Università Politecnica delle Marche

Wireless Sensor Network for IOT Project

Realize the coexistence of the "Eddystone beacon" and "iBeacon" applications (nRF SDK v16.0) inside "Light Switch Server" application (nRF SDK MESH v4.0.0)

Matteo Orlandini

Prof. Paola Pierleoni

supervised by
Andrea Gentili and Marco Mercuri

May 18, 2020

Contents

1 Introduction

Questa relazione mostra come realizzare un applicativo che implementi le funzionalità del Bluetooth Mesh e di beaconing. In particolare sono stati integrati gli applicativi "beacon Eddystone" e "iBeacon", presenti nell'SDK nRF5 v16.0, nell'esempio Light Switch dell'SDK Mesh v4.0.0.

Questo elaborato illustra tutti i passaggi necessari per il corretto funzionamento degli applicativi citati in precedenza. Sono anche presentati tutti i tentativi fatti nel corso del progetto, con la descrizione dei problemi incontrati e della relativa risoluzione, al fine di produrre l'applicazione finale funzionante.

La relazione è divisa in un'introduzione in cui viene presentata la tecnologia Bluetooth Low Energy, i beacon e il Bluetooth Mesh rispettivamente nei capitoli 1.1, 1.3 e 1.4, una descrizione degli esempi e applicativi usati e la procedura per integrare tra loro le relative funzionalità. L'esempio Light Switch è presentato nel capitolo 2, mentre gli altri esempi usati per implementare le funzioni precedentemente citate sono nei capitoli 3, 4 e 5. Una prima implementazione di applicazione che usa il Bluetooth Mesh mentre vengono trasmessi beacon è illustrata nel capitolo 6, mentre un'altra versione è descritta nel capitolo 7 e l'applicazione finale, con tutti i passaggi da seguire per sviluppare il progetto, è delineata nel capitolo 8.

1.1 Bluetooth Low Energy

La tecnologia Bluetooth è un sistema di comunicazione senza fili a corto raggio inteso a rimpiazzare i dispositivi elettronici con connessione via cavo. Le caratteristiche principali del Bluetooth sono la robustezza, il basso consumo e il basso costo.

Ci sono due forme di sistemi Bluetooth: Basic Rate (BR) e Low Energy (LE). Hanno in comune la ricerca del dispositivo, la costruzione e il meccanismo di connessione. Il Bluetooth LE include caratteristiche per lavorare con prodotti che richiedono un consumo di corrente, una complessità e un costo minori rispetto il BR.

I dispositivi che implementano entrambi i sistemi possono comunicare con altri apparati che supportano sia il Bluetooth BR sia LE. In alcuni casi è supportato solo uno dei due sistemi, quindi i dispositivi che implementano entrambi possono supportare la maggior parte delle situazioni di lavoro.

Il sistema centrale Bluetooth consiste in un host e in uno o più controller. Osservando la figura 1 si possono definire l'host come un'entità logica che contiene tutti i layer sopra la Host Controller Interface (HCI) e il controller come un'entità logica che si compone di tutti i livelli al di sotto di HCI.

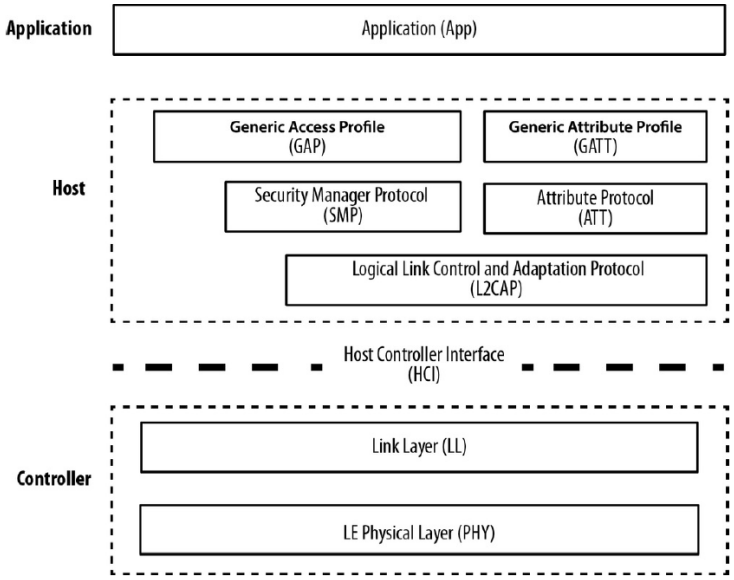


Figure 1: Stack protocollare Bluetooth Low Energy [6]

I layer inclusi nel livello host sono:

- Generic Access Profile (GAP)
- Generic Attribute Profile (GATT)
- Attribute Protocol (ATT)
- Security Manager Protocol (SMP)
- Logical Link Control and Adaptation Protocol (L2CAP)

I layer del livello controller sono:

- Link Layer (LL)
- Physical Layer (PHY)

Vengono ora descritti i vari blocchi. Il layer **Generic Access Profile** (GAP) definisce le procedure per scoprire e connettere servizi a un dispositivo Bluetooth. I servizi GAP comprendono il rilevamento dei dispositivi, le modalità di connessione, la sicurezza, l'autenticazione e i modelli di associazione.

Il **Generic Attribute Profile** (GATT) è un protocollo che definisce gli attributi e i servizi di un dispotivo BLE. Quando il profilo è configurato dal GAP, gli attributi sono assegnati a particolari servizi.

Si serve del protocollo ATT per trasportare i dati sotto forma di comandi, richieste, risposte, indicazioni, notifiche e conferme tra dispositivi. Quando si richiedono i dati sotto forma di notifiche, il server può inviare il valore di un attributo in qualsiasi momento. La figura 2 mostra la gerarchia del profilo GATT.

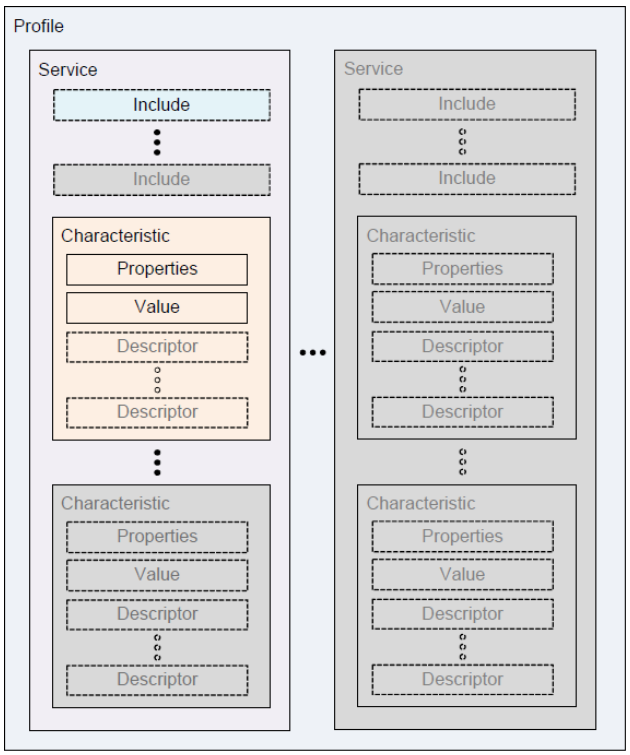


Figure 2: Gerarchia del profilo GATT [1]

Un *servizio* è una raccolta di dati e comportamenti associati per realizzare una particolare funzione o caratteristica. Nel GATT, una definizione di servizio può contenere servizi di riferimento, caratteristiche obbligatorie e caratteristiche opzionali. Esistono due tipi di servizi: primari e secondari. Un servizio primario espone la funzionalità principale utilizzabile di questo dispositivo. Un servizio secondario è un servizio a cui si intende fare riferimento solo da un servizio primario, un altro servizio secondario o altra specifica di livello superiore.

Una dichiarazione di servizio è un attributo con il tipo impostato sull'UUID per «Servizio primario» (0x2800) o «Servizio secondario» (0x2801). Il campo valore può essere un UUID a 16 o a 128 bit. Le autorizzazioni di attributo devono essere di sola lettura e non richiedono autenticazione o autorizzazione.

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
0xNNNN	0x2800 – UUID for «Primary Service» OR 0x2801 for «Secondary Service»]	16-bit Bluetooth UUID or 128-bit UUID for Service	Read Only, No Authentication, No Authorization

Figure 3: Dichiarazione di un servizio [1]

Una caratteristica è un attributo utilizzato in un servizio con proprietà e informazioni su come viene visualizzato o rappresentato il valore e su come accedervi. Una dichiarazione di una caratteristica è un attributo con il tipo impostato sull'UUID «Caratteristica» (0x2803) e il valore dell'attributo diviso in proprietà, handle e UUID. Le autorizzazioni devono essere leggibili e non richiedono autenticazione o autorizzazione.

Attribute Handle	Attribute Type s	Attribute Value			Attribute Permissions
0xNNNN	0x2803–UUID for «Characteristic»	Characteristic Properties	Characteristic Value Attribute Handle	Characteristic UUID	Read Only, No Authentication, No Authorization

Figure 4: Dichiarazione di una caratteristica [1]

Il campo valore di una caratteristica è disponibile solamente in lettura ed è illustrato nella figura di seguito.

Attribute Value	Size	Description
Characteristic Properties	1 octets	Bit field of characteristic properties
Characteristic Value Handle	2 octets	Handle of the Attribute containing the value of this characteristic
Characteristic UUID	2 or 16 octets	16-bit Bluetooth UUID or 128-bit UUID for Characteristic Value

Figure 5: Campo value nella dichiarazione della caratteristica [1]

- Il campo proprietà determina come è possibile utilizzare il valore o come accedere ai descrittori delle caratteristiche.
- Il campo handle contiente l'handle dell'attributo in cui è presente il valore della caratteristica.
- Il campo UUID è un UUID Bluetooth a 16 bit o UUID generico a 128 bit che descrive il tipo di valore della caratteristica.

I *descrittori* delle caratteristiche vengono utilizzati per contenere informazioni correlate al valore della caratteristica. Il profilo GATT definisce un set standard di descrittori che possono essere utilizzati da profili di livello superiore. Ogni descrittore di caratteristiche è identificato dall'UUID.

Il *Client Characteristic Configuration Descriptor* (CCCD) è un descrittore di caratteristiche facoltativo che definisce come la caratteristica possa essere configurata da uno specifico client. Un client può scrivere nel CCCD per controllare la configurazione della caratteristica. Possono essere richieste autenticazione e autorizzazione dal server per scrivere nel descrittore. Il descrittore è contenuto in un attributo il cui tipo deve essere impostato sull'UUID come «Client Characteristic Configuration» (0x2902). Il CCCD agisce come un interruttore, abilitando o disabilitando gli aggiornamenti del campo "value" del "characteristic value" della caratteristica in cui si trova. Il suo valore è un bitfield a due bit, uno corrispondente alle notifiche e l'altro alle indicazioni.

