consumir lo mínimo durante los tiempos de no escaneo. Se debe, por tanto implementar *BLE.sleep()* para el módulo BLE y el modo “sleep” para Wasmote.

La siguiente gráfica (**Fig. 6.12**) muestra el consumo de Wasmote con ciclo de cambio de modo cada 2 s.

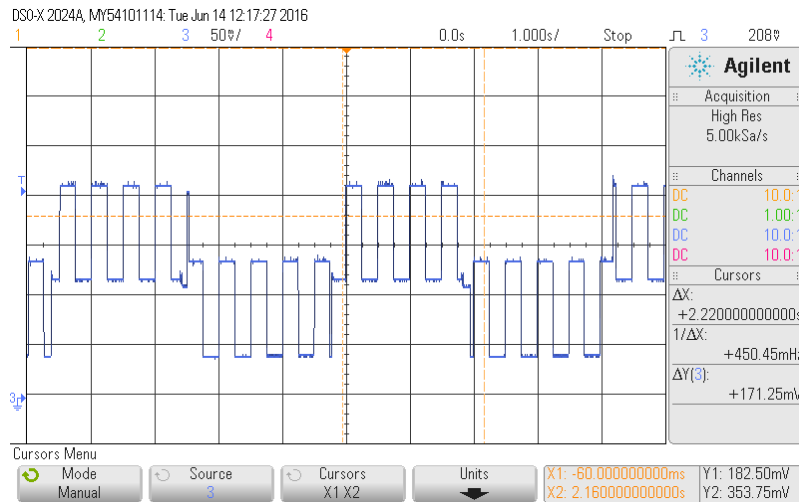


Fig. 6.12 Consumo de la Waspote con scan interval 500 ms y scan window 250 ms, cambio de ciclo cada 2 s sin *BLE.sleep()*

Como se puede apreciar, el consumo disminuye cada ciclo “sleep”. Durante estos dos segundos podemos apreciar 4 tiempos de intervalo, de los cuales, la mitad del tiempo se está escaneando.

Con la implementación de “sleep” para el módulo BLE, durante los ciclos de “sleep” de Waspote y en el tiempo de no escaneo, el consumo debería ser lo más cercano posible a 0mA.

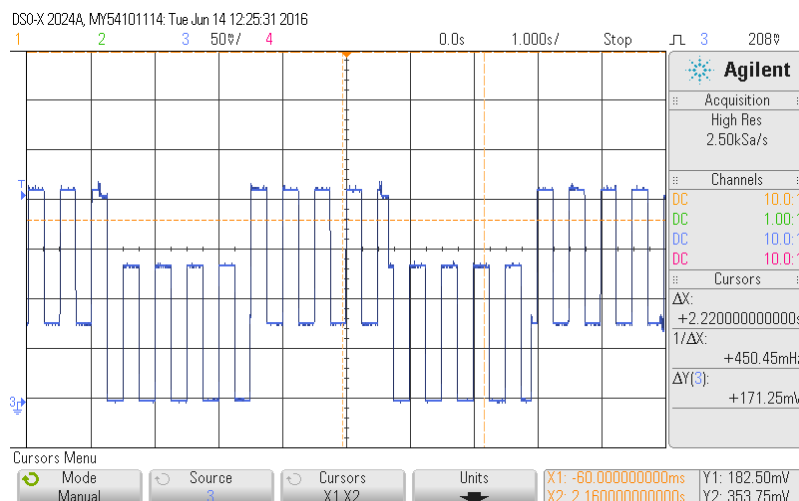


Fig. 6.13 Consumo de la Waspote con scan interval 500 ms y scan window 250 ms, cambio de ciclo cada 2 s con *BLE.sleep()*

Como se puede apreciar en la gráfica con *BLE.sleep()* implementado, en los tiempos de no escaneo, durante los ciclos “sleep” de Wasmote, el consumo es nulo.

6.3 Comparación “log” Wasmote vs “log” Wireshark

Se pretende saber con qué exactitud Wasmote recibe paquetes de “advertising”. Conforme el procesador de Wasmote recibe eventos de escaneo por parte del módulo BLE, o mejor dicho del chipset de Bluegiga, se ha desarrollado una modificación de la API para que se guarde este evento en un “log” generado en la SD. Este “log” contiene toda la información, la analiza y le añade un timestamp. El código se encuentra en el Anexo 3.

Del mismo modo Wireshark genera un “log” de la interfaz seleccionada, en este caso la interfaz Bluetooth, que está a la escucha del mismo dispositivo emisor que Wasmote.

El dispositivo emite paquetes cada 100 ms, o lo que es lo mismo 10 veces por segundo. Mientras que Wireshark recibe 7 u 8 paquetes de media por segundo, Wasmote genera un “log” con una media de 3 o 4 paquetes por segundo.

Esto viene dado por el hecho de que estamos ocupando el procesador con la generación del “log”, mientras que a la vez escucha los eventos que le manda el módulo BLE, y esto supone una pérdida de un porcentaje de los paquetes.

La modificación de la API ha consistido en tratar el evento una vez ha sido recibido por Wasmote. Inmediatamente se analiza el evento y se construye la información que será enviada a la tarjeta de memoria, donde se grabará el archivo de “log”. Mientras tanto el módulo BLE sigue recibiendo mensajes que son transmitidos a Wasmote para su manipulación. Esta circunstancia justifica la pérdida de paquetes recibidos por Wasmote.

CAPÍTULO 7. IMPLEMENTACIÓN RETRANSMISIÓN NODO A NODO

Este capítulo explica la implementación en Waspote de la propuesta realizada en [12]. Se plantea una situación de avería, como podría ser un accidente, obras o una grúa de auxilio. El vehículo averiado o la grúa transmiten los paquetes de “advertising” con la información de aviso de cada situación. El “beacon” de la señal más cercana debe tener la opción de transmitir información, incluso seleccionar qué tipo de aviso quiere transmitir, avería, obras, accidente..., en los paquetes de “advertising”. Estos avisos se pueden difundir mediante el método “flooding” (inundación en inglés), retransmitiendo la información de señal a señal de tráfico para avisar de posibles imprevistos y reducir la velocidad. En casos de avería y accidentes, esta información podría llegar hasta un centro de auxilio para el rescate.

Para poder llevar a cabo el “flooding”, los dispositivos “beacon” de las señales deben estar programados en modo escaneo y emisión simultáneamente. El dispositivo debe estar escuchando, a la espera, por ejemplo, de un paquete diferencial de aviso de peligro, y cambiar su configuración para seguir emitiendo a sus dispositivos cercanos esa información prioritaria.

En la gráfica (**Fig. 7.1**) se muestra el comportamiento de Waspote configurada en ambos modos simultáneamente. Para el modo de escaneo la configuración ha sido la siguiente: *scan interval* de 500 ms y *windows interval* de 250 ms. Por otro lado, el modo de emisión está configurado de la siguiente manera: potencia de transmisión -3 dBm y 100 ms de intervalo de emisión. El cambio de ciclo de “sleep” se ha configurado cada 2 s, de esta manera podemos comprobar la variación del consumo.

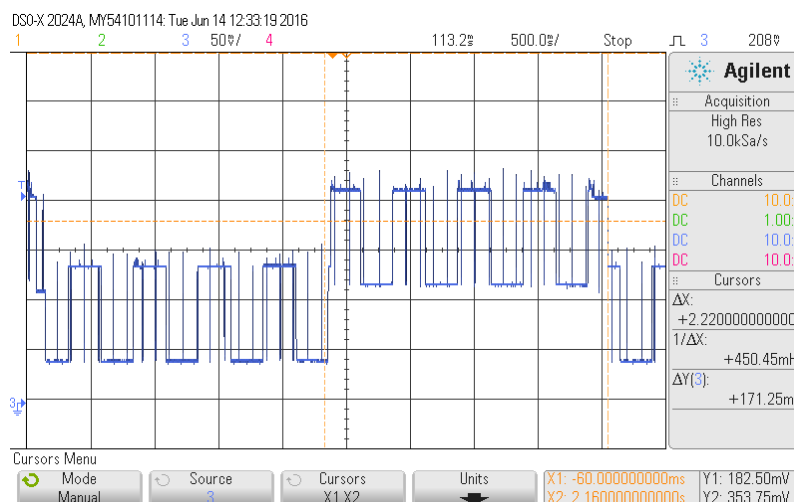


Fig. 7.1 Recepción y emisión simultánea

Se puede observar como la emisión y recepción simultánea es posible, pero un “beacon” no puede emitir y escuchar al mismo tiempo, de manera que durante la emisión de “advertisements”, éste deja de escanear momentáneamente.

Este proceso se puede apreciar en la siguiente gráfica.

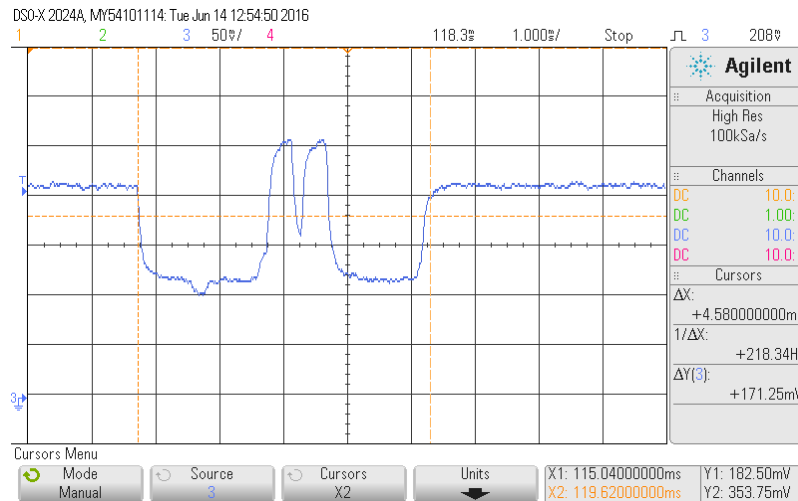


Fig. 7.2 Emisión de un advertisement por dos canales, con recepción simultánea

La implementación, se encuentra en el Anexo 3. Waspote emite información siguiendo la estructura de señalización explicada en capítulos anteriores y está a la escucha de un posible mensaje con tag “Comple Name” “SOS!”. En caso de recibir cualquier paquete con dicho tag, automáticamente cambia su “Complete Name” por “SOS!”, de manera que así el mensaje se pueda expandir.

CAPÍTULO 8. CONCLUSIONES E IMPACTO SOCIAL

El estudio realizado evalúa las características técnicas de Waspnote y del módulo BLE fabricado por Libelium, en especial en lo relacionado a la gestión de la energía en entornos de emisión de avisos, escaneo de dispositivos y emisión de avisos combinados con la retransmisión nodo a nodo tras recibir un mensaje.

El entorno de emisión de avisos, simula un escenario en el que una señal viaria emite un mensaje, por ejemplo, con un límite de velocidad establecido, de manera que pueda ser detectado por un vehículo en movimiento con un dispositivo Bluetooth. En este escenario las pruebas realizadas muestran que Waspnote y el módulo BLE emiten avisos con un nivel de consumo bajo dentro de lo esperado.

En un entorno combinado donde se emite un aviso y a su vez se está a la escucha de un paquete con una señal de emergencia que se retransmitirá a otros dispositivos, los niveles de consumo se mantienen en la línea del resto de pruebas.

Waspnote no se diferencia significativamente de las características de otros módulos Bluetooth que por un precio más asequible podrían responder de manera adecuada a los escenarios propuestos en este trabajo. Su cualidad más importante, es la posibilidad de montar una red de sensores de forma fácil, gracias a los múltiples módulos vendidos por Libelium y a su API, pero, en este caso, en el que solo se va a configurar como emisor Bluetooth, sus cualidades de bajo consumo no justifican su precio.

A lo largo del proyecto se han observado las características del consumo de energía que tienen Waspnote y el módulo BLE y si bien cumplen con lo esperado, se ha observado que podría mejorarse. Sería muy útil disponer de una interrupción asíncrona de Waspnote generada por el módulo BLE como respuesta a un evento detectado. Esto nos permitiría activar o desactivar el modo de bajo consumo de Waspnote en función de los eventos recibidos. Para ello se debe modificar la API y desarrollar la interrupción generada por el chipset de Bluegiga.

El impacto medioambiental de Waspnote y del módulo BLE es poco significativo. En los escenarios previstos, Waspnote se puede alimentar con una placa solar que suministre la energía necesaria sin necesidad de ninguna batería adicional o conexión a la red eléctrica. Por la naturaleza del sistema, la contaminación medioambiental es nula.