Attribute Handle	Attribute Type	Attribute Value	Attribute Permissions
0xNNNN	02902 – UUID for «Client Characteristic Configuration»	Characteristic Configuration Bits	Readable with no authentication or authorization. Writable with authentication and authorization defined by a higher layer specification or is implementation specific.

Figure 6: Dichiarazione *Client Characteristic Configuration* [1]

I *Characteristic Configuration Bits* possibili sono elencati in tabella:

Configurazione	Valore	Descrizione
Notifiche	0x0001	Il campo value sarà notificato
Indicazioni	0x0002	Il campo value sarà indicato

Table 1: Definizione dei *Characteristic Configuration Bits* [1]

Il **protocollo ATT** definisce due ruoli: *server* e *client*. Il server è il dispositivo che accetta comandi e richieste in arrivo dal client e a cui invia risposte, indicazioni e notifiche. Il client, invece, è il dispositivo che avvia comandi e richieste verso il server e può ricevere risposte, indicazioni e notifiche inviate dal server. Gli attributi di un server possono essere scoperti, scritti o letti da un client specifico. Un attributo è composto da quattro parti: *handle*, *tipo*, *valore* e *permessi*. La figura 7 mostra una rappresentazione logica di un attributo.

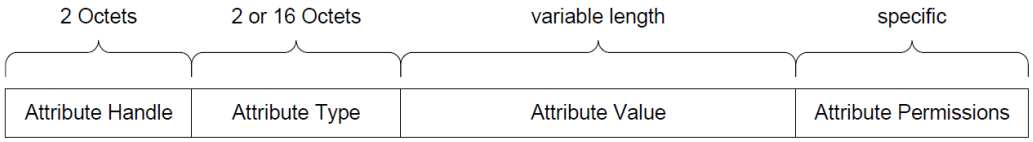


Figure 7: Rappresentazione logica di un attributo [1]

L'*handle* è un indice corrispondente a un attributo specifico. Il *tipo* dell'attributo è un UUID (universally unique identifier) che specifica ciò che esso rappresenta. Il *valore* è il dato descritto dal tipo dell'attributo e indicizzato dall'handle. Le *autorizzazioni* vengono utilizzate dal server per determinare se è consentito l'accesso in lettura o in scrittura per un determinato attributo e sono stabilite dal profilo GATT.

Il **Security Manager Protocol** (SMP) è utilizzato per generare chiavi di crittografia. Il protocollo opera su un canale L2CAP dedicato e gestisce anche l'archiviazione delle chiavi di crittografia e di identità. Si interfaccia direttamente con il controller per fornire chiavi memorizzate durante le procedure di crittografia o associazione.

Il blocco **Logical Link Control and Adaptation Protocol** (L2CAP) è responsabile della gestione dell'ordinamento dei frammenti dei pacchetti in banda base e della pianificazione dei canali L2CAP per garantire che questi non abbiano l'accesso negato al canale fisico a causa dell'esaurimento delle risorse del controller. Ciò è necessario perché l'architettura non presuppone che un controller abbia un buffering illimitato o che l'HCI abbia a disposizione una larghezza di banda infinita.

L'**Host Controller Interface** (HCI) è un'interfaccia standardizzata grazie alla quale dispositivi host possono accedere agli strati più bassi dello stack Bluetooth. Attraverso l'HCI un host può inviare dati agli altri dispositivi e ricevere informazioni dalle altre unità presenti nella piconet. Con questa interfaccia un host può ordinare al suo layer baseband di creare un link ad uno specifico dispositivo Bluetooth, eseguire richieste di autenticazione e passare una chiave per il link.

Il **Link Layer** (LL) è la parte che si interfaccia direttamente al livello fisico (PHY). È responsabile dell'advertising, della scansione e della creazione e gestione delle connessioni.

Il **Physical Layer** (PHY) è responsabile della trasmissione e della ricezione di pacchetti di informazioni sul canale fisico. Un percorso di controllo tra la banda di base e il blocco PHY consente al blocco di banda di base di controllare il timing e la portante del blocco PHY. Trasforma uno stream di dati da e verso il canale fisico e la banda base nei formati richiesti.

1.2 Panoramica delle operazioni Bluetooth Low Energy

La radio del Bluetooth LE opera nella banda 2.4 GHz con una ricetrasmissione frequency hopping, supportando il bit rate di 1 Megabit al secondo (1 Mb/s)

Il Bluetooth LE usa due tipi di accesso multiplo: Frequency Division Multiple Access (FDMA) e Time Division Multiple Access (TDMA). Nello

schema FDMA sono usati 40 canali fisici separati da 2 MHz, 3 canali sono usati per advertising e 37 sono usati per i dati. La TDMA è sfruttata quando un dispositivo trasmette ad un predeterminato intervallo temporale e il corrispondente apparato risponde dopo un periodo prestabilito.

Nella figura 8 viene presentata la suddivisione dei canali nel Bluetooth Low Energy.

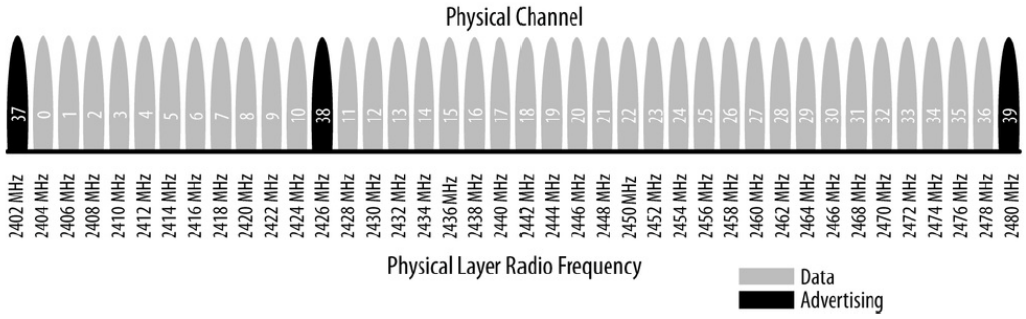


Figure 8: Suddivisione dei canali del Bluetooth LE [6]

Il canale fisico è suddiviso in unità temporali chiamate *events*. I dati sono trasmessi in pacchetti posizionati in questi eventi che si dividono in advertising e connection.

I dispositivi che trasmettono i pacchetti di advertising sono chiamati *advertisers*, quelli che il ricevono senza intenzioni di connettersi al dispositivo di advertising sono chiamati *scanners*. Ciascun dispositivo Bluetooth è identificato da un Bluetooth device address, cioè un numero di 48 bit (6 byte). Può essere di due tipi: pubblico, se è un indirizzo fisso, o random, se cambia ad ogni avvio dell'applicazione. Ogni dispositivo advertiser trasmette, ad intervalli regolari che vanno dai 20 ms ai 10.24 secondi pacchetti di grandezza fino a 31 byte contenenti: il nome, l'indirizzo, la classe del dispositivo, la lista dei servizi offerti e altre informazioni (es. marca). Questa operazione è detta advertising. In base al tipo di pacchetto, lo scanner può fare una richiesta all'advertiser, seguita eventualmente da una risposta da parte di quest'ultimo. Il canale di advertising cambia al successivo pacchetto trasmesso nello stesso evento. L'advertiser può terminare un evento in qualsiasi momento, all'inizio del successivo evento di advertising viene usato il primo canale fisico come mostrato in figura 9

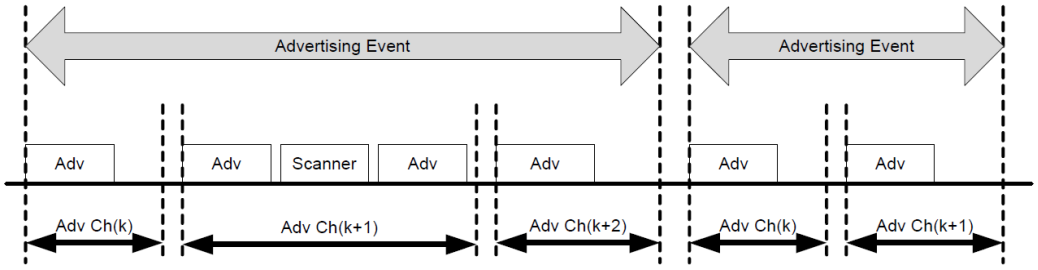


Figure 9: Eventi advertising [1]

I dispositivi che tentano di connettersi con un altro dispositivo vengono chiamati *iniziatori*. Se l'advertiser sta usando un evento di advertising, un iniziatore può effettuare una richiesta di connessione usando lo stesso canale in cui è stato ricevuto il pacchetto. L'evento è così terminato e la connessione inizia se l'advertiser riceve e accetta la richiesta mandata. Una volta che la connessione è stabilita, l'iniziatore diventa il master e l'advertiser è lo slave. Gli eventi connection sono usati per mandare pacchetti di dati tra master e slave nello stesso canale. Il master inizia e finisce ogni evento di connessione.

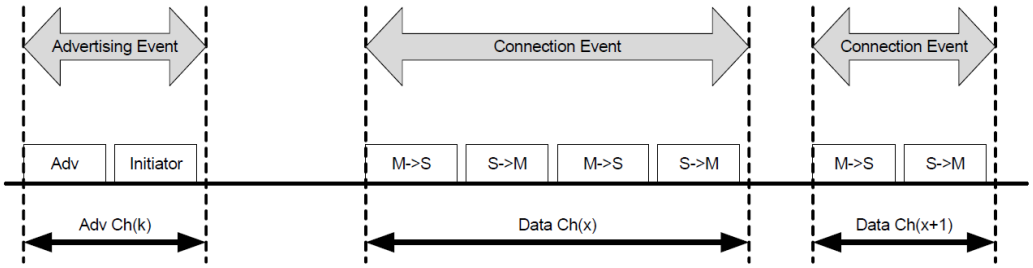


Figure 10: Eventi connection [1]

I dispositivi in una piconet, cioè un gruppo di apparati sincronizzati, usano la tecnica FHSS (Frequency Hopping Spread-Spectrum) che consiste nel commutare il canale di trasmissione con una frequenza di 1600 volte al secondo. L'algoritmo usato per cambiare le frequenze nel Bluetooth LE, condiviso tra il trasmettitore dati ed il ricevitore, è pseudo random nello scegliere le 37 frequenze disponibili, alcune delle quali possono essere escluse per evitare disturbi da altri dispositivi nella stessa area, permettendo inoltre di ridurre al minimo la probabilità di interferenza.

Nella richiesta di connessione, il master invia dei dati chiamati *connection parameters*: essi sono un set di parametri che determinano quando e come il central (master) ed il peripheral (slave) scambieranno i dati durante la connessione. Questi parametri vengono sempre stabiliti dal central, ma

la periferica può inviare una *connection parameter update request*, ovvero un pacchetto di dati contenente dei "suggerimenti" di possibili parametri da utilizzare. Questi parametri sono:

- *Connection Interval*: determina quanto spesso il central chiederà il dato alla periferica. Lo standard, infatti, stabilisce un intervallo di possibili valori compreso tra 7,5 ms e 4 s. La periferica suggerisce un ulteriore intervallo, intrinseco al connection interval, compreso tra due valori in ms (i cui estremi sono stabiliti dai parametri MIN_CONN_INTERVAL e MAX_CONN_INTERVAL), ma è il central ad adottarne uno tra i possibili.

- *Slave latency*: è il numero di richieste di lettura dei dati consecutive inviate dal central alla periferica che verranno ignorate. Il suo valore è uno dei possibili compresi nel range tra 0x0000 (cioè ad ogni richiesta di lettura la periferica risponde con l'invio dei dati) e 0x01F3 (499, cioè alla 500esima richiesta di lettura la periferica risponderà con l'invio dei dati, trascurando tutte le precedenti 499). Ciò consente alla periferica di rimanere in modalità sleep per un tempo lungo se non ha dei dati aggiornati da inviare, permettendo così di risparmiare energia.

- *Connection supervision timeout multiplier*: poiché i device coinvolti nella connessione non sono in grado di capire quando questa viene persa, è necessario che passi un tempo "abbastanza" lungo, chiamato timeout, affinché avvenga il trasferimento dei dati tra i due dispositivi prima di presupporre che la connessione sia stata persa. Ha un range compreso tra 100 ms e 32 secondi (3200 ms). È chiaro inoltre che il timeout dovrà essere maggiore del connection interval.

1.3 BLE Beacon

A beaconing wireless technology is the concept of broadcasting small pieces of information. The information may be anything, ranging from ambient data (temperature, air pressure, humidity, and so forth) to micro-location data (asset tracking, retail, and so forth) or orientation data (acceleration, rotation, and so forth).The transmitted data is typically static but can also be dynamic and change over time. With the use of Bluetooth Low Energy (BLE), beacons can be designed to run for years on a single coin cell battery.

BLE has the ability to exchange data in one of two states: connected and advertising modes, as described in chapter 1.2. Connected mode uses the Generic Attribute (GATT) layer to transfer data in a one-to-one connection. Advertising mode uses the Generic Access Profile (GAP) layer to broadcast

data out to anyone who is listening. Advertising mode is a one-to-many transfer and has no guarantees about data coherence. BLE Beacons take advantage of the GAP advertising mode to broadcast data out in periodic, specially formatted advertising packets. Each type of beacon uses a custom specification to partition up the advertising data, giving it meaning.

The non-connectable beacon is a BLE device in broadcasting mode. It simply transmits information that is stored internally. Because the non-connectable broadcasting does not activate any receiving capabilities, it achieves the lowest possible power consumption by simply waking up, transmit data and going back to sleep.

The connectable beacon is a BLE device in peripheral mode, which means that it can not only transmit, but receive as well. This allows a central device (for example, a smartphone) to connect and interact with services implemented on the beacon device. Services provide one or more characteristics that could be modified by a peer device.

The transmitted data from a BLE device is formatted according to the Bluetooth Core Specification and is comprised of the parts shown in Fig. 11. The Preamble is a 1 byte value used for synchronization and timing estima-

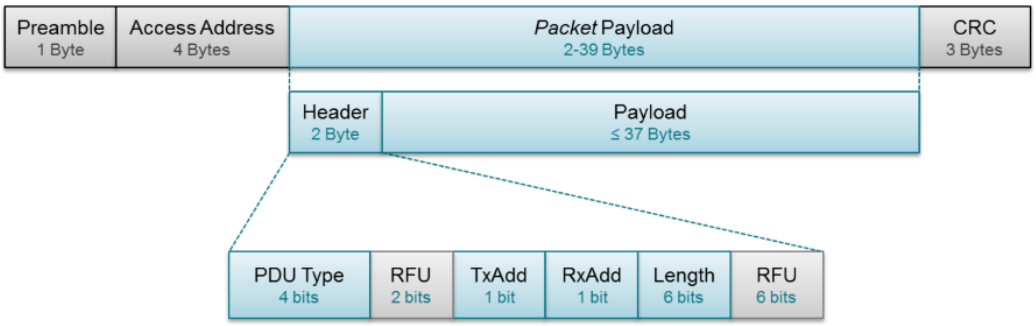


Figure 11: BLE Data Packet [5]

tion at the receiver. It will always be 0xAA for broadcasted packets. The Access Addressis also fixed for broadcasted packets, set to 0x8E89BED6. The packet payload consists of a header and payload. The header describes the packet type and the PDU Type defines the purpose of the device. For broadcasting applications, there are three different PDU Types, as shown in Table 2. ADV_IND and ADV_NONCONN_IND have been described previously (as connectable and non-connectable) while ADV_SCAN_IND is simply a non-connectable broadcaster that can provide additional information by scan responses.

PDU Type	Packet Name	Description
0000	ADV_IND	Connectable advertising event
0010	ADV_NONCONN_IND	Non-connectable advertising event
0110	ADV_SCAN_IND	Scannable advertising event

Table 2: Advertising PDU Types for Broadcasting Data [5]

The TxAdd bit indicates whether the advertiser's address(contained in the Payload) is public (TxAdd = 0) or random (TxAdd = 1). RxAdd is reserved for other types of packets not covered in this application note,as they do not apply to beacons. The final part of the transmitted packet is the Cyclic Redundancy Check (CRC). CRC is an error-detecting code used to validate the packet for unwanted alterations.It ensures data integrity for all transmitted packets over the air. The Payload of the packet includes the advertiser's address along with the user defined advertised data as shown in Fig. 12. These fields represent the beacon's broadcasted address and data.

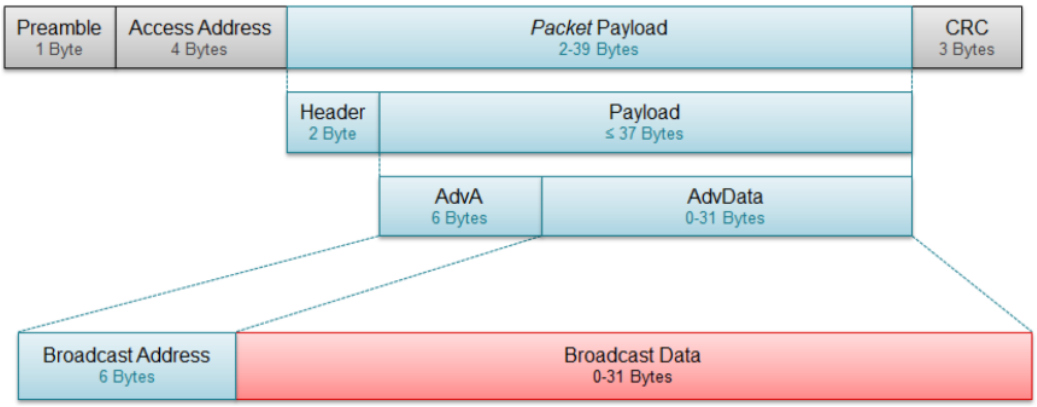


Figure 12: BLE Broadcast Data [5]

The broadcast address can be either public or random. A public address is an IEEE802-2001 standard and uses an Organizationally Unique Identifier (OUI) obtained from the IEEE Registration Authority. Random addresses can be directly generated by the beacon

The broadcasted data can be formatted according to Bluetooth SIG specified data formats, with some examples shown in Table 3. The first byte of broadcast data is the length of the data. For the purpose of this application report, the focus is on the Flags and Manufacturer-Specific Data.

AD Data Type	Data Type Value	Description
Flags	0x01	Device discovery capabilities
Service UUID	0x02 - 0x07	Device GATT services
Local Name	0x08 - 0x09	Device name
TX Power Level	0x0A	Device output power
Manufacturer Specific Data	0xFF	User defined

Table 3: Advertisement Data Types [5]

For the flags, the first three bytes of the broadcasted data defines the capabilities of the device. When using Manufacturer-Specific Data,the 0xFF flag is used to indicate so. The first two bytes of the data itself should be a company identifier code.

Now I'm going to take a look at two existing types of beacons: Eddystone and iBeacon.

1.3.1 Eddystone

Eddystone is an open beacon format developed by Google and designed with transparency and robustness in mind. Eddystone can be detected by both Android and iOS devices. The Eddystone format builds on lessons learned from working with industry partners in existing deployments, as well as the wider beacon community. The Eddystone PDU is shown in Fig. 13

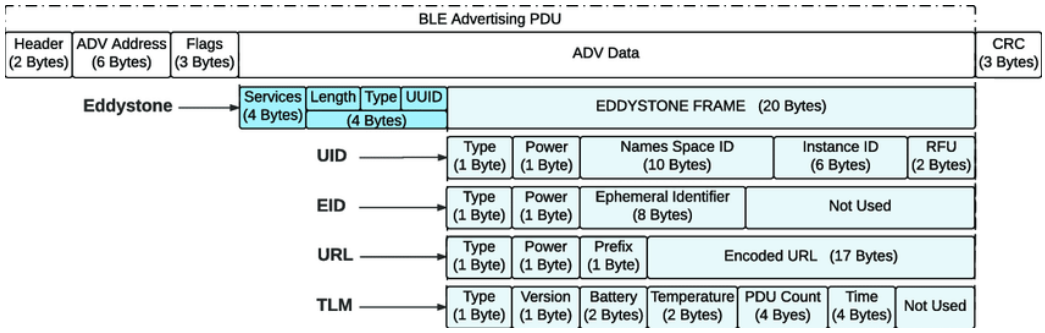


Figure 13: Eddystone PDU [2]

Several different types of payload can be included in the frame format, including:

- Eddystone-UID: A unique, static ID with a 10-byte Namespace component and a 6-byte Instance component.

- Eddystone-URL: A compressed URL that, once parsed and decompressed, is directly usable by the client.
- Eddystone-TLM: Beacon status data that is useful for beacon fleet maintenance, and powers Google Proximity Beacon API's diagnostics endpoint. -TLM should be interleaved with an identifying frame such as Eddystone-UID or Eddystone-EID (for which the encrypted eTLM version preserves security).
- Eddystone-EID: A time-varying beacon frame that can be resolved to a stable identifier by a linked resolver, such as Proximity Beacon API.

The application that shows the transmission of Eddystone beacon is described in the chapter 3.

1.3.2 iBeacon

Apple's iBeacon was the first BLE Beacon technology to come out, so most beacons take inspiration from the iBeacon data format. iBeacons are enabled in several of the Apple SDKs and can be read and broadcast from any BLE-enabled iDevice. The iBeacon is a proprietary, closed standard. There is a large ecosystem around iBeacons and a large pool of resources for developers, but you have to be part of Apple's developer community.

iBeacons broadcast four pieces of information:

- A UUID that identifies the beacon.
- A Major number identifying a subset of beacons within a large group.
- A Minor number identifying a specific beacon.
- A TX power level in 2's compliment, indicating the signal strength one meter from the device. This number must be calibrated for each device by the user or manufacturer.