El proyecto nos permite ser optimistas ante una futura mejora del sistema viario, ya que, la identificación de la señalización viaria mejoraría sustancialmente al no verse afectada por los diversos problemas de reconocimiento de imagen. El coste de un receptor Bluetooth en un vehículo es

muy inferior al de un sistema de reconocimiento de imágenes como el que se utiliza actualmente. Al tener un coste inferior es posible que un gran número de vehículos puedan utilizar esta tecnología y no sólo los vehículos de alta gama, como sucede actualmente.

Al establecer una comunicación entre los elementos de las vías de circulación y los vehículos, se amplía enormemente la información del entorno, mejorando así la circulación y acercándonos un poco más a un modelo de ciudad inteligente.

La mayor dificultad para el despliegue de una solución de este tipo, consiste en la inversión inicial para dotar a la señalización viaria de los dispositivos “beacon” necesarios para emitir los avisos. Sin embargo, esta inversión inicial se ve justificada por una mejora sustancial del sistema y un coste de mantenimiento poco elevado.

BIBLIOGRAFÍA

- [1] Apple. "iBeacon for Developers". <https://developer.apple.com/ibeacon>. Apple Inc, 4 Sep. 2015. Web. 23 Feb. 2016.
- [2] Google. "Google "beacons"". <https://developers.google.com/beacons>. Google, 17 Jun. 2016. Web. 23 Feb. 2016.
- [3] Radius Networks. "AltBeacon". <http://altbeacon.org>. Radius Networks, 11 Mar. 2015. Web. 23 Feb. 2016.
- [4] David Perez-Diaz de Cerio, José Luis Valenzuela. "Provisioning Vehicular Services and Communications Based on a Bluetooth Sensor Network Deployment". <http://www.mdpi.com/1424-8220/15/6/12765>. UPC, Castelldefels (2015).
- [5] Xataka. "¿Cómo saber cuándo una embarazada necesita un asiento en el transporte público? Con "beacons"". <http://www.xataka.com/internet-of-things/como-saber-cuando-una-embarazada-necesita-un-asiento-en-el-transporte-publico-con-beacons>. Radius Networks, 6 Jun. 2016. Web. 8 Feb. 2016.
- [6] Mikko Savolainen. "[REFERENCE]: BLE112 TX power settings". <https://bluegiga.zendesk.com/entries/22326643--REFERENCE-BLE112-TX-power-settings>. Bluegiga, 5 Nov. 2012. Web. 5 Mar. 2016.
- [7] Argenox. http://www.argenox.com/wp-content/uploads/ble_advertisement_packet_format_payload.png
- [8] SIG. "Generic Access Profile". <https://www.bluetooth.org/en-us/specification/assigned-numbers/generic-access-profile>. Bluetooth SIG, Web. 5 Mar. 2016.
- [9] Akhayad, Yassir, "IMPLEMENTACIÓN DE PROCEDIMIENTOS DE CONFIGURACIÓN", Cap. 3 en *Bluetooth 4.0 Low Energy: Análisis de las prestaciones y aplicaciones para la automoción*, 15-18, UPC, Castelldefels (2016).
- [10] Wireshark. "Wireshark, Go Deep". <https://www.wireshark.org>. Wireshark Foundation, Web. 9 Dic. 2015.
- [11] BlueZ. "BlueZ Official Linux Bluetooth protocol stack". <https://www.bluez.org>. Bluez Project, Web. 9 Dic. 2015.
- [12] Akhayad, Yassir, "PROPUESTAS PARA EL RECONOCIMIENTO DE SEÑALES DE TRÁFICO ", Cap. 8 en *Bluetooth 4.0 Low Energy: Análisis de las prestaciones y aplicaciones para la automoción*, 48-49, UPC, Castelldefels (2016).
- [13] Group, Bluetooth Special Interest. "Advertising Interval", Cap 4.4.2.2 en *Bluetooth Core Specification v4.0*, 59. SIG, https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737
- [14] Libelium. "Sleep mode", Cap. 4.3 en *Bluetooth Low Energy Networking Guide*, http://www.libelium.com/downloads/documentation/bluetooth-low-energy-networking_guide.pdf. Libelium, 30 Jun. 2010. PDF. 3 Jun. 2016.
- [15] Instrument, Texas. 0-1A, Single-Supply, Low-Side, Current Sensing Solution.

- [16] Libelium. "Waspote Technical Guide". http://www.libelium.com/downloads/documentation/waspote_technical_guide.pdf. Libelium, Jun. 2016. PDF. 27 Jun. 2016.
- [17] Libelium. "Waspote Interruptions Programming Guide". http://www.libelium.com/downloads/documentation/waspote-interruptions-programming_guide.pdf. Libelium, Sep. 2013. PDF. 27 Jun. 2016.



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ANEXOS

TÍTULO DEL TFG: Estudio de la tecnología de iBeacons utilizando una Waspote

TITULACIÓN: Grado en Ingeniería de Sistemas de Telecomunicación

AUTOR: Eric Murillo López

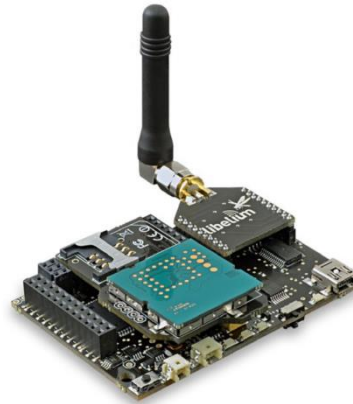
DIRECTOR: David Pérez Díaz de Cerio

FECHA: 5 de julio del 2016

Anexo 1. Descripción detallada de Waspote

A1.1 Datos generales

Microcontrolador: ATmega1281
 Frecuencia: 14,7456 MHz
 SRAM: 8KB
 EEPROM: 4KB
 FLASH: 128KB
 Tarjeta SD: 2GB
 Peso: 20g
 Dimensiones: 73,5 x 51 x 13 mm
 Rango de temperaturas: [-10°C, +65°C]
 Reloj: RTC (32kHz)

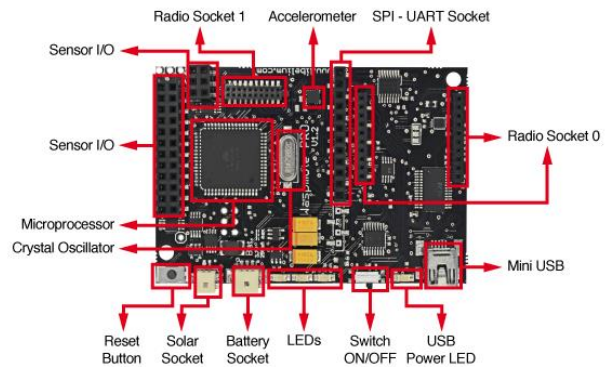


Consumos:

ON: 15mA
 Sleep: 55µA
 Deep Sleep: 55µA
 Hibernate: 0,07µA

Inputs/Outputs:

7 Analog (I), 8 Digital (I/O), 1 PWM,
 2 UART, 1 I²C, 1USB, 1SPI

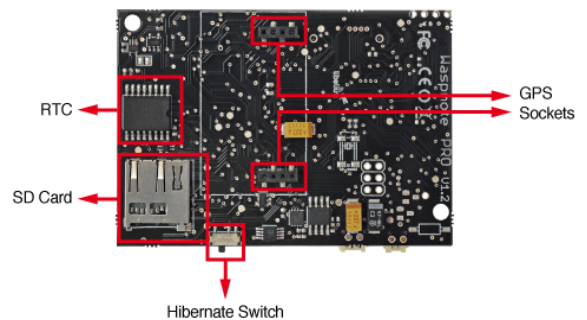


Datos eléctricos:

Voltaje de la batería: 3,3V - 4,2V
 Carga USB: 5V – 100mA
 Carga Panel Solar: 6 - 12V - 280mA

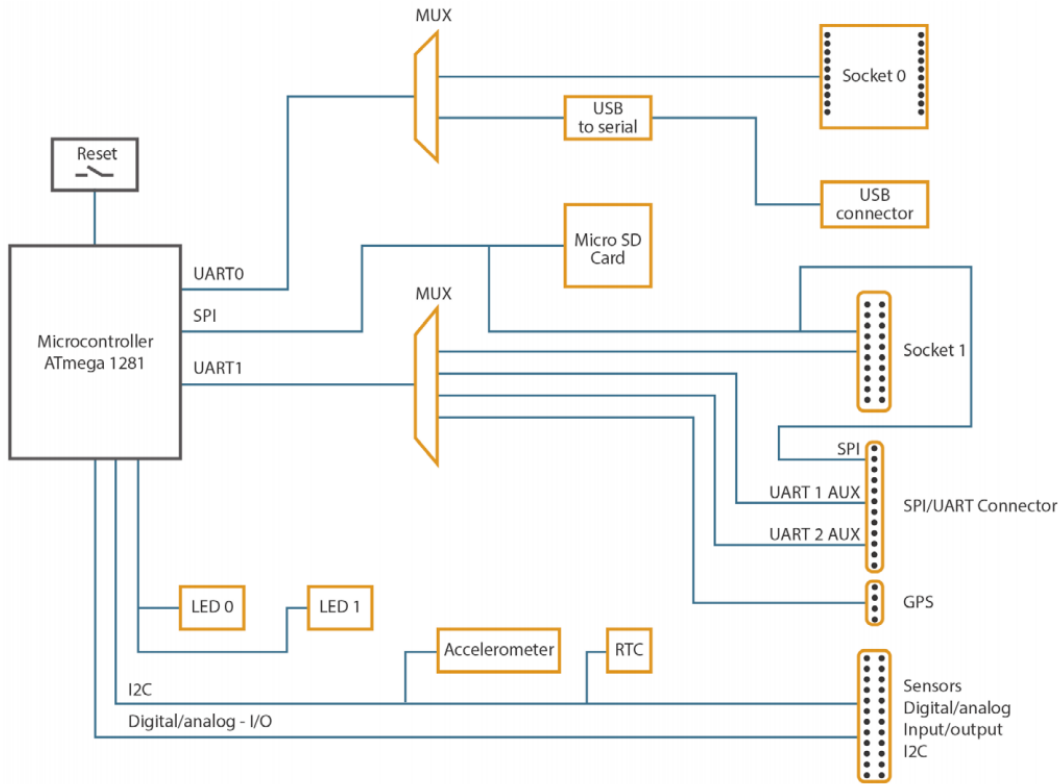
Sensores incorporados a la plataforma:

Temperatura (+/-): -40°C, +85°C.
 Resolución: 0,25°C
 Acelerómetro: ±2g/±4g/±8g
 Baja potencia: 0,5 Hz/1 Hz/2 Hz/5 Hz/10 Hz
 Modo normal: 50 Hz/100 Hz/400 Hz/1000 Hz

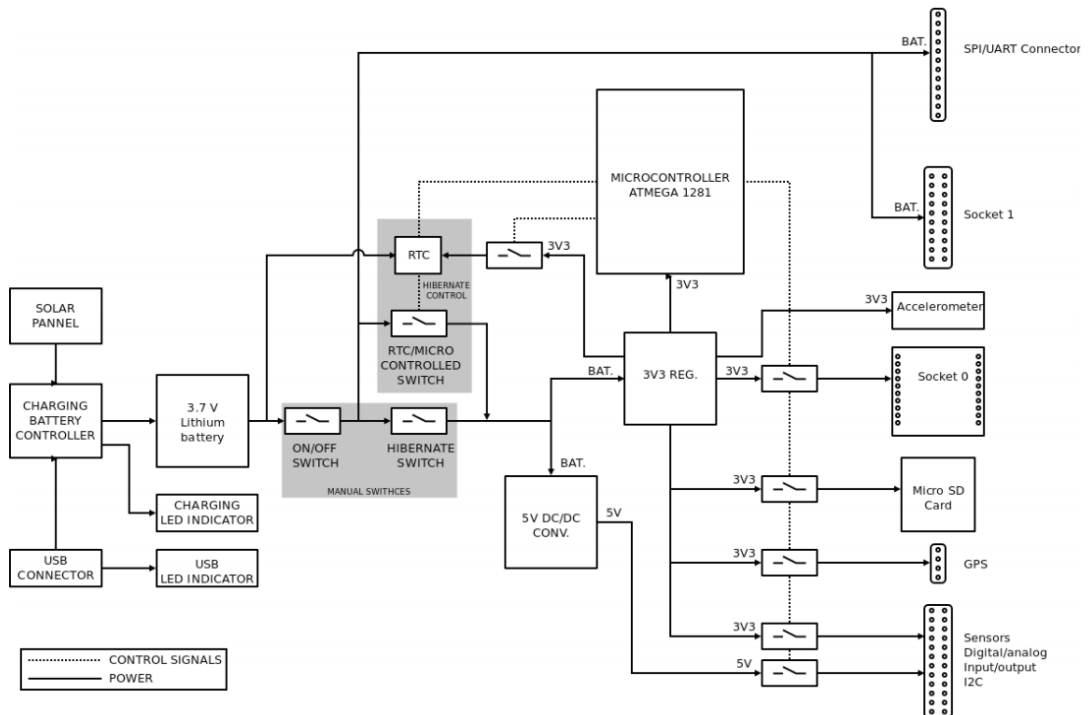


A1.2 Diagrama de bloques de Wasmote

Señales de datos:

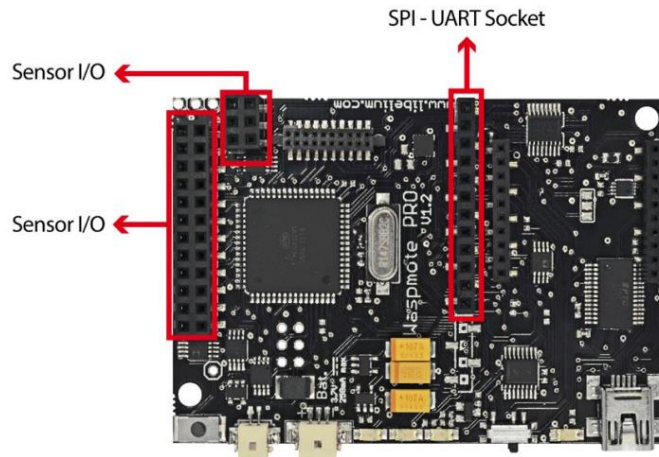


Señales de potencia:

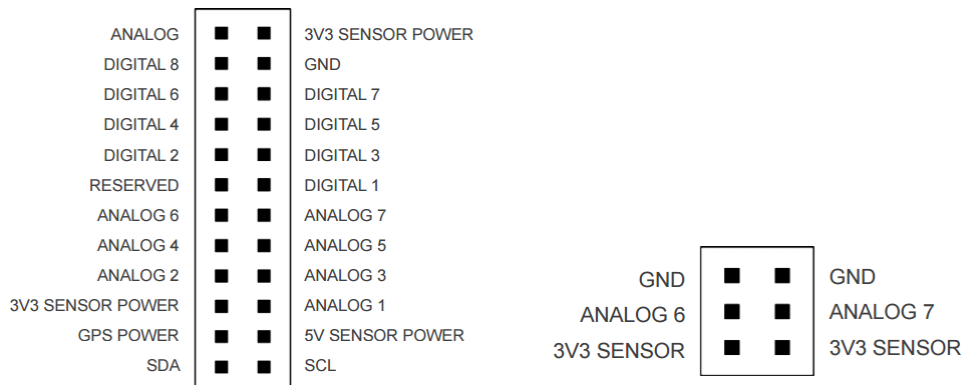


A1.3 Input/Output

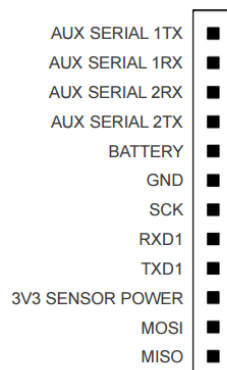
Waspnote puede comunicarse con otros dispositivos externos a través de los diferentes puertos de entrada/salida (I/O):



Conectores para sensores:



SPI-UART conector:



A1.4 Entradas analógicas

Waspnote tiene 7 entradas analógicas accesibles para la conexión de sensores. Cada entrada se encuentra directamente conectada al microcontrolador, que utiliza un convertidor analógico/digital (ADC) de aproximación sucesiva de 10 bits. La tensión de referencia en las entradas es de 0V (GND), y el valor máximo de tensión de entrada es 3,3V, que se corresponde con la tensión general de alimentación del microcontrolador.

Para la obtención de valores de entrada, se usa la función `analogRead(analog input)`, el parámetro de entrada de la función será el nombre de la entrada que será leído "ANALOG1, ANALOG2...". El valor obtenido de esta función será un número entero entre 0 y 1023, donde 0 corresponde a 0V y 1023 a 3,3V.

Los pines de entrada analógicas también se pueden utilizar como pines de entrada/salida digital. En este caso, se debe tener en cuenta la siguiente lista de correspondencia para los nombres de los pines:

Pin analógico		Pin digital
ANALOG1	=>	14
ANALOG2	=>	15
ANALOG3	=>	16
ANALOG4	=>	17
ANALOG5	=>	18
ANALOG6	=>	19
ANALOG7	=>	20

A1.5 Entradas digitales

Waspnote tiene pines digitales que se pueden configurar como entrada o salida en función de las necesidades de la aplicación. Los valores de tensión correspondientes a los diferentes valores digitales serían:

- 0V for logic 0
- 3,3V for logic 1

Las instrucciones para el control de los pines digitales son:

```
{  
  //set DIGITAL3 pin as input and read its value  
  
  pinMode(DIGITAL3, INPUT);  
  valor = digitalRead(DIGITAL3);  
  
  //set DIGITAL3 pin as output and set it LOW  
  
  pinMode(DIGITAL3, OUTPUT);  
  digitalWrite(DIGITAL3, LOW);  
}
```

A1.6 PWM

El pin DIGITAL1 también se puede utilizar como salida PWM (modulación por ancho de pulso) con la que una señal analógica puede ser "simulada". En realidad, se trata de una onda cuadrada entre 0V y 3,3V para la que la proporción de tiempo cuando la señal es alta, puede cambiar de 0% a 100%, simulando una tensión de 0V (0%) a 3,3V (100%). Posee una resolución de 8 bits, por lo que se pueden configurar hasta 255 valores entre 0-100%.

La instrucción para controlar la salida del PWM es `analogWrite(DIGITAL1, valor)`; donde el parámetro "valor" es el valor analógico de 0 a 255.

```
{  
  analogWrite(DIGITAL1, 127);  
}
```

A1.7 UART

UART, son las siglas en inglés de Universal Asynchronous Receiver-Transmitter, en español: Transmisor-Receptor Asíncrono Universal. Es el dispositivo que controla los puertos y dispositivos serie y se encuentra integrado en la placa base o en la tarjeta adaptadora del dispositivo.

Las funciones principales del chip UART son: manejar las interrupciones de los dispositivos conectados al puerto serie y convertir los datos en formato paralelo, transmitidos al bus de sistema, a datos en formato serie, para que puedan ser transmitidos a través de los puertos y viceversa.

El controlador del UART es el componente clave del subsistema de comunicaciones series de una placa. El UART toma bytes de datos y transmite los bits individuales de forma secuencial. En el destino, un segundo UART reensambla los bits en bytes completos. La transmisión serie de la información digital (bits) a través de un cable único u otros medios es mucho más efectiva en cuanto a coste que la transmisión en paralelo a través de múltiples cables. Se utiliza un UART para convertir la información transmitida entre su forma secuencial y paralela en cada terminal de enlace. Cada UART contiene un registro de desplazamiento que es el método fundamental de conversión entre la forma secuencial y paralela.

En Waspmote tenemos dos UARTs: UART0 y UART1. Además, hay varios puertos que pueden ser conectados a los UARTs a través de dos multiplexores (Texas Instruments SN74CB3Q3253) diferentes, uno para cada uno.

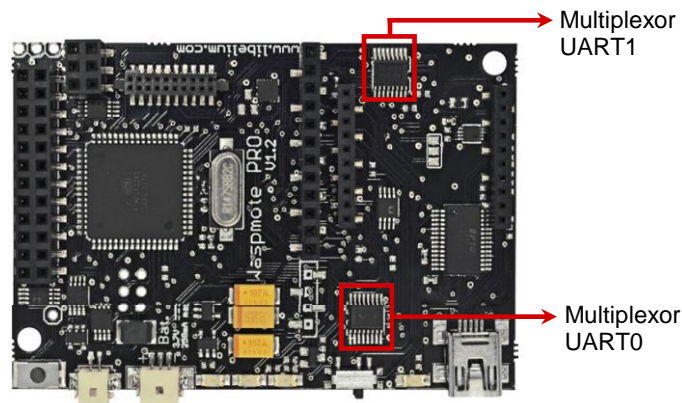
- UART0 está compartido por el puerto USB y el Socket0. Esta toma se utiliza para los módulos XBee, módulos RFID, módulos Bluetooth, módulos Wi-fi, etc. El multiplexor en este UART controla la señal de datos que por defecto siempre es emitida por el Socket0. Cuando el USB necesita enviar información a través de la UART0, el multiplexor se

conecta momentáneamente al puerto USB y retrocede de nuevo a Socket0 después de la impresión.

- UART1 está compartida por cuatro puertos: Socket1, Socket GPS, y Auxiliar1 y Auxiliar2. Es posible seleccionar en el mismo programa cuál de los cuatro puertos se conecta a UART1 en el microcontrolador.

La configuración del multiplexor UART1 se lleva a cabo mediante las siguientes instrucciones:

```
{
  Utils.setMuxAux1(); // Establece la toma Auxiliar1
  Utils.setMuxAux2(); // Establece la toma Auxiliar2
  Utils.setMuxGPS(); // Establece la toma GPS
  Utils.setMuxSocket1(); // Establece la Socket1
}
```



A1.8 I²C

I²C, en inglés Inter-Integrated Circuit, es un bus de datos serial desarrollado en 1982 por Philips Semiconductors (hoy NXP Semiconductors).

Se utiliza, principalmente, para la comunicación interna entre diferentes partes de un circuito, por ejemplo, entre un controlador y circuitos periféricos integrados.

El bus de comunicación I²C es utilizado por dos dispositivos, el acelerómetro y el RTC, conectados en paralelo. En todos los casos, el microcontrolador actúa como master (maestro), mientras que los otros dispositivos conectados al bus son slaves (esclavos).

A1.9 SPI

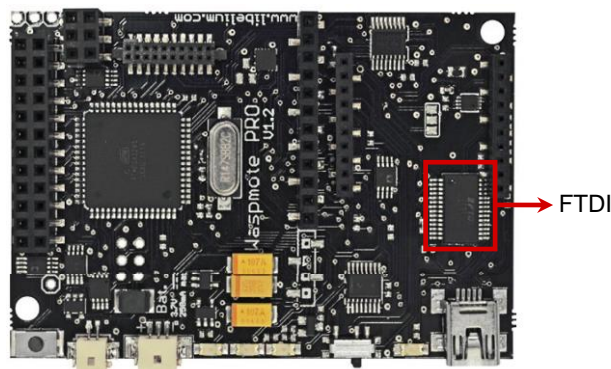
El Bus SPI (del inglés Serial Peripheral Interface) es un estándar de comunicaciones, usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos. Es un estándar para controlar casi cualquier dispositivo electrónico digital que acepte un flujo de bits serie regulado por un reloj (comunicación sincrónica).

El puerto SPI se usa en el microcontrolador para la comunicación con la tarjeta micro-SD. Todas las operaciones que utilizan el bus son realizadas por la biblioteca específica. El puerto SPI está disponible también en el conector SPI/UART.

A1.10 USB



Este puerto se utiliza para la comunicación con un ordenador o dispositivos USB compatibles. Esta comunicación permite la carga del programa en el microcontrolador. Para la comunicación USB, se utiliza la toma UART0. El chip FTDI FT232RL lleva a cabo la conversión al estándar USB.



A1.11 Microcontrolador ATmega1281

El ATmega1281 es un microcontrolador CMOS de baja potencia de 8 bits, de la familia AVR del fabricante Atmel, basado en la arquitectura RISC. Mediante la ejecución de potentes instrucciones en un solo ciclo de reloj, el ATmega1281 logra rendimientos que se acercan a 1 MIPS² a 1MHz por MHz, lo que permite optimizar el consumo de energía en comparación con la velocidad de procesamiento. A continuación, se pueden ver las características genéricas del ATmega1281:

² MIPS: Millones de Instrucciones Por Segundo.

- FLASH: 128KB
- EEPROM: 4KB
- RAM: 8KB
- Pines E/S de propósito general: 54
- Canales PWM de resolución 16 bits: 6
- Puertos USARTs serie: 2
- Canales ADC: 8

Es bastante conocido debido a su diseño simple y a su facilidad de programación. Para maximizar las prestaciones, el AVR está dotado de arquitectura Harvard (con buses y memorias separadas para programas y datos). Mientras se está ejecutando una instrucción, se realiza la fase de búsqueda en memoria de la siguiente. Este concepto permite que se ejecute una instrucción en cada ciclo de reloj.