A scanning application reads the UUID, major number and minor number and references them against a database to get information about the beacon; the beacon itself carries no descriptive information - it requires this external database to be useful. The TX power field is used with the measured signal strength to determine how far away the beacon is from the smart phone. Please note that Tx Power must be calibrated on a beacon-by-beacon basis by the user to be accurate. An example of iBeacon is shown in Fig. 14.

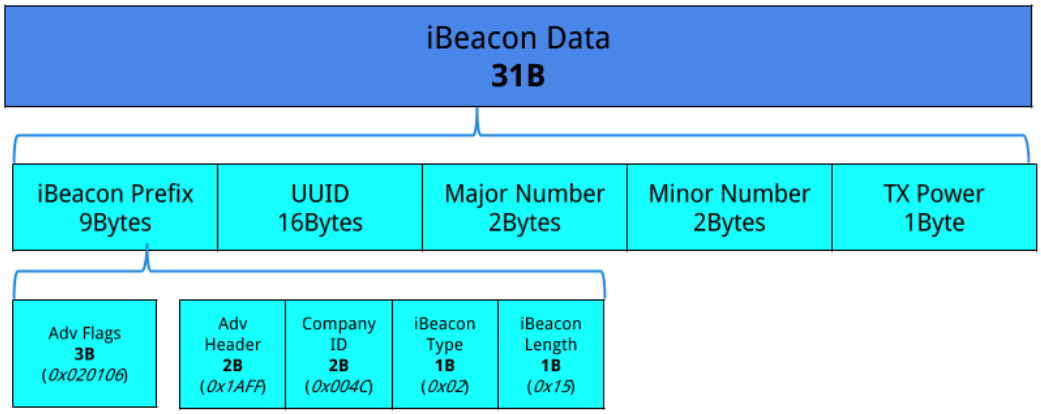


Figure 14: iBeacon Data

The iBeacon Prefix contains the hex data : 0x0201061AFF004C0215. This breaks down as follows:

- 0x020106 defines the advertising packet as BLE General Discoverable and BR/EDR high-speed incompatible. Effectively it says this is only broadcasting, not connecting.
- 0x1AFF says the following data is 26 bytes long and is Manufacturer Specific Data.
- 0x004C is Apple's Bluetooth Sig ID and is the part of this spec that makes it Apple-dependent.
- 0x02 is a secondary ID that denotes a proximity beacon, which is used by all iBeacons.
- 0x15 defines the remaining length to be 21 bytes (16+2+2+1).

The remaining fields are rather self explanatory. The proximity UUID is a standard 16byte/128bit BLE UUID and is typically unique to a company. The major and minor numbers are used to denote assets within that UUID; common uses are major numbers being stores (so 65,536 stores possible) with minor numbers being individual tags within the stores (again 65,536 possible tags per store).

The application that shows the transmission of iBeacon is described in the chapter 4.

1.4 Mesh Bluetooth

Bluetooth Mesh is a profile specification developed and published by the Bluetooth SIG. This document explains the basic concepts of the Bluetooth Mesh and gives an overview of the operation and capabilities of the profile, as well as explaining the life cycle of a mesh device.

Bluetooth mesh runs on top of the Bluetooth Low Energy (LE) stack. Figure 15 below outlines the Bluetooth mesh stack and defines the functionality of each layer. On-air, the Bluetooth Mesh physical representation is compatible with existing Bluetooth low energy devices, as mesh messages are contained inside the payload of Bluetooth low energy advertisement packets. However, Bluetooth Mesh specifies a completely new host layer, and although some concepts are shared, Bluetooth Mesh is incompatible with the Bluetooth low energy host layer.

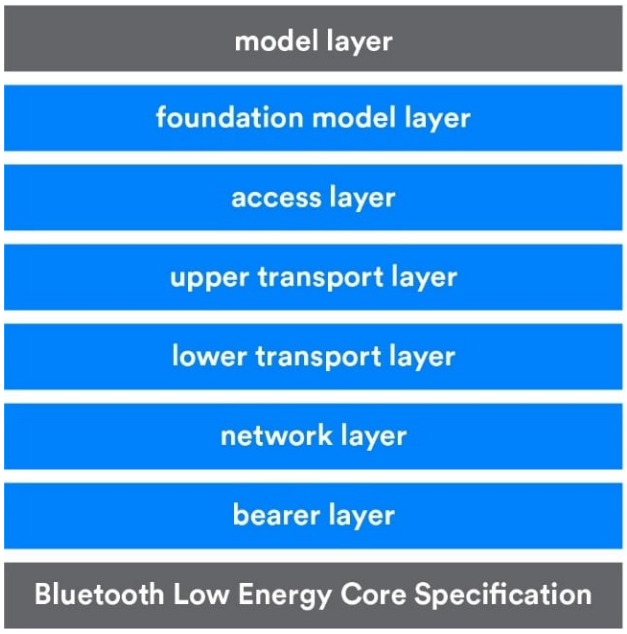


Figure 15: Bluetooth Mesh Architecture [7]

1.4.1 Bluetooth Mesh - The Basics

In this section, we'll explore the most fundamental of the terms and concepts of Mesh Bluetooth.

Mesh vs. Point-to-Point

Most Bluetooth LE devices communicate with each other using a simple **point-to-point** network topology enabling one-to-one device communications. In the Bluetooth core specification, this is called a "piconet". Imagine

a smartphone that has established a point-to-point connection to a heart rate monitor over which it can transfer data. One nice aspect of Bluetooth is that it enables devices to set up multiple connections. That same smartphone can also establish a point-to-point connection with an activity tracker. In this case, the smartphone can communicate directly with each of the other devices, but the other devices cannot communicate directly with each other.In contrast, a **mesh** network has a **many-to-many** topology, with each device able to communicate with every other device in the mesh. Communication is achieved using messages, and devices are able to relay messages to other devices so that the end-to-end communication range is extended far beyond the radio range of each individual node.

Devices and Nodes

Devices which are part of a mesh network are called **nodes** and those which are not are called "**unprovisioned devices**". The process which transforms an unprovisioned device into a node is called **provisioning**. Consider purchasing a new Bluetooth light with mesh support, bringing it home and setting it up. To make it part of your mesh network, so that it can be controlled by your existing Bluetooth light switches and dimmers, you would need to provision it. Provisioning is a secure procedure which results in an unprovisioned device possessing a series of encryption keys and being known to the **provisioner** device, typically a tablet or smartphone. One of these keys is called the network key or NetKey for short. You can read more about mesh security in the Security section, below. All nodes in a mesh network possess at least one NetKey and it is possession of this key which makes a device a member of the corresponding network and as such, a node. There are other requirements that must be satisfied before a node can become useful, but securely acquiring a NetKey through the provisioning process is a fundamental first step.

Elements

Some nodes have multiple, constituent parts, each of which can be independently controlled. In Bluetooth mesh terminology, these parts are called **elements**. An LED lighting product with three lights if added to a Bluetooth mesh network, would form a single node with three elements, one for each of the individual LED lights.

Messages

When a node needs to query the status of other nodes or needs to control other nodes in some way, it sends a **message** of a suitable type. If a node needs to report its status to other nodes, it sends a message. All communi-

cation in the mesh network is "message-oriented" and many message types are defined, each with its own, unique opcode. Messages fall within one of two broad categories; **acknowledged** or **unacknowledged**.

Acknowledged messages require a response from nodes that receive them. The response serves two purposes: it confirms that the message it relates to was received, and it returns data relating to the message recipient to the message sender.The sender of an acknowledged message may resend the message if it does not receive the expected response(s) and therefore, acknowledged messages must be idempotent. This means that the effect of a given acknowledged message, arriving at a node multiple times, will be the same as it had only been received once. Unacknowledged messages do not require a response.

Addresses

Messages must be sent from and to an **address**. Bluetooth mesh defines three types of address. A **unicast address** uniquely identifies a single element. Unicast addresses are assigned to devices during the provisioning process. A **group address** is a multicast address which represents one or more elements. Group addresses are either defined by the Bluetooth Special Interest Group (SIG) and are known as SIG Fixed Group Addresses or are assigned dynamically. 4 SIG Fixed Group Addresses have been defined. These are named All-proxies, All-friends, All-relays and All-nodes. The terms Proxy, Friend, and Relay will be explained later in this paper.It is expected that dynamic group addresses will be established by the user via a configuration application and that they will reflect the physical configuration of a building, such as defining group addresses which correspond to each room in the building. A **virtual address** is an address which may be assigned to one or more elements, spanning one or more nodes. It takes the form of a 128-bit UUID value with which any element can be associated and is much like a label. Virtual addresses will likely be preconfigured at the point of manufacture and be used for scenarios such as allowing the easy addressing of all meeting room projectors made by this manufacturer.

Publish/Subscribe

The act of sending a message is known as **publishing**. Nodes are configured to select messages sent to specific addresses for processing, and this is known as **subscribing**. Typically, messages are addressed to group or virtual addresses. Group and virtual address names will have readily understood meaning to the end user, making them easy and intuitive to use. In Figure 16 we can see that the node "Switch 1" is publishing to group address Kitchen. Nodes Light 1, Light 2, and Light 3 each subscribe to the Kitchen

address and therefore receive and process messages published to this address. In other words, Light 1, Light 2, and Light 3 can be switched on or off using Switch 1. Switch 2 publishes to the group address Dining Room. Light 3 alone subscribed to this address and so is the only light controlled by Switch 2. Note that this example also illustrates the fact that nodes may subscribe to messages addressed to more than one distinct address. This is both powerful and flexible.Similarly, notice how both nodes Switch 5 and Switch 6 publish to the same Garden address. The use of group and virtual addresses with the publish/subscribe communication model has an additional, substantial benefit in that removing, replacing or adding new nodes to the network does not require reconfiguration of other nodes. Consider what would be involved in installing an additional light in the dining room. The new device would be added to the network using the provisioning process and configured to subscribe to the Dining Room address. No other nodes would be affected by this change to the network. Switch 2 would continue to publish messages to Dining Room as before but now, both Light 3 and the new light would respond.