A1.12 Tarjeta SD

Es una tarjeta micro-SD (Secure Digital) que se utiliza específicamente para reducir espacio en la placa, al mínimo. Waspote utiliza el sistema de archivos FAT16 y puede soportar tarjetas de hasta 2 GB.

Para comunicarse con el módulo SD se usa el bus SPI, que incluye líneas para el reloj, los datos de entrada y salida de datos, y un pin de selección. La tarjeta SD se alimenta a través de un pin digital del microcontrolador, y por tanto no es necesario el uso de un interruptor para cortar la alimentación, poniendo un pin a valor bajo es suficiente para establecer el consumo de la SD a 0µA.

A1.13 Real Time Clock – RTC

Waspote lleva un RTC (Maxim DS3231SN) incorporado que permite programar la placa para ejecutar acciones relacionadas con el tiempo.

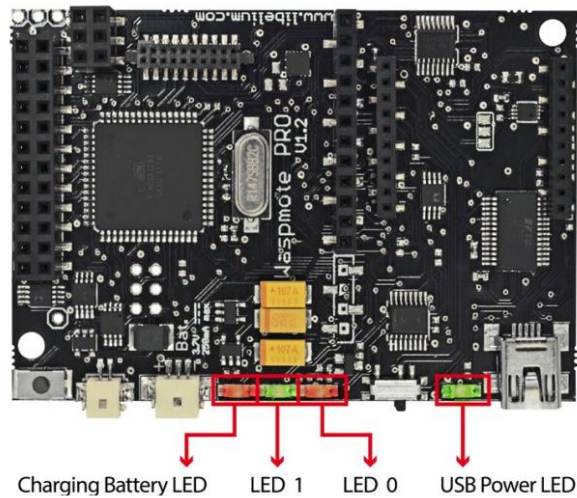


El sistema de alarmas del RTC pueden ser programadas especificando día/hora/minuto/segundo. Eso permite un control total sobre cuándo la Waspote despierta para capturar valores de los sensores y realizar acciones programadas en él. Esto permite estar en modo ahorro de energía (modos Deep Sleep e Hibernate) y hacer que despierte justo en el momento requerido.

El RTC se alimenta de la batería "principal", y por tanto no requiere de una batería de botón adicional. Debemos de tener en cuenta que si se extrae la batería, no se mantienen los datos de tiempo. Es por ello que se aconseja utilizar el tiempo en términos relativos y no absolutos.

Todo el control y programación del RTC se hace mediante el bus I²C.

A1.14 LEDs



Hay 4 indicadores LED en la placa:

- Indicador LED de carga de la batería: Un LED rojo indica que hay una batería conectada en Wasmote que está siendo cargada. Una vez que la batería está completamente cargada, el LED se apagará automáticamente.
- Indicador LED USB: Cuando el LED está encendido indica que el cable USB está conectado correctamente al Wasmote. Al retirar el cable USB el LED se apagará automáticamente.
- LED 0: Se identifica mediante un LED verde. Es totalmente programable por el usuario mediante código de programación. Además, indica el reseteo del dispositivo mediante un parpadeo.
- LED 1: Se identifica mediante un LED rojo. De forma análoga al anterior es totalmente programable por el usuario desde el código de programa.

Anexo 2. Descripción detallada de Wasmote IDE

El compilador y las librerías de la API están empaquetadas en un archivo comprimido, descargable desde la web, existiendo versiones para sistemas operativos Linux de 32 y 64 bits, Mac OS y Windows. El entorno de desarrollo se utiliza para escribir código que se descarga en Wasmote. También permite monitorizar la salida por el puerto serie de Wasmote y depurar el código, si es necesario.

En el proceso de instalación del entorno se crean tres carpetas en el sistema operativo: “examples”, “libraries” y “wasmote-api”. Para reducir el consumo de memoria RAM, un programa utiliza las únicas librerías “core” instaladas en la carpeta “wasmote-api” y sólo las librerías opcionales de los módulos instalados en Wasmote y que se encuentran instaladas en la carpeta “libraries”. La carpeta “examples” como su nombre indica contiene una gran variedad de ejemplos de código facilitado por Libelium.

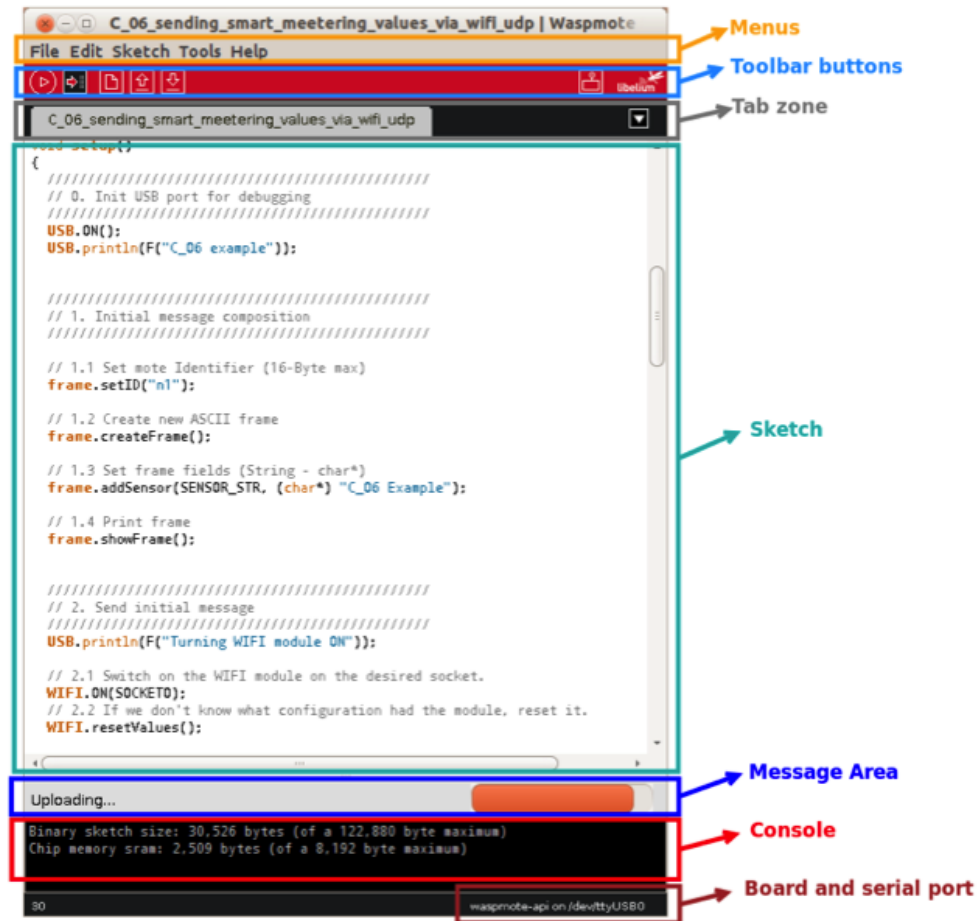
A través de la conexión mini USB se conecta Wasmote al ordenador. Wasmote incorpora un procesador de la empresa FTDI que permite utilizar la conexión USB. La primera vez que se conecta Wasmote con un ordenador mediante la conexión USB se debe instalar el driver de FTDI. Esto permite que las aplicaciones puedan conectarse con Wasmote utilizando el protocolo RS232 de puerto serie, en modo emulación, a través del conector mini USB.

A2.1 Descripción del entorno de desarrollo

El entorno de desarrollo se compone de una pantalla principal con todas las funciones disponibles a través de un menú y una barra de herramientas. Cada archivo de programa, “Sketch” en terminología de Wasmote, se muestra en una pestaña. En la parte inferior de la pantalla principal del IDE se muestran mensajes, la consola del sistema y la visualización del dispositivo emulador del puerto serie que se está utilizando.





Todos los programas para Wasmote tiene extensión “.pde”, también pueden contener código los archivos de lenguaje C (extensión “.c”), los archivos C++ (extensión “.cpp”) o archivos de cabecera del lenguaje C (extensión “.h”).


En la siguiente figura se muestra la pantalla principal del entorno de desarrollo:




La barra de herramientas permite el uso de las acciones más habituales:

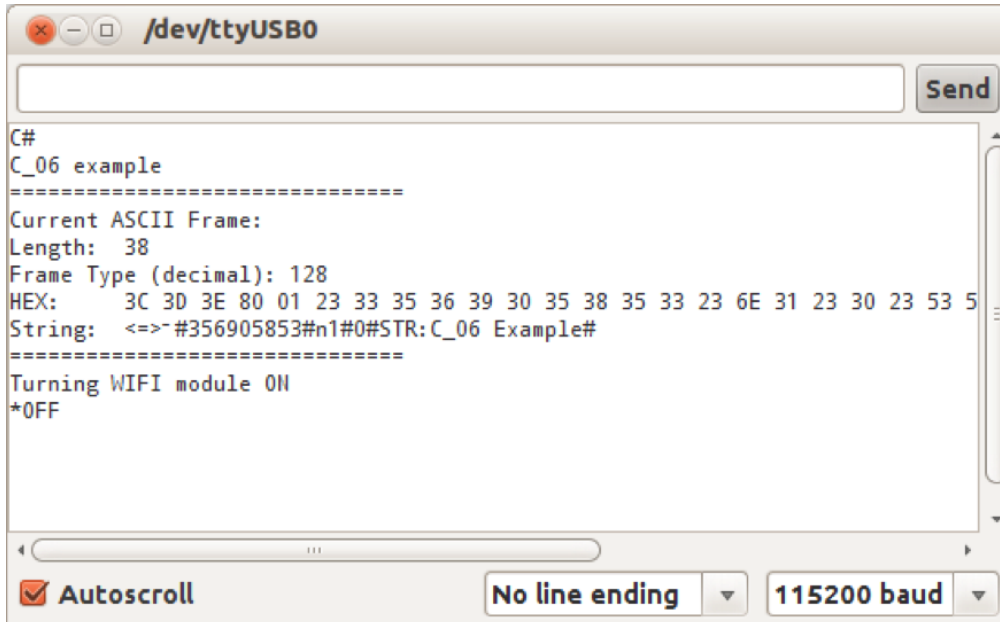


-  Abre un archivo de código ya existente.
-  Guarda el código escrito en una pestaña.
-  Crea un nuevo archivo de código
-  Verifica un archivo de código para detectar errores.

-  Compila el código, reinicializa la Waspnote y le carga el código a través del puerto serie definido en la opción de menú Tools→Target and Tools→Serial Port. En Windows será del tipo COMx, en Mac será del tipo /dev/tty.usbserialxxx y en Linux del tipo /dev/ttyUSBx.

 Abre la ventana de monitorización del puerto serie donde se muestran los datos que envía la Waspote a través de la emulación del puerto serie vía el conector mini USB.

En la siguiente figura se muestra la ventana de monitorización:



```
C#
C_06 example
=====
Current ASCII Frame:
Length: 38
Frame Type (decimal): 128
HEX: 3C 3D 3E 80 01 23 33 35 36 39 30 35 38 35 33 23 6E 31 23 30 23 53 5
String: <=>#356905853#n1#0#STR:C_06 Example#
=====
Turning WIFI module ON
*0FF
```

Para enviar datos a la Waspote se utiliza el cuadro de texto de la parte superior de la ventana y el botón “Send”.

Los parámetros de comunicación a través de la emulación del puerto serie son: velocidad 115.200 baudios, 8 bits, 1 bit stop y sin paridad.

A2.2 Arquitectura del sistema

La arquitectura de Waspote se basa en el microcontrolador ATmega1281 de ATMEL. Esta unidad de procesamiento comienza a ejecutar el bootloader, que es responsable de cargar en la memoria los programas y las bibliotecas compiladas, previamente almacenadas en la memoria flash, por lo que el programa principal que se ha creado, puede empezar a ejecutarse. Cuando Waspote se conecta e inicia el bootloader, hay un tiempo de espera de 62,5 ms antes de comenzar la primera instrucción para iniciar la carga de nuevas versiones de programas compilados.

A2.3 Estructura de un programa Wasmote

Los archivos de código de programa Wasmote tiene la extensión “.pde” y se dividen en funciones básicas: *setup()* y *loop()* incluidas en el siguiente esqueleto de programa:

```
// 1. Inclusión de librerías
// 2. Definiciones
// 3. Declaración de variables globales

void setup()
{
  // 4. Inicialización de módulos a usar
}
void loop()
{
  // 5. Medir
  // 6. Enviar información
  // 7. Entrar en estado de bajo consumo
}
```

La función *setup()* se ejecuta una única vez cuando el código se inicializa o Wasmote se reinicia.

La función *loop()* se ejecuta como un bucle indefinido. Wasmote incorpora un sistema de detección de interrupciones que permite bifurcar la ejecución del código dentro del bucle indefinido.

Todos los programas “.pde” invocan automáticamente las librerías “core” de la carpeta “Wasmote-api” y, por lo tanto, no necesitan ser incluidas mediante la instrucción `#include “nom_lib.h”` en las líneas de código anteriores a la función *setup()*.

Cualquier otra librería dla API de Wasmote almacenada en la carpeta “libraries” debe ser incluida en el código de programa “.pde” utilizando la instrucción `#include “nom_lib.h”` de manera obligatoria antes de la función *setup()*. La opción de menú Sketch→Import Library incluye las instrucciones `#include` de manera automática en el código del programa “.pde”.

A2.4 Estilo de programación Wasmote y buenas prácticas

La programación de un archivo “.pde” sigue el estilo del lenguaje de programación C. A continuación, se destacan algunas cuestiones relativas a los elementos de programación de Wasmote.

El código Wasmote admite la declaración de variables siguiendo el estilo del lenguaje C. Los tipos de variable se hayan definidos en `<stdint.h>`. Se recomienda la declaración de variables globales antes de la función *setup()*.

Cada instrucción finaliza con punto y coma “;” al estilo del lenguaje C. El programa se comenta siguiendo la misma sintaxis que el lenguaje C utilizando

el prefijo “//” delante del comentario o bien encerrando el comentario entre “/*” al inicio y “*/” al final.

Recomendaciones de Libelium como estilo de programación:

- Organizar y dividir el código en acciones funcionales coherentes entre sí.
- Al nombrar las variables con un nombre compuesto, las palabras deben estar concatenadas y la segunda y siguientes palabras deben comenzar en mayúsculas y el resto siempre en minúsculas. Por ejemplo: “miHumedad” sería aconsejable respecto a “MiHumedad”.
- Las constantes deben estar en mayúsculas. Por ejemplo # define WAIT 1000 para definir una constante que almacene el valor 1000.
- Utilizar la indentación usando simplemente dos espacios.
- Separar los bloques de código claramente, dejando el inicio de llave y el final de llaves solas en una línea completa.

Para asegurar el uso eficiente de los recursos de Wasmote, en especial en lo relativo al consumo de energía o en la calidad de la comunicación, Libelium recomienda las siguientes prácticas de programación:

- Utilizar la ventana de monitorización del puerto serie para mostrar todos los mensajes de depuración (DEBUG) que permitan determinar la línea donde se produce un error
- Crear una variable global que cuente el número de veces que se ha ejecutado la función loop() .
- Crear una variable que cuente el número de “frames” que envía/recibe la Wasmote.
- La comparación entre las dos variables anteriores nos puede ayudar a determinar el grado de pérdida de paquetes de comunicación.
- Es conveniente utilizar la instrucción if() para ser más eficiente en la ejecución del programa.
- En algunas ocasiones conviene no enviar “frames” para ahorrar batería, pero es una buena práctica enviar un “frame” cada cierto número de pasos del bucle loop() para asegurar que Wasmote está “trabajando”.
- Una instrucción while() debe contener una condición adicional sobre el número de veces que se está ejecutando el bucle que garantice que Wasmote no espere de manera indefinida. Es habitual añadir una condición con el operador OR (|) que se cumpla cuando han sucedido un número determinado de bucles en la función loop(). Se trata de establecer un “timeout” en la ejecución de la instrucción while().
- Las instrucciones para reducir el consumo de energía de Wasmote deben ir al final de la función loop() .
- Utilizar la función millis() para determinar el tiempo de ejecución de las funciones que se programen.

A2.5 Descripción de las librerías API

La interfaz de programación de aplicaciones (API) ha sido desarrollada en C y C++.

Las librerías “core” contienen la API que siempre se compila e incluye, entre otros, los siguientes módulos:

- Configuración general: relación de clases, variables globales, constantes, nombres de los pins del microprocesador y otras utilidades.
- Almacenamiento SD: librerías para leer y escribir en una tarjeta SD.
- Acelerómetro: librerías para leer el acelerómetro y activar/desactivar interrupciones en él.
- Control de energía: librerías para activar/desactivar los estados de energía “sleep”, “deep sleep” o “hibernate”. Incluye funciones para determinar el valor de batería restante, cerrar el bus I²C y limpiar interrupciones.
- RTC: librerías para obtener el día y hora del reloj en tiempo real interno. Incluye las funciones para activar las alarmas y las interrupciones generadas por el módulo.
- USB: librerías para gestionar la comunicación con un ordenador a través del puerto mini USB.
- Interrupciones: Librería que incluye las funciones necesarias para activar interrupciones y su posterior tratamiento en las subrutinas de interrupción.
- Xbee: librerías para gestionar todos los módulos Xbee.
- GPS pro: librerías para gestionar GPRS y GPRS+GPS módulos que incluyen funciones para enviar comandos AT, peticiones http y transferencias FTP.

Las librerías opcionales están dedicadas a diferentes módulos que pueden usarse con Waspote y que deben declararse explícitamente para que el entorno de desarrollo pueda compilarlas de manera adecuada.

Entre otras librerías, se incluyen en la API las que gestionan un GPS o diferentes tipos de sensores. También hay librerías para gestionar redes 802.15.4/ZigBee, Wifi, RFID o el caso que nos ocupa, una librería BLE para gestionar un módulo de baja energía Bluetooth.

A2.6 Tipos de memoria de Waspote

A2.6.1 Memoria Flash

El microcontrolador de Waspote dispone de 128 KB de memoria Flash donde se almacena el programa de arranque (bootloader) y el programa descargado desde el entorno de programación. Libelium no permite reescribir el bootloader con Waspote IDE. Si es modificado por otros medios, la garantía es anulada.

A2.6.2 Memoria EEPROM

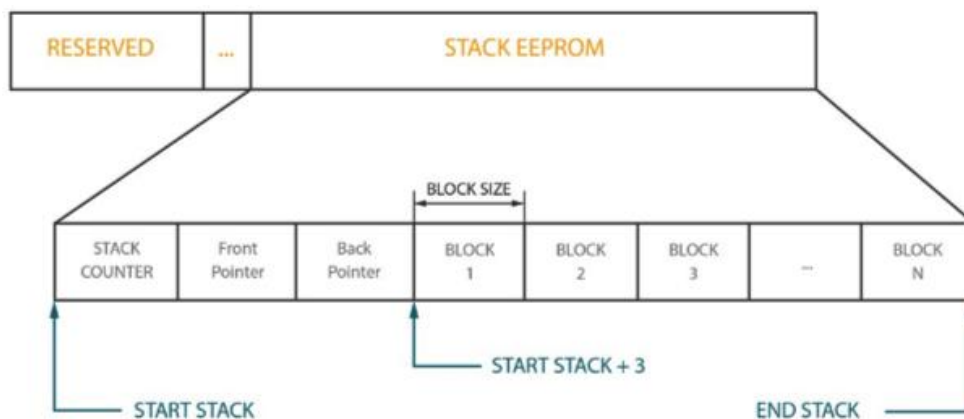
Para poder almacenar variables permanentemente cuando Waspote está apagada o bien en estado de hibernación, se utilizan 4 KB de memoria EEPROM no volátil. El primer KB está reservado para el sistema y se tienen disponible 3 KB desde la posición de memoria 1024 a 4095 para almacenar variables que permitan mantener las condiciones o las variables incluso si se hace un reinicio o una hibernación de Waspote.

La función que escribe en dicha memoria es *Utils.writeEEPROM(dirección de memoria, valor)* y para leer un dato se utiliza la instrucción *data = Utils.readEEPROM(dirección de memoria)*.

Waspote dispone de la clase *WaspStackEEPROM* para facilitar la grabación en la memoria EEPROM de “frames” en lugar de hacerlo en la memoria SD. Se admiten dos tipos de pilas: LIFO para leer el último dato grabado de una pila o FIFO para leer los datos de la pila en el mismo orden en el que se grabaron. Es muy útil para almacenar tramas de comunicación que no se han podido entregar y que se quieren guardar en memoria no volátil para intentar enviarlas en otro momento en el que sea posible.

La clase tiene un método para inicializar la pila entre las posiciones de memoria 1023 y 4095 y si es una pila LIFO o FIFO. La pila se divide en bloques de memoria que tienen como máximo 255 bytes. Para grabar un bloque de datos en la pila se utiliza la función *stack.push(data.buffer, data.length)*. Para leer un bloque de datos de la pila se utiliza la instrucción *data.length = stack.pop(data.buffer)*

Esquemáticamente la pila en la memoria EEPROM responde a este diagrama:

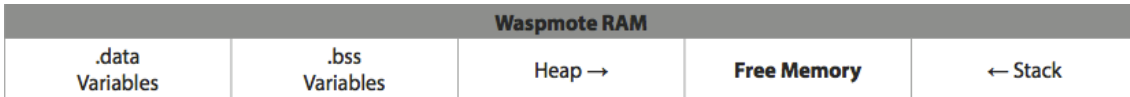


A2.6.3 Memoria RAM

También se dispone de 8 KB de memoria SRAM del microcontrolador de Waspote. Esta memoria volátil es compartida por variables inicializadas o no, por el gestor de memoria y por la pila que es usada para llamar a subrutinas y

almacenar variables locales. Los programas de Wasmote deben hacer un uso eficiente de dicha memoria, borrando las variables cuando han sido utilizadas o utilizando variables estáticas y no dinámicas, entre otras buenas prácticas.

La memoria RAM se distribuye según el siguiente diagrama:



dónde *.data* es la parte del segmento que almacena las variables inicializadas o no, *.bss* es la parte del segmento que almacena las variables estáticas, *heap* almacena las variables dinámicas y *stack* se utiliza para llamar a las subrutinas y para almacenar variables locales de manera automática.

Con esta distribución de memoria, el espacio libre que queda de memoria RAM requiere de la utilización de la función `freeMemory()` para obtener el número de bytes disponibles.

A2.7 Gestión de la energía en Wasmote

La estructura de funcionamiento de un programa Wasmote tiene una primera parte en la que se inicializa el código mediante la función `setup()` y una función `loop()` que se ejecuta continuamente. En consecuencia, para ahorrar energía se utiliza una técnica de programación muy común que consiste en bloquear el programa, manteniendo el microprocesador en bajo consumo, hasta que alguna interrupción disponible en Wasmote indica al microprocesador que un evento se ha producido y en ese caso se ejecutará la función asociada al evento que previamente ha sido almacenada en un vector de interrupciones.

Wasmote dispone de tres modos de operación con bajo consumo, que en inglés nombramos como “Sleep”, “Deep Sleep” o “Hibernate”. Wasmote IDE implementa la gestión de energía con la librería WaspPWR.

A2.7.1 Modos de bajo consumo

Cuando un programa ejecuta una instrucción para que el microprocesador entre en estado “Sleep” o “Deep Sleep”, éste pasa a un estado latente que podrá ser despertado por cualquier interrupción asíncrona lanzada por un sensor, el acelerómetro o el módulo XBee con protocolo Digimesh. También se despertará del modo “Sleep” por la interrupción síncrona generada por el reloj “Watchdog”, mientras que la única interrupción síncrona que puede despertar del modo “Deep Sleep” es la lanzada por el RTC (Real Time Clock). En el

apartado “Gestión de interrupciones” se detalla el manejo que hace Waspote de las interrupciones.

El flujo de ejecución del programa cuando el microprocesador está en modo “Sleep” es el siguiente:

1. El procesador entra en modo “Sleep” y deja de ejecutar el programa principal al ejecutarse la instrucción *PWR.sleep()* . Se guardan todas las variables
2. Se produce una interrupción asíncrona o una interrupción del timer “Watchdog” que es capturada por el microprocesador.
3. Se activa la variable *intFlag* almacenando qué módulo ha generado la interrupción capturada.
4. Se devuelve el control al programa principal que ejecuta la rutina que trata la interrupción.
5. Vuelve el control al programa principal con las variables guardadas previamente.

El flujo de ejecución del programa cuando el microprocesador está en modo “Deep Sleep” es el siguiente:

1. El procesador entra en modo “Deep Sleep” y deja de ejecutar el programa principal al ejecutarse la instrucción *PWR.deepSleep()*. Se guardan todas las variables
2. Se produce una interrupción asíncrona o una interrupción del RTC que es capturada por el microprocesador.
3. Se activa la variable *intFlag* que almacena el módulo que ha generado la interrupción capturada.
4. Se devuelve el control al programa principal que ejecuta la rutina que trata la interrupción.
5. Vuelve el control al programa principal con las variables guardadas previamente.