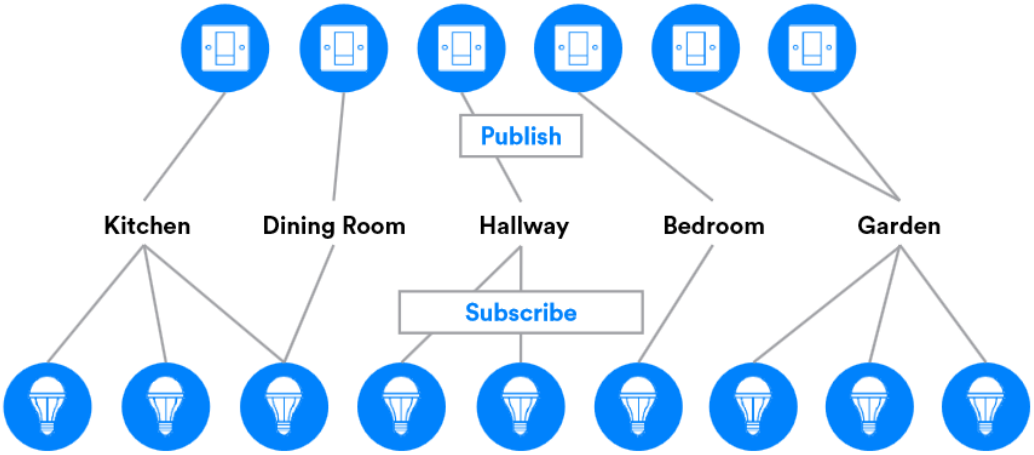


Figure 16: Publish/Subscribe.[7]

States and Properties

Elements can be in various conditions and this is represented in Bluetooth mesh by the concept of **state** values. A state is a value of a certain type, contained within an element (within a server model - see below). As well as values, States also have associated behaviors and may not be reused in other contexts. As an example, consider a simple light which may either be on or off. Bluetooth mesh defines a state called **Generic OnOff**. The light would possess this state item and a value of On would correspond to and

cause the light to be illuminated whereas a Generic OnOff state value of Off would reflect and cause the light to be switched off.The significance of the term Generic will be discussed later. **Properties** are similar to states in that they contain values relating to an element. But they are significantly different to states in other ways. Readers who are familiar with Bluetooth LE will be aware of characteristics and recall that they are data types with no defined behaviors associated with them, making them reusable across different contexts. A property provides the context for interpreting a characteristic. To appreciate the significance and use of contexts as they relate to properties, consider for example, the characteristic Temperature 8, an 8-bit temperature state type which has a number of associated properties, including Present Indoor Ambient Temperature and Present Outdoor Ambient Temperature. These two properties allow a sensor to publish sensor readings in a way that allows a receiving client to determine the context the temperature value has, making better sense of its true meaning.Properties are organized into two categories: **Manufacturer**, which is a read-only category and **Admin** which allows read-write access.

Messages, States and Properties

Messages are the mechanism by which operations on the mesh are invoked. Formally, a given message type represents an operation on a state or collection of multiple state values. All messages are of three broad types, reflecting the types of operation which Bluetooth mesh supports. The shorthand for the three types is **GET**, **SET** and **STATUS**. GET messages request the value of a given state from one or more nodes. A STATUS message is sent in response to a GET and contains the relevant state value. SET messages change the value of a given state. An acknowledged SET message will result in a STATUS message being returned in response to the SET message whereas an unacknowledged SET message requires no response. STATUS messages are sent in response to GET messages, acknowledged SET messages or independently of other messages, perhaps driven by a timer running on the element sending the message, for example. Specific states referenced by messages are inferred from the message opcode. Properties on the other hand, are referenced explicitly in generic property related messages using a 16-bit property ID.

Models

Models pull the preceding concepts together and define some or all of the functionality of an element as it relates to the mesh network. Three categories of model are recognized. A **server** model defines a collection of states, state transitions, state bindings and messages which the element containing

the model may send or receive. It also defines behaviors relating to messages, states and state transitions. A **client** model does not define any states. Instead, it defines the messages which it may send or receive in order to GET, SET or acquire the STATUS of states defined in the corresponding server model. **Control** models contain both a server model, allowing communication with other client models and a client model which allows communication with server models.Models may be created by extending other models. A model which is not extended is called a root model. Models are immutable, meaning that they may not be changed by adding or removing behaviors. The correct and only permissible approach to implementing new model requirements is to extend the existing model.

Generics

It is recognized that many different types of device, often have semantically equivalent states, as exemplified by the simple idea of ON vs OFF. Consider lights, fans and power sockets, all of which can be switched on or turned off. Consequently, the Bluetooth mesh model specification, defines a series of reusable, **generic states** such as, for example, Generic OnOff and Generic Level. Similarly, a series of **generic messages** that operate on the generic states are defined. Examples include Generic OnOff Get and Generic Level Set. Generic states and generic messages are used in **generalized models**, both generic server models such as the Generic OnOff Server and Generic Client Models such as the Generic Level Client. **Generics** allow a wide range of device type to support Bluetooth mesh without the need to create new models. Remember that models may be created by extending other models too. As such, generic models may form the basis for quickly creating models for new types of devices.

Provisioning

Provisioning is the process by which a device joins the mesh network and becomes a node. It involves several stages, results in various security keys being generated and is itself a secure process.Provisioning is accomplished using an application on a device such as a tablet. In this capacity, the device used to drive the provisioning process is referred to as the *Provisioner*. The provisioning process progresses through five steps and these are described next.

Step 1. Beaconing

In support of various different Bluetooth mesh features, including but not limited to provisioning, new GAP AD types have been introduced, including the «Mesh Beacon» AD type. An unprovisioned device indicates its availability to be provisioned by using the *Mesh Beacon* AD type in advertising

packets. The user might need to start a new device advertising in this way by, for example, pressing a combination of buttons or holding down a button for a certain length of time.

Step 2. Invitation

In this step, the Provisioner sends an invitation to the device to be provisioned, in the form of a Provisioning Invite PDU. The Beaconing device responds with information about itself in a *Provisioning Capabilities* PDU.

Step 3. Exchanging Public Keys

The Provisioner and the device to be provisioned, exchange their public keys, which may be static or ephemeral, either directly or using an out-of-band (OOB) method.

Step 4. Authentication

During the authentication step, the device to be provisioned outputs a random, single or multi-digit number to the user in some form, using an action appropriate to its capabilities. For example, it might flash an LED several times. The user enters the digit(s) output by the new device into the Provisioner and a cryptographic exchange takes place between the two devices, involving the random number, to complete the authentication of each of the two devices to the other.

Step 5. Distribution of the Provisioning Data

After authentication has successfully completed, a session key is derived by each of the two devices from their private keys and the exchanged, peer public keys. The session key is then used to secure the subsequent distribution of the data required to complete the provisioning process, including a security key known as the network key (NetKey). After provisioning has completed, the provisioned device possesses the network's NetKey, a mesh security parameter known as the IV Index and a Unicast Address, allocated by the Provisioner. It is now known as a node.

Features

All nodes can transmit and receive mesh messages but there are a number of optional features which a node may possess, giving it additional, special capabilities. There are four such optional features: the **Relay**, **Proxy**, **Friend**, and the **Low Power** features. A node may support zero or more of these optional features and any supported feature may, at a point in time, be enabled or disabled.

Relay Nodes

Nodes which support the Relay feature, known as **Relay** nodes, are able to retransmit received messages. Relaying is the mechanism by which a message can traverse the entire mesh network, making multiple "*hops*" between de-

vices by being relayed. Mesh network PDUs include a field called TTL (Time To Live). It takes an integer value and is used to limit the number of hops a message will make across the network. Setting TTL to 3, for example, will result in the message being relayed, a maximum number of three hops away from the originating node. Setting it to 0 will result in it not being relayed at all and only traveling a single hop. Armed with some basic knowledge of the topology and membership of the mesh, nodes can use the TTL field to make more efficient use of the mesh network.

Low Power Nodes and Friend Nodes

Some types of node have a limited power source and need to conserve energy as much as possible. Furthermore, devices of this type may be predominantly concerned with sending messages but still have a need to occasionally receive messages.Consider a temperature sensor which is powered by a small coin cell battery. It sends a temperature reading once per minute whenever the temperate is above or below configured upper and lower thresholds. If the temperature stays within those thresholds it sends no messages. These behaviors are easily implemented with no particular issues relating to power consumption arising.However, the user is also able to send messages to the sensor which change the temperature threshold state values. This is a relatively rare event but the sensor must support it. The need to receive messages has implications for duty cycle and as such power consumption. A 100% duty cycle would ensure that the sensor did not miss any temperature threshold configuration messages but use a prohibitive amount of power. A low duty cycle would conserve energy but risk the sensor missing configuration messages.The answer to this apparent conundrum is the **Friend** node and the concept of friendship. Nodes like the temperature sensor in the example may be designated **Low Power** nodes (LPNs) and a feature flag in the sensor's configuration data will designate the node as such. LPNs work in tandem with another node, one which is not power-constrained (e.g. it has a permanent AC power source). This device is termed a Friend node. The Friend stores messages addressed to the LPN and delivers them to the LPN whenever the LPN polls the Friend node for "waiting messages". The LPN may poll the Friend relatively infrequently so that it can balance its need to conserve power with the timeliness with which it needs to receive and process configuration messages. When it does poll, all messages stored by the Friend are forwarded to the LPN, one after another, with a flag known as MD (More Data) indicating to the LPN whether there are further messages to be sent from the Friend node.The relationship between the LPN and the Friend node is known as friendship. Friendship is key to allowing very power constrained nodes which need to receive messages, to function in a Bluetooth mesh net-

work whilst continuing to operate in a power-efficient way.

Proxy Nodes

There are an enormous number of devices in the world that support Bluetooth LE, most smartphones and tablets being amongst them. In-market Bluetooth devices, at the time Bluetooth mesh was adopted, do not possess a Bluetooth mesh networking stack. They do possess a Bluetooth LE stack however and therefore have the ability to connect to other devices and interact with them using GATT, the Generic Attribute Profile. **Proxy nodes** expose a GATT interface which Bluetooth LE devices may use to interact with a mesh network. A protocol called the *Proxy Protocol*, intended to be used with a connection-oriented bearer, such as GATT is defined. GATT devices read and write Proxy Protocol PDUs from within GATT characteristics implemented by the Proxy node. The Proxy node transforms these PDUs to/from mesh PDUs. In summary, Proxy nodes allow Bluetooth LE devices that do not possess a Bluetooth mesh stack to interact with nodes in a mesh network.

Node Configuration

Each node supports a standard set of **configuration states** which are implemented within the standard Configuration Server Model and accessed using the Configuration Client Model. Configuration State data is concerned with the node's capabilities and behavior within the mesh, independently of any specific application or device type behaviors. For example, the features supported by a node, whether it is a Proxy node, a Relay node and so on, are indicated by Configuration Server states. The addresses to which a node has subscribed are stored in the Subscription List. The network and subnet keys indicating the networks the node is a member of are listed in the configuration block, as are the application keys held by the mode. A series of configuration messages allow the Configuration Client Model and Configuration Server Model to support GET, SET and STATUS operations on the Configuration Server Model states.

1.4.2 Mesh System Architecture

In this section, we'll take a closer look at the Bluetooth mesh architecture, its layers and their respective responsibilities. We'll also position the mesh architecture relative to the Bluetooth LE core architecture. At the bottom of the mesh architecture stack, as shown in Fig. 15, we have a layer entitled Bluetooth LE. In fact, this is more than just a single layer of the mesh architecture, it's the full Bluetooth LE stack, which is required to provide

fundamental wireless communications capabilities which are leveraged by the mesh architecture which sits on top of it. It should be clear that the mesh system is dependent upon the availability of a Bluetooth LE stack. We'll now review each layer of the mesh architecture, working our way up from the bottom layer.