A2.7.2 Modo de hibernación

En este modo el microcontrolador no almacena ninguna variable y cuando despierta, el microcontrolador es reinicializado y, por lo tanto, se ejecutan las rutinas *setup()* y *loop()* tal y como lo hace cuando el interruptor de la alimentación se enciende.

Para poder usar el modo de hibernación es obligatorio ejecutar la función *PWR.ifHibernate()* dentro de la función *setup()*. Esta función determina si el programa que se está ejecutando viene de un reinicio normal o de un reinicio desde una hibernación. También es la función que ilumina el LED rojo intermitentemente. Cuando el RTC lanza la interrupción, el microcontrolador de la Waspote se reinicia y se ejecuta la función *PWR.ifHibernate()*, que al verificar que se ha producido un reinicio desde una hibernación, activa la variable *intFlag*, almacenando así, que se ha producido una interrupción de

hibernación. Posteriormente, en la función *loop()* se puede verificar la variable *intFlag* y ejecutar la rutina que trata la interrupción.

A2.7.3 Recomendaciones sobre la gestión de la batería

Libelium recomienda escribir el programa pensando en dos tipos de bucles, uno normal y otro de bajo consumo que debe ejecutarse si el nivel de batería está por debajo de un valor preestablecido, utilizando la función *PWR.getBatteryLevel()* para detectar el nivel de batería.

Cuando se detecta un nivel bajo de batería se puede optar por ejecutar sólo parte del código del programa, notificar la situación enviando una trama o entrar en modo de bajo consumo para permitir que la batería se cargue.

También se recomienda seleccionar adecuadamente el tiempo que un dispositivo está activo. Por ejemplo, determinar la menor frecuencia posible con la que se leerá un sensor para que el dispositivo esté el menor tiempo posible ocupado.

Se debe estimar el tiempo de vida de la batería detectando el tiempo que tarda el nivel de batería en pasar de 100% a 0% y en función del resultado, optimizar al máximo el código escrito.

Para reducir el consumo de una batería debe optimizarse el uso del módulo de comunicación ya que es un módulo con gran consumo. Por lo tanto, debe determinarse cuándo es conveniente enviar/recibir información en el bucle *loop()*. Aunque es conveniente enviar una trama cada cierto tiempo para asegurar que el sistema está “vivo”.

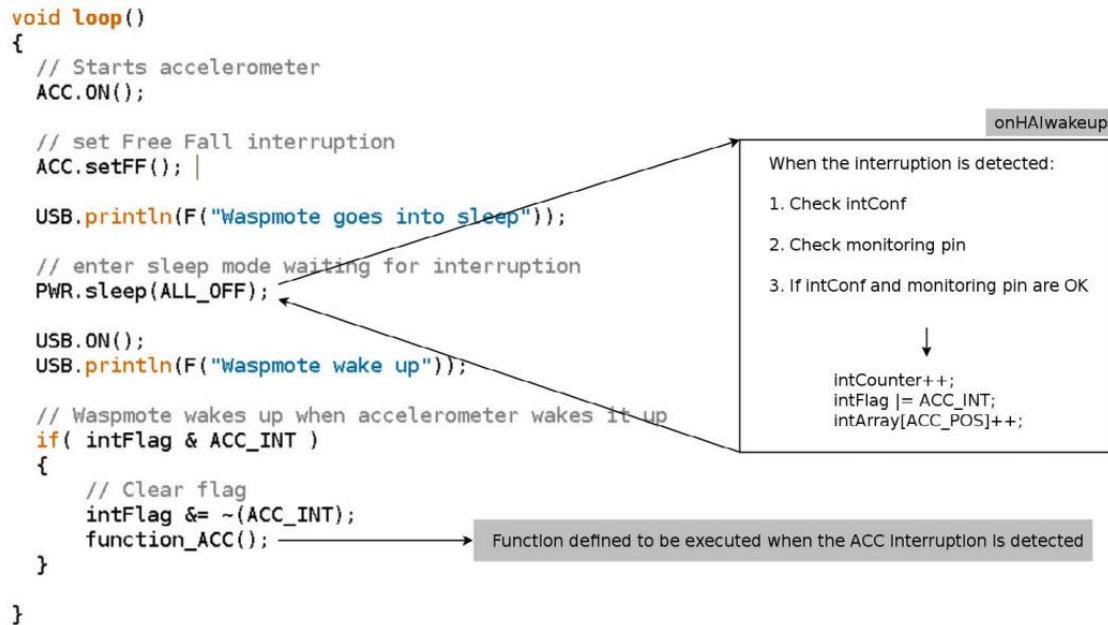
A2.8 Gestión de interrupciones en Wasmote

Wasmote utiliza las funciones almacenadas en “*Winterrupts.c*” y declaradas en “*wiring.h*” para gestionar las interrupciones que el microcontrolador puede recibir.

Como se ha visto en el apartado de “Gestión de la energía”, el microcontrolador de Wasmote permanece en un estado de bajo consumo esperando cualquier interrupción. Si un sensor supera un valor determinado (umbral) entonces advierte a Wasmote generando una interrupción, y de esa manera Wasmote recupera el control saliendo de su estado de bajo consumo.

A2.8.2 Esquema de programación de interrupciones

El uso más habitual de las interrupciones es cuando se combina con los estados de bajo consumo de Wasmote. Un ejemplo de programa ilustrativo sería:



En primer lugar se activa la posibilidad de que el acelerómetro genere una interrupción si sucede una “caída libre” mediante la instrucción `ACC.setFF()`. Después Waspote entra en el modo de bajo consumo “Sleep” y espera cualquier interrupción mediante la instrucción `PWR.sleep(ALL_OFF)`. A partir de esa línea de código no se ejecutará ninguna instrucción salvo que se produzca una interrupción. Si se produce la interrupción Waspote ejecuta el código de la función `onHALwakeUP()` o `onLALwakeUP()`. La primera se utiliza si el módulo activa la interrupción en HIGH y la segunda si el módulo activa la interrupción en LOW.

Ambas funciones ejecutan el siguiente código:

```

if( intConf & CONSTANTE_INT )
{
  if( digitalRead(CONSTANTE_INT_PIN_MON) )
  {
    intCounter++;
    intFlag |= CONSTANTE_INT;
    intArray[CONSTANTE_POS]++;
  }
}

```

Paso 1)

Paso 2)

Paso 3)

Paso 4)

Paso 5)

Paso 1): Comprueba si la interrupción que se ha producido es aceptada. Para ello verifica si el bit correspondiente del vector de 16 bits `intConf` tiene valor 1. Un 0 en el bit asignado a una interrupción significa que Waspote no aceptará interrupciones de ese módulo. Sólo las interrupciones previamente activadas en el vector `intConf` serán recibidas.

En `WaspConstants.h` se definen la `CONSTANTE_INT` que representa el módulo del que se quiere verificar si se aceptan interrupciones. Los valores más importantes utilizados son:

<code>ACC_INT</code>	Acelerómetro
<code>RTC_INT</code>	Reloj interno
<code>SENS_INT</code>	Sensores
<code>PLV_INT</code>	Pluviómetro
<code>HIB_INT</code>	Estado de hibernación
<code>XBEE_INT</code>	Módulo XBee
<code>PIR_3G_INT</code>	Sensor del módulo de videocámara
<code>WTD_INT</code>	Reloj interno Watchdog
<code>RAD_INT</code>	Sensor de radiación
<code>BAT_INT</code>	Batería crítica
<code>UART1_INT</code>	3G/GPRS interrupción

Paso 2): Se verifica si la interrupción ha sido lanzada y para ello verifica el pin activado por el módulo que lanza la interrupción comprobando si `digitalRead(pin)` es `TRUE` en la rutina `onHALwakeUP()`. O verificando si devuelve `FALSE` en la rutina `onLAIwakeUP()`.

Paso 3): Incrementa en uno el número de interrupciones recibidas y lo almacena en `intCounter`

Paso 4): Activa el bit asignado al módulo que ha generado la interrupción para indicar que la interrupción detectada proviene de dicho módulo. Se utilizan las mismas constantes que las mencionadas en el paso 1. Posteriormente, en el programa principal, el vector `intFlag` se utiliza para determinar qué interrupción se ha activado y en consecuencia, cual es la rutina que se debe lanzar para tratar la interrupción.

Paso 5: Se incrementa en uno el elemento del vector `intArray` correspondiente al módulo que ha lanzado la interrupción. Sirve para almacenar el número total de interrupciones producidas por cada módulo.

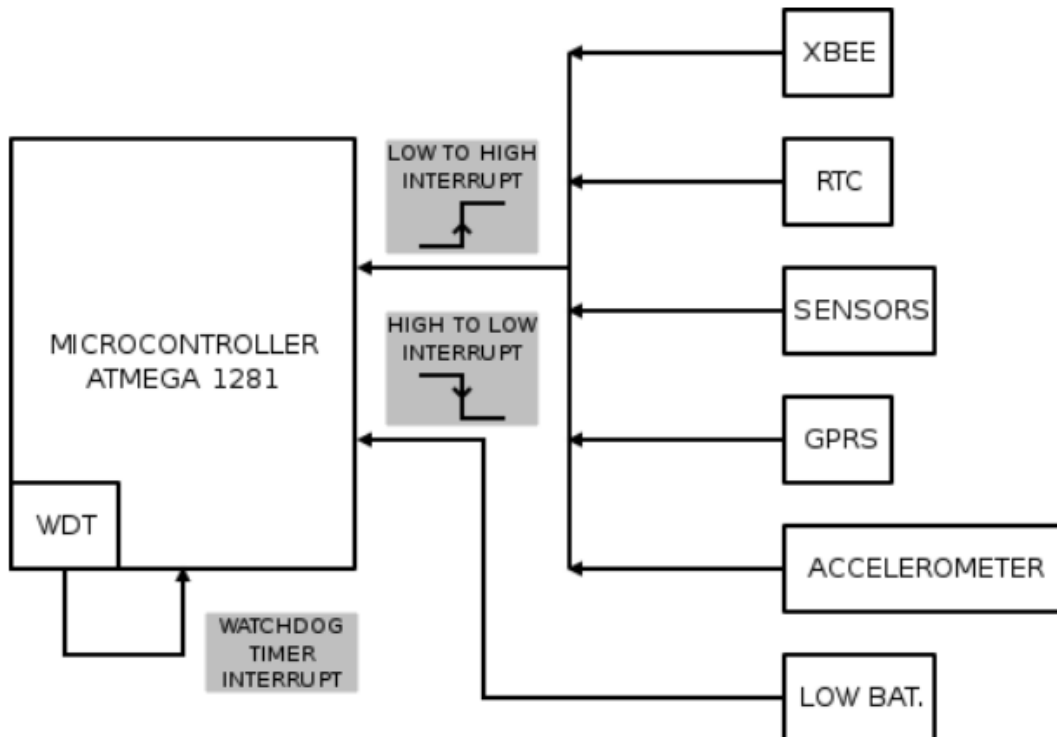
Una vez se ha ejecutado la rutina `onHALwakeUP()` o `onLAIwakeUP()`, el programa continúa en la siguiente instrucción. A partir de ese momento, el programa debe determinar qué interrupción se ha lanzado, comprobando el vector `intFlag` y ejecutando la rutina que se desee. En el ejemplo se ejecuta la función `function_ACC()`.

A2.8.2 Arquitectura de interrupciones

Waspote lleva definidos en sus librerías los módulos que pueden lanzar interrupciones. Ya hemos visto que admite interrupciones asíncronas y síncronas. También hemos visto que una vez lanzada una interrupción por un módulo cuando Waspote está en un estado de bajo consumo, se ejecutan

una de las siguientes funciones *onHALwakeUP()* o *onLAIwakeUP()* según el pin del microcontrolador que utilice el módulo para activar la interrupción.

Se dice que *onHALwakeUP()* se ejecuta cuando el módulo lanza interrupciones de alto nivel y *onLAIwakeUP()* cuando las interrupciones son de bajo nivel. Cada interrupción será tratada por *onHALwakeUP()* o *onLAIwakeUP()* según el siguiente diagrama, “low to high interrupt” y “high to low interrupt”, respectivamente.



Las señales de interrupción del RTC, acelerómetro y XBee están conectadas al pin RXD1 que es usado para capturar la interrupción y para monitorizarla.

La señal de interrupción de batería crítica está conectada al pin TXD1.

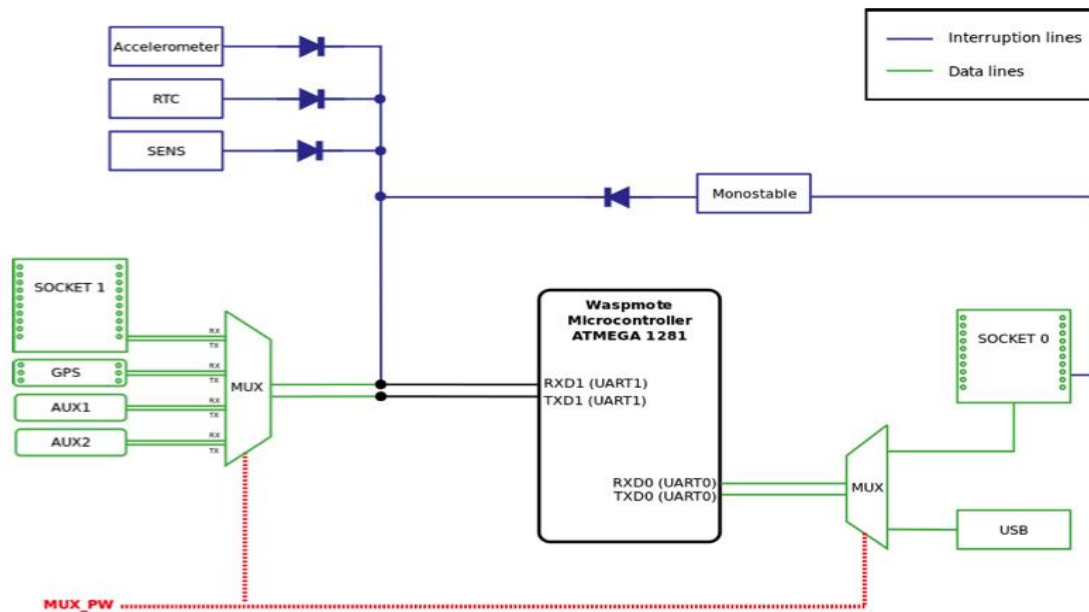
El reloj interno Watchdog simula una interrupción hardware usando un pin digital reservado (DIGITAL0). Así sus interrupciones son tratadas de la misma manera que el resto de interrupciones, utilizando la rutina *onLAIwakeup()*

Varios sensores desarrollados por Libelium utilizan las entradas analógicas o digitales del microprocesador de Waspote. Cada sensor tiene varios pines para gestionar la generación y captura de interrupciones. Hay un pin conectado al pin de interrupción del microprocesador y también existe otro pin usado para monitorización.

El módulo GSM/GPRS y el módulo 3G/GPRS está conectado al UART1 y utiliza los pines RXD1 y TXD1 para enviar y recibir datos. El módulo genera

interrupciones por el pin RXD1 y en consecuencia, cuando se produce una interrupción, se ejecuta la rutina *onLAIwakeUP()*.

El dispositivo UART1 de Wasmote y los pines de interrupciones son el mismo pin de interrupción del controlador, lo que puede producir interferencias. Por lo tanto, se recomienda no utilizar a la vez UART y los pines de interrupciones. El esquema de las líneas de interrupciones muestra la compartición del pin del microprocesador.



En particular, el módulo BLE objeto de este trabajo utiliza el Socket0 y en consecuencia, el pin del microcontrolador que puede recibir una interrupción es el RXD1, en el supuesto de que dicho módulo esté diseñado con la funcionalidad de responder a eventos que generen una interrupción dirigida al microcontrolador.

Anexo 3. Código fuente de los programas

A3.1 Programa sendAdv.pde

```

#include <WaspBLE.h>

uint16_t interval = 160; // 100 ms de intervalo por defecto
int chan = 7; // Por defecto todos los canales activos
uint8_t flags = 0x08;
bool start = false;

// Variable que almacena advData
uint8_t data[31] = {
  0x02, 0x01, 0x08, 0x05, 0x09, 0x45, 0x72, 0x69, 0x63, 0x03, 0x77, 0x00, 0x00, 0x03, 0x88, 0x00, 0x00, 0x02, 0x99,
  0x00, 0x09, 0x66, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

// uint8_t data[31] = { 0x02, 0x01, 0x1a, 0x1a, 0xff, 0x4c, 0x00, 0x02, 0x15, 0xe2, 0xc5, 0x6d, 0xb5, 0xdf, 0xfb, 0x48,
  0xd2, 0xb0, 0x60, 0xd0, 0xf5, 0xa7, 0x10, 0x96, 0xe0, 0x00, 0x00, 0x00, 0x00, 0xc5, 0x00}; iBeacon

// Inicialización Waspnote
void setup()
{
  USB.ON(); // Activa el puerto miniUSB
  PWR.switchesOFF(SENS_OFF); // Desactiva todos los sensores
}

// Módulo que se ejecuta indefinidamente
void loop(){
  if (start){
    // Si se pulsa una tecla se para la comunicación
    USB.println();
    USB.println("Send any key to stop");
    USB.println();

    USB.flush();
    while (!USB.available()) {
    }

    BLE.OFF();

    //Si se pulsa una tecla se inicia la comunicación
    USB.flush();
    USB.println("Send any key to start");
    USB.println();
    while (!USB.available()) {
    }

    // Se envía un mensaje
    runAdv();
  }
  else{
    // Configura la comunicación
    start = config();
  }
}

bool config(){
  // Seleccionar el parámetro de la comunicación que se quiere configurar
  // Opción 1 para configurar el intervalo del envío
  // Opción 2 para configurar el canal del envío
  // Opción 3 para configurar el dato flag enviado
  // o pulsad cualquier otra tecla para iniciar comunicación

  USB.println("Configure advertisement: ");
  USB.println();

```

```

USB.println(F("\t1 - Set interval advertisement"));
USB.println(F("\t2 - Select channels"));
USB.println(F("\t3 - Set flags"));
USB.println();
USB.println("Press any other key to start");
USB.println();

USB.flush();
while (!USB.available()) {
}

delay(400);

char c = USB.read();

if (c == '1')
{
    interval = setIntervals();
}

else if (c == '2')
{
    chan = (uint8_t) setChannels();
}

else if (c == '3')
{
    setFlags();
}

else
{
    USB.println("Transmitting...");
    USB.println();
    //USB.OFF();
    //delay(400);
    runAdv();
    return true;
}

return false;
}

void runAdv(){

    BLE.ON(SOCKET0);

    // 1.1 Pone el dispositivo en modo no descubrible para que no envíe.
    BLE.setDiscoverableMode(BLE_GAP_NON_DISCOVERABLE);
    USB.println(F("\tA - Stop advertisements"));

    // 1.2 Actualiza el intervalo de emisión y los canales
    /* NOTE : intervals are specified in units of 625 uS
    Example: 100 ms / 0.625 = 160
    */

    // 1.3 Da formato al mensaje
    USB.println(F("\tB - Setting advertisement interval"));
    USB.println(F("\tC - Setting data"));

    BLE.setAdvParameters(interval, interval, chan);
    BLE.setAdvData(BLE_GAP_ADVERTISEMENT, data, 31 );

    // 1.4 Activa el modo descubrible con datos de usuario e inicia la comunicación
    BLE.setDiscoverableMode(BLE_GAP_USER_DATA);
    USB.println(F("\tD - Start advertisements"));

    // 1.5 Entrar en ahorro de energía enviando mensajes
    BLE.sleep();
    USB.flush();
}

uint16_t setIntervals(){

```

```

USB.print("Set advertisement interval in ms, e.g: 100/ = 100 ms --- ");
while (!USB.available()) {
}

delay(400);

// Lee el intervalo en ms
int inbyte;
unsigned long serialdata = 0;
while (inbyte != '/')
{
  inbyte = USB.read();
  if (inbyte > 0 && inbyte != '/')
  {
    serialdata = serialdata * 10 + inbyte - '0';
  }
}
inbyte = 0;

// Conversión a las unidades de tiempo del módulo
uint16_t interval = (uint16_t) serialdata/0.625;

USB.print(serialdata);
USB.println(" ms");
USB.println();
USB.flush();

return interval;
}

int setChannels(){
// Se explica cómo se determina los canales activos
USB.println("Set advertisement channels, e.g: 7 = 39,98,37");
USB.println("0 0 0 0 0 1 0 0 - 39");
USB.println("0 0 0 0 0 0 1 0 - 38");
USB.println("0 0 0 0 0 0 0 1 - 37");
USB.println();

while (!USB.available()) {
}

delay(400);

int channels = USB.read() - '0';

USB.print("Set advertisement channels: ");
USB.println(channels);

if(channels & 0x01){
  USB.println("Channel 37 selected");
}

if(channels & 0x02){
  USB.println("Channel 38 selected");
}

if(channels & 0x04){
  USB.println("Channel 39 selected");
}

if(channels > 7){
  USB.println("No valid value");
  channels = 7;
}

USB.println();
USB.flush();
return channels;
}

void setFlags(){
// Se indica cómo seleccionar el valor flag
// que forma parte de la estructura de advData
USB.println("Set advertisement flags, e.g: 08 = Simultaneous LE and BR/EDR to Same Device Capable (Controller)");
USB.println("0 0 0 0 0 0 1 - LE Limited Discoverable Mode");

```

```
USB.println("0 0 0 0 0 1 0 - LE General Discoverable Mode");
USB.println("0 0 0 0 0 1 0 0 - BR/EDR Not Supported");
USB.println("0 0 0 0 1 0 0 0 - Simultaneous LE and BR/EDR to Same Device Capable (Controller)");
USB.println("0 0 0 1 0 0 0 0 - Simultaneous LE and BR/EDR to Same Device Capable (Host)");
USB.println();

while (!USB.available()) {
}

delay(400);

int flags = USB.read() - '0';
if(flags >= 17){
  flags -= 7;
  data[2] = ("%X", flags);
}
else{
  data[2] = ("%X", flags);
}

USB.print("Set advertisement flags: ");
USB.println(flags);

if(flags & 0x01){
  USB.println("LE Limited Discoverable Mode");
}

if(flags & 0x02){
  USB.println("LE General Discoverable Mode");
}

if(flags & 0x04){
  USB.println("BR/EDR Not Supported");
}
if(flags & 0x08){
  USB.println("Simultaneous LE and BR/EDR to Same Device Capable (Controller)");
}
USB.println();
USB.flush();
}
```

A3.2 Programa scanAdv.pde

```
#include <WaspBLE.h>
uint8_t discover = BLE_GAP_DISCOVER_OBSERVATION;
uint16_t scan_interval = 75;
uint16_t scan_window = 50;
uint8_t type = BLE_PASSIVE_SCANNING;
uint8_t time = 5;

bool start = false;

// Inicialización Wasp mote
void setup()
{
  USB.ON(); // Activa el puerto miniUSB
  PWR.switchesOFF(SENS_OFF); // Desactiva todos los sensores
}

// Módulo que se ejecuta indefinidamente
void loop(){

  if (start){
    // Inicia el escaneo
    runScan();
  }
  else{
    // Configura el escaneo
    start = config();
  }
}

bool config(){

  USB.println("Configure scanning: ");
  USB.println();
  USB.println(F("\t1 - Set discover mode"));
  USB.println(F("\t2 - Set scan interval"));
  USB.println(F("\t3 - Set scan window"));
  USB.println(F("\t4 - Set scan type"));
  USB.println(F("\t5 - Set scan time"));
  USB.println();
  USB.println("Press any other key to start");
  USB.println();

  USB.flush();
  while (!USB.available()) {
  }

  delay(400);

  char c = USB.read();

  if (c == '1')
  {
    discover = setDiscover();
  }

  else if (c == '2')
  {
    scan_interval = setInterval();
  }

  else if (c == '3')
  {
    scan_window = setWindow();
  }

  else if (c == '4')
  {
    type = setType();
  }
}
```

```

else if (c == '5')
{
    time = setScanTime();
}

else
{
    USB.println("Scanning...");
    USB.println();
    confScan();
    runScan();
    return true;
}

return false;
}

void confScan(){

    BLE.ON(SOCKET0);

    // 1.1 Set GAP Discover mode.
    BLE.setDiscoverMode(discover);

    // 1.2 set scan interval y scan window,
    // y set del tipo de escaneo
    BLE.setScanningParameters(scan_interval,scan_window, type);
}

void runScan(){
    // 2. Inicia el escaneo con los parámetros configurados y durante el tiempo definido
    USB.println(F("New scan..."));
    BLE.scanNetwork(time);
}

uint16_t setInterval(){

    USB.print("Set scan interval in ms, e.g: 100/ = 100 ms --- ");
    while (!USB.available()) {
    }

    delay(400);

    int inbyte;
    unsigned long serialdata = 0;
    while (inbyte != '/')
    {
        inbyte = USB.read();
        if (inbyte > 0 && inbyte != '/')
        {
            serialdata = serialdata * 10 + inbyte - '0';
        }
    }
    inbyte = 0;

    uint16_t interval = (uint16_t) serialdata/0.625;

    USB.print(serialdata);
    USB.println(" ms");
    USB.println();
    USB.flush();
    return interval;
}

uint16_t setScanTime(){

    USB.print("Set scan time in s, e.g: 5/ = 5 s --- ");
    while (!USB.available()) {
    }

    delay(400);

    int inbyte;
    unsigned long serialdata = 0;
    while (inbyte != '/')