Bearer Layer

Mesh messages require an underlying communications system for their transmission and receipt. The bearer layer defines how mesh PDUs will be handled by a given communications system. At this time, two bearers are defined and these are called the **Advertising Bearer** and the **GATT Bearer**. The Advertising Bearer leverages Bluetooth LE's GAP advertising and scanning features to convey and receive mesh PDUs. The GATT Bearer allows a device which does not support the Advertising Bearer to communicate indirectly with nodes of a mesh network which do, using a protocol known as the Proxy Protocol. The Proxy Protocol is encapsulated within GATT operations involving specially defined GATT characteristics. A mesh Proxy node implements these GATT characteristics and supports the GATT bearer as well as the Advertising Bearer so that it can convert and relay messages between the two types of bearer.

Network Layer

The network layer defines the various message address types and a network message format which allows transport layer PDUs to be transported by the bearer layer. It can support multiple bearers, each of which may have multiple network interfaces, including the local interface which is used for communication between elements that are part of the same node. The network layer determines which network interface(s) to output messages over. An input filter is applied to messages arriving from the bearer layer, to determine whether or not they should be delivered to the network layer for further processing. Output messages are subject to an output filter to control whether or not they are dropped or delivered to the bearer layer.The Relay and Proxy features may be implemented by the network layer.

Lower Transport Layer

The lower transport layer takes PDUs from the upper transport layer and sends them to the lower transport layer on a peer device. Where required, it performs segmentation and reassembly of PDUs. For longer packets, which will not fit into a single Transport PDU, the lower transport layer will perform segmentation, splitting the PDU into multiple Transport PDUs. The receiving lower transport layer on the other device, will reassemble the seg-

ments into a single upper transport layer PDU and pass this up the stack.

Upper Transport Layer

The upper transport layer is responsible for the encryption, decryption and authentication of application data passing to and from the access layer. It also has responsibility for transport control messages, which are internally generated and sent between the upper transport layers on different peer nodes. These include messages related to friendship and heartbeats.

Access Layer

The access layer is responsible for defining how applications can make use of the upper transport layer. This includes:

- Defining the format of application data.
- Defining and controlling the encryption and decryption process which is performed in the upper transport layer.
- Verifying that data received from the upper transport layer is for the right network and application, before forwarding the data up the stack.

Foundation Models Layer

The foundation model layer is responsible for the implementation of those models concerned with the configuration and management of a mesh network.

Models Layer

The model layer is concerned with the implementation of Models and as such, the implementation of behaviors, messages, states, state bindings and so on, as defined in one or more model specifications.

1.5 Introduction to BLE IoT

The nRF5 SDK lets nRF5 devices connect and communicate with other devices over the Internet using Bluetooth low energy (BLE). By providing drivers, libraries, examples, and APIs, the SDK is an all-purpose tool for getting started with the Internet of Things (IoT).

All devices in the Internet of Things must be uniquely identifiable, so that direct communication between the devices is possible and each device can be individually addressed. One way of accomplishing this is to assign a unique IPv6 address to each device and handle all communication through IPv6.

This SDK provides means to use BLE links to connect IoT devices to a BLE enabled router, which in turn is connected to the Internet through

IPv6. All IoT devices are assigned individual IPv6 addresses, and the BLE link is used to transmit the IPv6 packages. In this way, the SDK abstracts applications from the underlying BLE technology and provides a way to directly address all devices using IPv6.

Each IoT device can directly communicate with other devices using their IPv6 address; no matter if the other device is connected to the same router or is located somewhere else on the Internet, and no matter if the other device is using a BLE link or not. In addition, all applications can use the same protocol (for example MQTT, CoAP, or even HTTP) to communicate, which means that heterogeneous types of devices - wired, wireless, or clouds - can talk a common language.

Figure 17 illustrates an IoT network with devices that are connected to the Internet through BLE enabled routers.

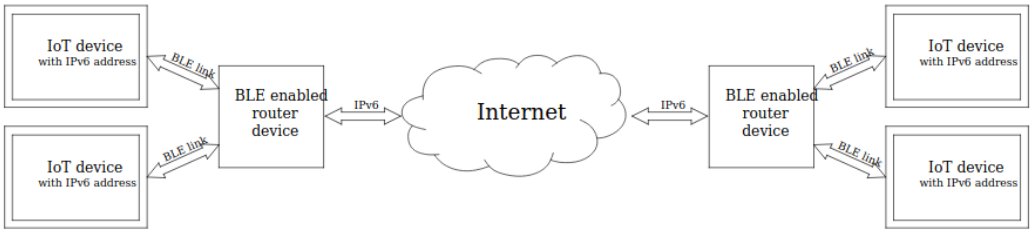


Figure 17: IoT network using BLE links [3]

1.6 Installare l'ambiente di sviluppo e scaricare l'SDK Nordic

1. Installare IDE Segger Embedded Studio dal seguente link.
2. Scaricare nRF5 SDK dal seguente link ed estrarre il file zip nella directory che si desidera utilizzare per lavorare con l'SDK.
3. Scaricare nRF5 SDK for Mesh dal seguente link ed estrarre il file zip nella directory che si desidera utilizzare per lavorare con l'SDK. consigliato estrarre il contenuto nella stessa cartella del SDK scaricato al punto 2.

1.7 Building the mesh stack and examples

1.7.1 First time setup

SEGGER Embedded Studio (SES) provides a way of quickly getting the example code up and running with full debug capability.

Before building the mesh examples with SEGGER Embedded Studio for the first time, you must complete a one-time setup of the SDK_ROOT macro in SEGGER Embedded Studio. This macro is used to find the nRF5 SDK files.

You can either:

- Use the default settings of the SDK_ROOT macro. It defaults to an nRF5 SDK instance unzipped right next to the mesh folder.
- Set the SDK_ROOT macro to a custom nRF5 SDK instance.

To set the SDK_ROOT macro manually in SEGGER Embedded Studio:

Use the default settings of the SDK_ROOT macro. It defaults to an nRF5 SDK instance unzipped right next to the mesh folder. Set the SDK_ROOT macro to a custom nRF5 SDK instance.

To set the SDK_ROOT macro manually in SEGGER Embedded Studio:

1. Go to **Tools** → **Options**.
2. Select **Building**.
3. Under **Build** in the configuration list, edit **Global macros** to contain SDK_ROOT=<the path to nRF5 SDK instance>.
4. Save the configuration.

You can verify the path by opening one of the source files under the nRF5 SDK file group. If the macro is set correctly, the file opens in the editor window. If not, an error message is displayed with information that the file cannot be found.

1.7.2 Building with SES

By default, the nRF5 SDK for Mesh package includes the SES project files for all examples. This allows you to quickly start building examples with SES.

However, if you make changes to any of the CMakeLists.txt files, you must generate the SES project files again using CMake.

To build an example with SEGGER Embedded Studio:

1. Open the desired project file located in the examples/ folder, for instance examples/light_switch/client/light_switch_client_nrf52832_xxAA_s132_6_1_1.emP
2. Go to **Build** → **Build** < **name of the emProject file**>, for instance **Build light_switch_client_nrf52832_xxAA_s132_7.0.1**.

3. Wait for the compilation to finish.

You can now run examples using SEGGER Embedded Studio.

1.7.3 Running examples using SEGGER Embedded Studio

Running examples using SEGGER Embedded Studio
The following procedure only works if you have built the example with SEGGER Embedded Studio.

To run the examples in a build environment based on SEGGER Embedded Studio (SES):

1. Connect the Development Kit with the USB cable to your computer.
2. In SES, connect to the development kit with **Target** → **Connect J-Link**.
3. Erase the device from the SES options menu: **Target** → **Erase all**.
4. Run the example with **Debug** → **Go**. This downloads the matching SoftDevice and the compiled example and starts the debugger.
5. When the download is complete, select **Debug** → **Go** again to start the code execution.

If the debugging does not start, reset the J-Link: **Target** → **Reset J-Link**.

2 Light Switch Example

This example demonstrates the mesh ecosystem that contains devices acting in two roles: Provisioner role and Node role (also referred to as provisionee role). It also demonstrates how to use Mesh models by using the Generic OnOff model in an application. The example is composed of three minor examples:

- Light switch server: A minimalistic server that implements a Generic OnOff server model, which is used to receive the state data and control the state of LED 1 on the board.
- Light switch client: A minimalistic client that implements four instances of a Generic OnOff client model. When a user presses any of the buttons, an OnOff Set message is sent out to the configured destination address.

- Mesh Provisioner: A simple static provisioner implementation that sets up the demonstration network. This provisioner provisions all the nodes in one mesh network. Additionally, the provisioner also configures key bindings and publication and subscription settings of mesh model instances on these nodes to enable them to talk to each other.

Note

For provisioning purposes, you can either use the static provisioner example or use the nRF Mesh mobile app.

The Generic OnOff Client/Server is used for manipulating the on/off state. Note that when the server has a publish address set (as in this example), the server will publish any operation of its state change to its publish address.

The Fig. 18 gives the overall view of the mesh network that will be set up by the static provisioner. Numbers in parentheses indicate the addresses that are assigned to these nodes by the provisioner.

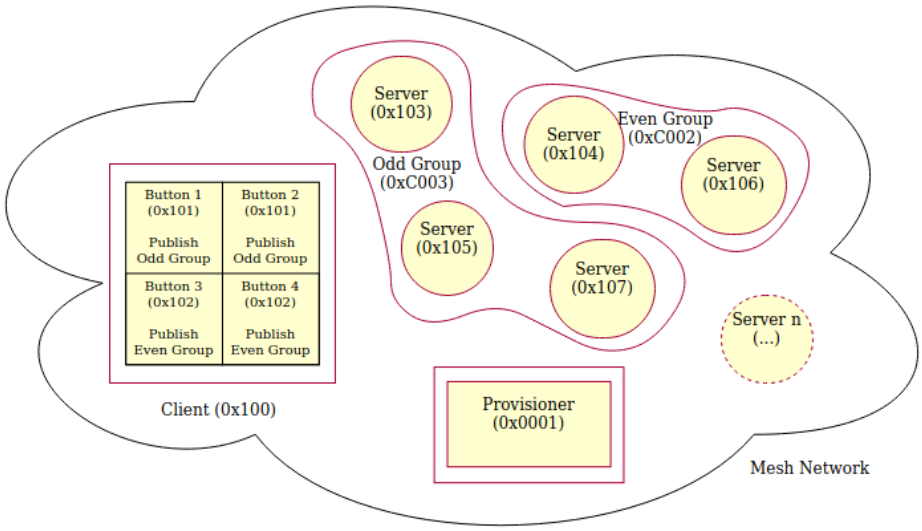


Figure 18: Mesh network demonstration [4]

Both the light switch server and light switch client examples have provisionee role. They support provisioning over Advertising bearer (PB-ADV) and GATT bearer (PB-GATT) and also support Mesh Proxy Service (Server).

2.1 Hardware requirements

You need at least two supported boards for this example:

- One nRF5 development board for the client.
- One or more nRF5 development boards for the servers.

Additionally, you need one of the following:

- One nRF5 development board for the provisioner if you decide to use the static provisioner example.
- nRF Mesh mobile app (iOS or Android) if you decide to provision using the application.

2.2 Setup

You can find the source code of this example and its minor examples in the following folder: <InstallFolder>\examples\light_switch

2.3 LED and button assignments

The buttons (1 to 4) are used to initiate certain actions, and the LEDs (1 to 4) are used to reflect the status of actions as follows:

- Server:
  - During provisioning process:
    - LED 3 and 4 blinking: Device identification active.
    - LED 1 to 4: Blink four times to indicate provisioning process is completed.
  - After provisioning and configuration is over:
    - LED 1: Reflects the value of OnOff state on the server.
      - LED ON: Value of the OnOff state is 1 (true).
      - LED OFF: Value of the OnOff state is 0 (false).
- Client:
  - During provisioning process:
    - LED 3 and 4 blinking: Device identification active.
    - LED 1 to 4: Blink four times to indicate provisioning process is completed.
  - After provisioning and configuration is over, buttons on the client are used to send OnOff Set messages to the servers:

    * Button 1: Send a message to the odd group (address: 0xC003) to turn on LED 1.
    * Button 2: Send a message to the odd group (address: 0xC003) to turn off LED 1.
    * Button 3: Send a message to the even group (address: 0xC002) to turn on LED 1.
    * Button 4: Send a message to the even group (address: 0xC002) to turn off LED 1.

- Provisoner:
  - Button 1: Start provisioning.
  - LED 1: Reflects the state of the provisioning.
    * LED ON: Provisioning of the node is in progress.
    * LED OFF: No ongoing provisioning process.
  - LED 2: Reflects the state of the configuration.
    * LED ON: Configuration of the node is in progress.
    * LED OFF: No ongoing configuration process.

2.4 Evaluating using the static provisioner

1. Flash the examples by following the instructions in running examples using SEGGER Embedded Studio, including:
   (a) Erase the flash of your development boards and program the SoftDevice.
   (b) Flash the provisioner and the client firmware on individual boards and the server firmware on other boards.
2. After the reset at the end of the flashing process, press Button 1 on the provisioner board to start the provisioning process:
   - The provisioner first provisions and configures the client and assigns the address 0x100 to the client node.
   - The two instances of the OnOff client models are instantiated on separate secondary elements. For this reason, they get consecutive addresses starting with 0x101.
   - Finally, the provisioner picks up the available devices at random, assigns them consecutive addresses, and adds them to odd and even groups.

3. Observe that the LED 1 on the provisioner board is turned ON when provisioner is scanning and provisioning a device.

4. Observe that the LED 2 on the provisioner board is turned ON when configuration procedure is in progress.

5. Wait until LED 1 on the provisioner board remains ON steadily for a few seconds, which indicates that all available boards have been provisioned and configured.

If the provisioner encounters an error during the provisioning or configuration process for a certain node, you can reset the provisioner to restart this process for that node.

2.5 Evaluating using the nRF Mesh mobile app

Per testare l'esempio con lo smartphone è necessario scaricare l'app nRF Mesh. Quando l'applicazione finisce la scansione si possono vedere i due dispositivi client e server come in figura 19

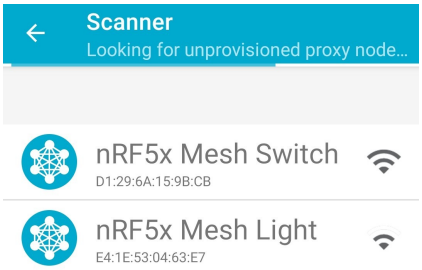


Figure 19: Fase di scanning dei dispositivi

Cliccando ad esempio su nRF5x Mesh Switch si ottiene la seguente schermata

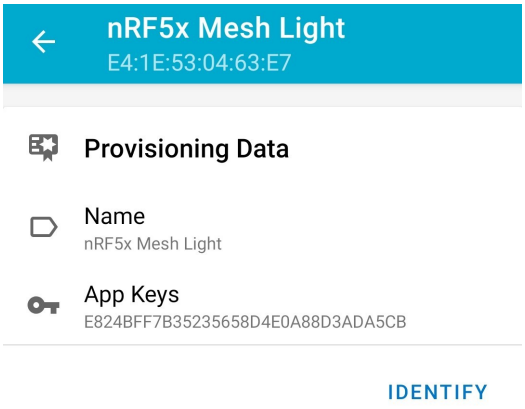


Figure 20: Provisioning Data

Cliccando su IDENTIFY, compaiono poi l'elenco delle **capabilities** del nodo. A questo punto cliccare su PROVISION per fare il provisioning del nodo.

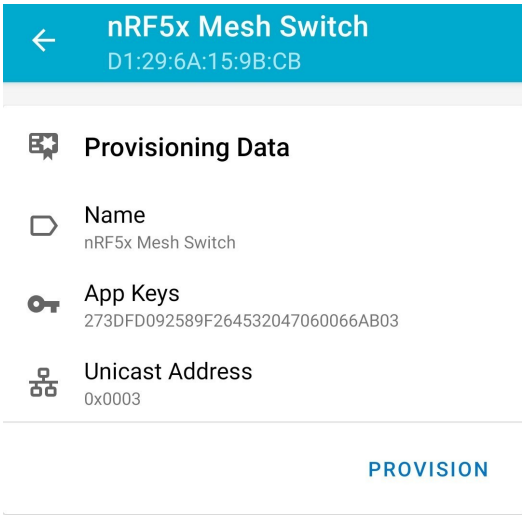


Figure 21: Provision del nodo

I nodi ora configurati e si ottiene la seguente schermata

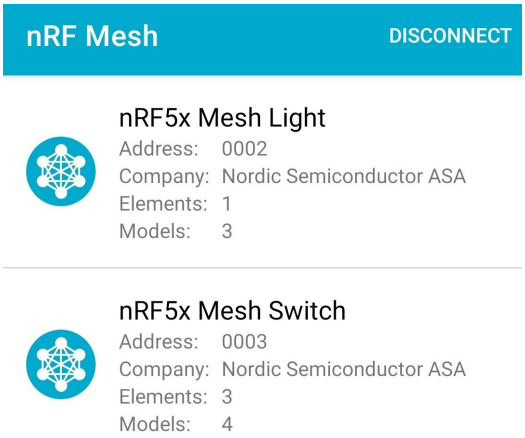


Figure 22: Nodi configurati

Per accendere o spegnere il led è necessario associare un chiave. Cliccare quindi su BIND KEY

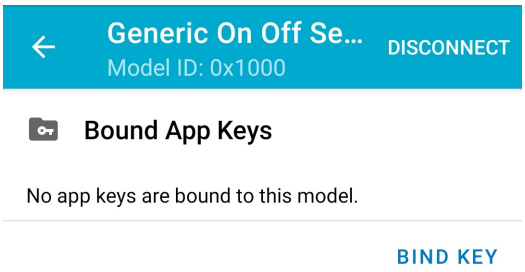


Figure 23: App key

La figura 24 mostra la schermata finale in cui si può leggere lo stato del nodo e accendere o spegnere il led.

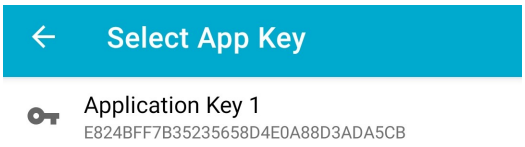


Figure 24: Application key

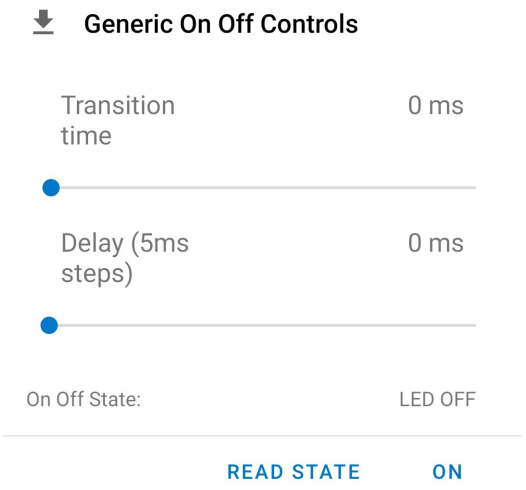


Figure 25: Controlli On Off generici

2.6 Interacting with the boards

Once the provisioning and the configuration of the client node and at least one of the server nodes are complete, you can press buttons on the client to see the LEDs getting toggled on the associated servers. See led and button assignments section.

If an RTT terminal is available and connected to the client, sending the ASCII numbers 0–3 will have the same effect as pressing the buttons.

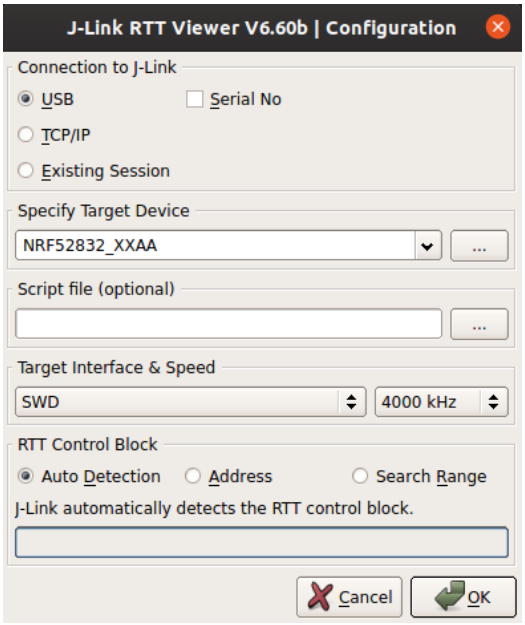
If you are using RTT log, you can also press Button 1 on the servers to locally toggle the state of their LED 1, and the status reflecting this state will be sent to the client board. You can see the status printed in the RTT log of the client board.

If any of the devices is powered off and back on, it will remember its flash configuration and rejoin the network. For more information about the flash manager, see Flash manager.

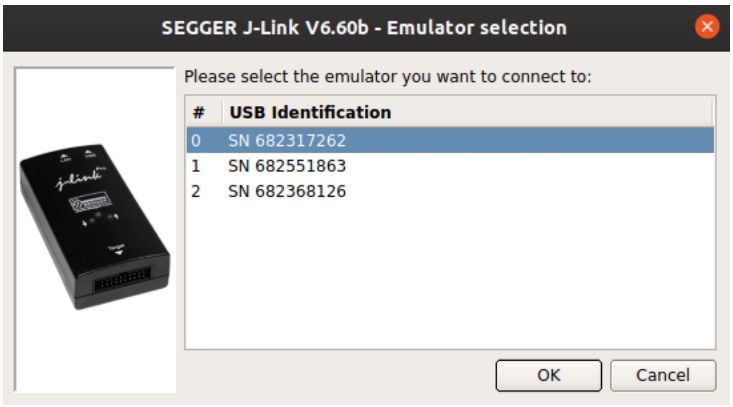
2.7 J-Link RTT Viewer

You can use J-Link RTT Viewer to debug the code. J-Link RTT Viewer is a GUI application to use all features of RTT in one application on the debugging host (Windows, macOS and Linux).