```

```

{
  inbyte = USB.read();
  if (inbyte > 0 && inbyte != '/')
  {
    serialdata = serialdata * 10 + inbyte - '0';
  }
}
inbyte = 0;

uint8_t time = (uint8_t) serialdata;

USB.print(serialdata);
USB.println(" s");
USB.println();
USB.flush();
return time;
}

uint16_t setWindow(){

  USB.print("Set scan window in ms, e.g: 100/ = 100 ms --- ");
  while (!USB.available()) {
  }

  delay(400);

  int inbyte;
  unsigned long serialdata = 0;
  while (inbyte != '/')
  {
    inbyte = USB.read();
    if (inbyte > 0 && inbyte != '/')
    {
      serialdata = serialdata * 10 + inbyte - '0';
    }
  }
  inbyte = 0;

  uint16_t window = (uint16_t) serialdata/0.625;

  USB.print(serialdata);
  USB.println(" ms");
  USB.println();

  if(window > scan_interval){
    USB.println("Scan window can not exceed scan interval");
    USB.println();
    USB.flush();
    return scan_interval;
  }
  else {
    USB.flush();
    return window;
  }
}

uint8_t setDiscover(){

  USB.println("Set discover mode");
  USB.println();
  USB.println(F("\t1 - BLE_GAP_DISCOVER_LIMITED"));
  USB.println(F("\t2 - BLE_GAP_DISCOVER_GENERIC"));
  USB.println(F("\t3 - BLE_GAP_DISCOVER_OBSERVATION"));
  USB.println();

  while (!USB.available()) {
  }

  delay(400);

  char c = USB.read();

  USB.print("Set discover mode to: ");

  if(c == '1'){

```

```
    USB.println("BLE_GAP_DISCOVER_LIMITED");
    USB.println();
    USB.flush();
    return BLE_GAP_DISCOVER_LIMITED;
}

if(c == '2'){
    USB.println("BLE_GAP_DISCOVER_GENERIC");
    USB.println();
    USB.flush();
    return BLE_GAP_DISCOVER_GENERIC;
}

if(c == '3'){
    USB.println("BLE_GAP_DISCOVER_OBSERVATION");
    USB.println();
    USB.flush();
    return BLE_GAP_DISCOVER_OBSERVATION;
}
}

int setType(){

    USB.println("Set discover mode");
    USB.println();
    USB.println(F("\t1 - BLE_PASSIVE_SCANNING"));
    USB.println(F("\t2 - BLE_ACTIVE_SCANNING"));
    USB.println();

    while (!USB.available()) {
    }

    delay(400);

    char c = USB.read();

    USB.print("Set type mode to: ");

    if(c == '1'){
        USB.println("BLE_PASSIVE_SCANNING");
        USB.println();
        USB.flush();
        return BLE_PASSIVE_SCANNING;
    }

    if(c == '2'){
        type = BLE_ACTIVE_SCANNING;
        USB.println("BLE_ACTIVE_SCANNING");
        USB.println();
        USB.flush();
        return BLE_ACTIVE_SCANNING;
    }
}
```



```
        found = 1;
    }
    #else
        found = 1;
    #endif
}

// Condition to avoid an overflow (DO NOT REMOVE)
if( millis() < previous ) previous=millis();
}

// Now it is necessary to stop scan.
if(endProcedure() != 0)
{
    #if BLE_DEBUG > 0
        // copy "Stop fail. err: %x\n" form flash
        char message[25] = "";
        strcpy_P(message, (char*)pgm_read_word(&(table_BLE[5])));
        USB.printf(message, errorCode);
    #endif
    return errorCode;
}

return found;
}
```

A3.4 Modificación de la API para la generación del log

```

uint8_t WaspBLE::parseEvent()
{
    bool led = false;

    // if no event preset on event buffer, exit.
    if (event[0] == 0)
    {
        return 0;
    }

    // check class and method bytes

    if (event[2] == 0)
    {
        switch (event[3])
        {
            case 0: return BLE_EVENT_SYSTEM_BOOT;
            case 1: return BLE_EVENT_SYSTEM_DEBUG;
            case 2: return BLE_EVENT_SYSTEM_ENDPOINT_WATERMARK_RX;
            case 3: return BLE_EVENT_SYSTEM_ENDPOINT_WATERMARK_TX;
            case 4: return BLE_EVENT_SYSTEM_SCRIPT_FAILURE;
            case 5: return BLE_EVENT_SYSTEM_NO_LICENSE_KEY;
        }
    }
    else if (event[2] == 1)
    {
        switch (event[3])
        {
            case 0: return BLE_EVENT_FLASH_PS_KEY;
        }
    }
    else if (event[2] == 2)
    {
        switch (event[3])
        {
            case 0: return BLE_EVENT_ATTRIBUTES_VALUE;
            case 1: return BLE_EVENT_ATTRIBUTES_USER_READ_REQUEST;
            case 2: return BLE_EVENT_ATTRIBUTES_STATUS;
        }
    }
    else if (event[2] == 3)
    {
        switch (event[3])
        {
            case 0: return BLE_EVENT_CONNECTION_STATUS;
            case 1: return BLE_EVENT_CONNECTION_VERSION_IND;
            case 2: return BLE_EVENT_CONNECTION_FEATURE_IND;
            case 3: return BLE_EVENT_CONNECTION_RAW_RX;
            case 4: return BLE_EVENT_CONNECTION_DISCONNECTED;
        }
    }
    else if (event[2] == 4)
    {
        switch (event[3])
        {
            case 0: return BLE_EVENT_ATTCLIENT_INDICATED;
            case 1: return BLE_EVENT_ATTCLIENT_PROCEDURE_COMPLETED;
            case 2: return BLE_EVENT_ATTCLIENT_GROUP_FOUND;
            case 3: return BLE_EVENT_ATTCLIENT_ATTRIBUTE_FOUND;
            case 4: return BLE_EVENT_ATTCLIENT_FIND_INFORMATION_FOUND;
            case 5: return BLE_EVENT_ATTCLIENT_ATTRIBUTE_VALUE;
            case 6: return BLE_EVENT_ATTCLIENT_READ_MULTIPLE_RESPONSE;
        }
    }
    else if (event[2] == 5)
    {
        switch (event[3])

```

```

        {
            case 0: return BLE_EVENT_SM_SMP_DATA;
            case 1: return BLE_EVENT_SM_BONDING_FAIL;
            case 2: return BLE_EVENT_SM_PASSKEY_DISPLAY;
            case 3: return BLE_EVENT_SM_PASSKEY_REQUEST;
            case 4: return BLE_EVENT_SM_BOND_STATUS;
        }
    }
    else if (event[2] == 6)
    {
        switch (event[3])
        {
            case 0:
            {
                #if BLE_LOG == 1
                char toWrite[200];
                Utils.hex2str( event, toWrite, 65);
                SD.appendln("/data/log", toWrite);

                uint8_t mac[6];
                int8_t rssi = event[4];
                uint8_t b = 5;
                for(uint8_t a = 0; a < 6 ; a++)
                {
                    mac[a] = event[b+6];
                    b--;
                }

                char s[100];
                sprintf(s,"MAC: ");
                for (int i = 0; i < 6; i++){
                    sprintf(s + strlen(s),"%02X", mac[i]);
                }
                sprintf(s + strlen(s)," - RSSI (dB): %d - Data: ", rssi);
                for (int i = 0; i < event[14]; i++){
                    sprintf(s + strlen(s),"%02X", event[i+15]);
                }
                sprintf(s + strlen(s)," - Time: %u - ", millis());
                sprintf(s + strlen(s),RTC.getTime());

                SD.appendln("/data/log", s);
                #endif

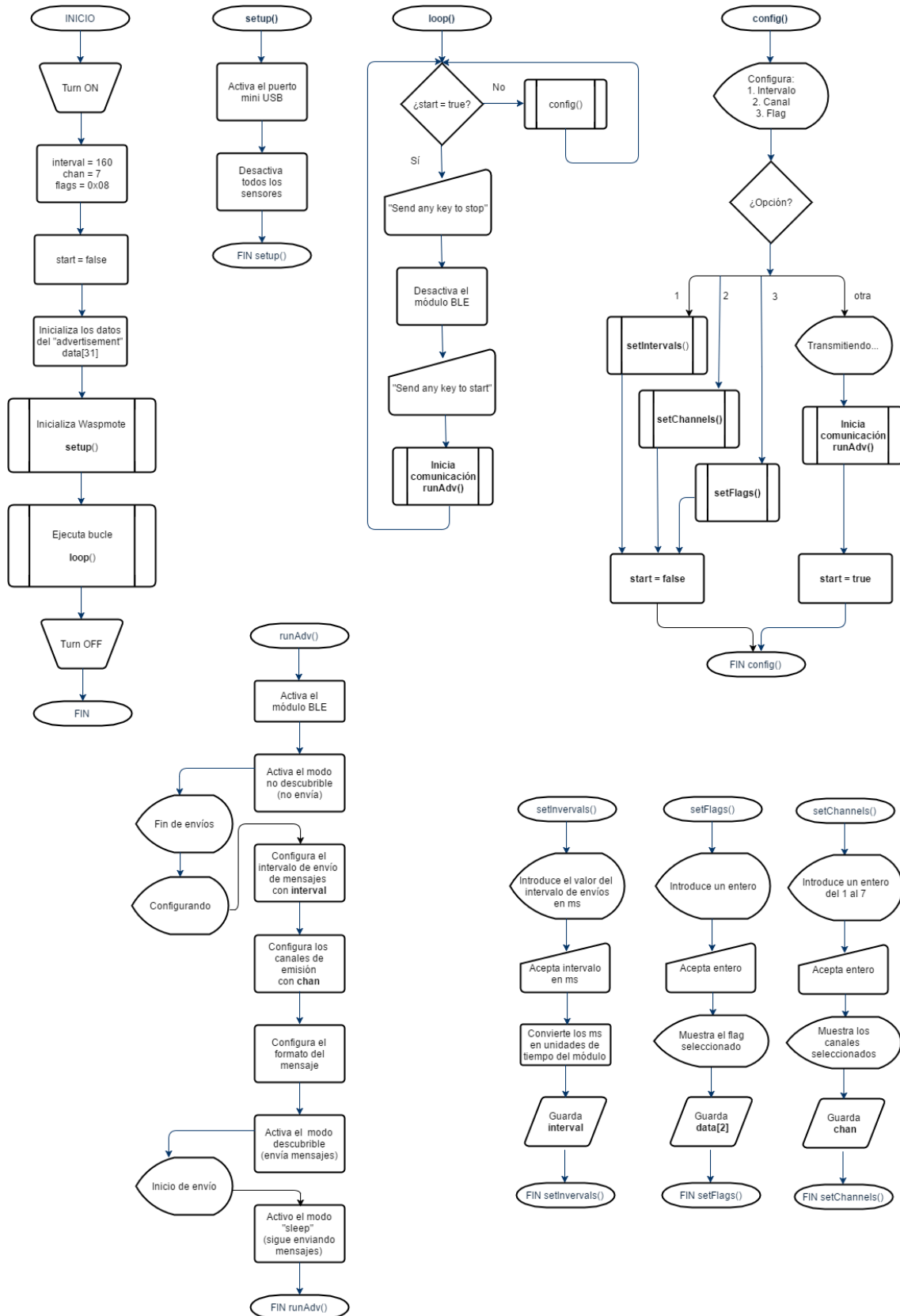
                return BLE_EVENT_GAP_SCAN_RESPONSE;
            }
            case 1: return BLE_EVENT_GAP_MODE_CHANGED;
        }
    }
    else if (event[2] == 7)
    {
        switch (event[3])
        {
            case 0: return BLE_EVENT_HARDWARE_IO_PORT_STATUS;
            case 1: return BLE_EVENT_HARDWARE_SOFT_TIMER;
            case 2: return BLE_EVENT_HARDWARE_ADC_RESULT;
        }
    }

    // if here, an error occurred.
    return 0;
}

```

Anexo 4. Diagramas de flujo

A4.1 Diagrama sendAdv.pde



A4.2 Diagrama scanAdv.pde