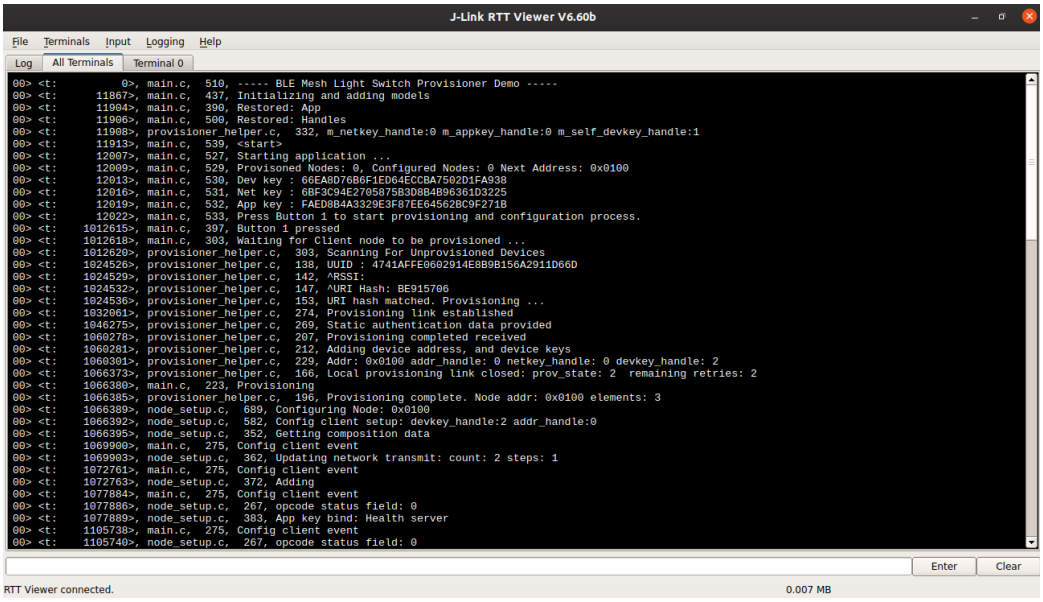
When you open the application the main windows is showed below. It's possible to select the device by the Specify Target Device option.



The three board used for the project have the following serial codes as shown in the figure below



An example of use registered with the card acting as a provisioner is now presented. You can see that when button 1 is selected, provisioning begins. The client is assigned and assigned 0x0100.


J-Link RTT Viewer V6.60b
File Terminals Input Logging Help
Log All Terminals Terminal 0
00> <t: 0>, main.c, 510, ----- BLE Mesh Light Switch Provisioner Demo -----
00> <t: 11867>, main.c, 437, Initializing and adding models
00> <t: 11904>, main.c, 390, Restored: App
00> <t: 11906>, main.c, 500, Restored: Handles
00> <t: 11908>, provisioner_helper.c, 332, m_netkey_handle:0 m_appkey_handle:0 m_self_devkey_handle:1
00> <t: 11913>, main.c, 539, <start>
00> <t: 12007>, main.c, 527, Starting application ...
00> <t: 12009>, main.c, 529, Provisoned Nodes: 0, Configured Nodes: 0 Next Address: 0x0100
00> <t: 12013>, main.c, 530, Dev key : 66EA8D76B6F1ED64ECCBA7502D1FA938
00> <t: 12016>, main.c, 531, Net key : 6BF3C94E27058758D0B4B96361D3225
00> <t: 12019>, main.c, 532, App key : FAED0B4A3329E3F87EE64562BC9F271B
00> <t: 12022>, main.c, 533, Press Button 1 to start provisioning and configuration process.
00> <t: 1012615>, main.c, 397, Button 1 pressed
00> <t: 1012618>, main.c, 303, Waiting for Client node to be provisioned ...
00> <t: 1012620>, provisioner_helper.c, 303, Scanning For Unprovisioned Devices
00> <t: 1024526>, provisioner_helper.c, 138, UUID : 4741AFFE0602914E8B9B156A2911D66D
00> <t: 1024529>, provisioner_helper.c, 142, ^RSSI:
00> <t: 1024532>, provisioner_helper.c, 147, ^URI Hash: BE915706
00> <t: 1024536>, provisioner_helper.c, 153, URI hash matched. Provisioning ...
00> <t: 1032061>, provisioner_helper.c, 274, Provisioning link established
00> <t: 1046275>, provisioner_helper.c, 269, Static authentication data provided
00> <t: 1060278>, provisioner_helper.c, 207, Provisioning completed received
00> <t: 1060281>, provisioner_helper.c, 212, Adding device address, and device keys
00> <t: 1060301>, provisioner_helper.c, 229, Addr: 0x0100 addr_handle: 0 netkey_handle: 0 devkey_handle: 2
00> <t: 1066373>, provisioner_helper.c, 166, Local provisioning link closed: prov_state: 2 remaining retries: 2
00> <t: 1066380>, main.c, 223, Provisioning
00> <t: 1066385>, provisioner_helper.c, 196, Provisioning complete. Node addr: 0x0100 elements: 3
00> <t: 1066389>, node_setup.c, 689, Configuring Node: 0x0100
00> <t: 1066392>, node_setup.c, 582, Config client setup: devkey_handle:2 addr_handle:0
00> <t: 1066395>, node_setup.c, 352, Getting composition data
00> <t: 1069900>, main.c, 275, Config client event
00> <t: 1069903>, node_setup.c, 362, Updating network transmit: count: 2 steps: 1
00> <t: 1072761>, main.c, 275, Config client event
00> <t: 1072763>, node_setup.c, 372, Adding
00> <t: 1077884>, main.c, 275, Config client event
00> <t: 1077886>, node_setup.c, 267, opcode status field: 0
00> <t: 1077889>, node_setup.c, 383, App key bind: Health server
00> <t: 1105738>, main.c, 275, Config client event
00> <t: 1105740>, node_setup.c, 267, opcode status field: 0
Enter
Clear
RTT Viewer connected.
0.007 MB

3 Eddystone Beacon Application

Important: Before you run this example, make sure to program the SoftDevice.

Compiling the example for the first time, the following error appears: micro_ecc_lib_nrf52.lib: No such file or directory. It's possible to see in the following link micro_ecc backend that to solve this error you need to:

1. Install latest version of the GCC compiler toolchain for ARM. You can use ARM's Launchpad to find the toolchain for your operating system.

2. Make sure that make is installed (see, for example, MinGW for Windows or GNU Make for Linux).

3. Clone the micro-ecc GitHub repository into InstallFolder\external\micro-ecc\micro-ecc.

4. Enter the subdirectory for the SoC and the toolchain that you are using to build your application:

   - InstallFolder\external\micro-ecc\nrf52_keil\armgcc
   - InstallFolder\external\micro-ecc\nrf52_iar\armgcc
   - InstallFolder\external\micro-ecc\nrf52_armgcc\armgcc (come nel progetto)

5. Modificare nel file Makefile.posix nel percorso InstallFolder\components\toolchain\gcc la riga GNU_INSTALL_ROOT con il proprio percorso di installazione della toolchain arm, come ad esempio \home\gcc-arm-none-eabi-9-2019-q4-major\bin\

6. Run make to compile the micro-ecc library.

The Eddystone Beacon Application example can be used to turn your nRF5 Development Kit board into an Eddystone Beacon. The application is intended to be used without modifications. When it is running, the beacon can be configured through the Eddystone Configuration Service. The Fig. 26 shows the UUID of the Eddystone Configuration Service.

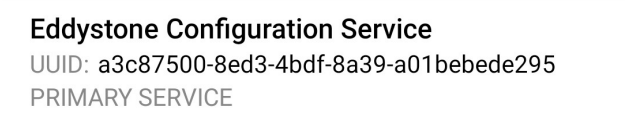


Figure 26: Eddystone Configuration Service

3.1 Eddystone protocol

Eddystone is an open beacon format designed by Google that defines a Bluetooth low energy (BLE) message format for proximity beacon messages. See the Eddystone GitHub repository for the protocol specification, the Eddystone Configuration Service specification, and further information. The example application supports all features defined in the specifications except for per-slot variable advertising intervals.

The Eddystone protocol describes different formats for advertising packets, called frame types, that can be used to create beacons. The following frame types are available:

- **Eddystone-UID** for broadcasting unique 16-byte beacon IDs
- **Eddystone-URL** for broadcasting a URL in a compressed encoding format
- **Eddystone-TLM** for broadcasting telemetry information about the beacon (encrypted or unencrypted)
- **Eddystone-EID** for broadcasting an encrypted ephemeral identifier that changes periodically

The example provides five advertising slots, and the user can configure each slot to use any of these frame types. There must not be more than one slot of the Eddystone-TLM frame type, but all other frame types can be used in more than one slots. The slot configuration is done through the Eddystone Configuration Service.

3.2 Security features

While the Eddystone-UID frame type is broadcast unprotected, the Eddystone-EID frame type provides protection against known beacon vulnerabilities.

The Eddystone-EID frame type randomizes the device ID of the beacon, as well as the encrypted advertising data. This process protects against spoofing and malicious asset tracking, because the encrypted and changing advertising data makes it difficult to fake or track an Eddystone-EID.

If a beacon is configured to broadcast the Eddystone-EID frame type, the Eddystone-TLM frame type is automatically encrypted. Otherwise, the transmitted information would make the device uniquely identifiable. Using encrypted TLM frames also ensures message integrity, which means that you can rely on the telemetry information being sent from the expected beacon. Encrypted Eddystone-TLM is often referred to as Eddystone-eTLM.

Note that you should never use an Eddystone-UID frame type together with Eddystone-EID, because the UID could be used to bypass the security protection.

3.3 Setup

You can find the source code and the project file of the example in the following folder: <InstallFolder>\examples\ble_peripheral\ble_app_eddystone

There is no need for you to modify the example code to test the beacon application, because most of the configuration is done through the Eddystone Configuration Service when the beacon is up and running. However, you can configure some basic settings in es_app_config.h. Before going into production, make sure to change the following settings:

- The **device name** (APP_DEVICE_NAME) that is advertised in the scan response when the beacon is in connectable mode. Default: "nRF5x_Eddystone".
- The **default frame type** (DEFAULT_FRAME_TYPE) that is used for all five advertising slots before the beacon is configured. Default: Eddystone-URL.
- The **lock code** (APP_CONFIG_LOCK_CODE) that is required to configure the beacon. Default: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF (32 F characters).

3.4 Testing

To test the Eddystone Beacon Application, you must install the nRF Beacon for Eddystone Android app. It is available from Google Play and the nRF Beacon for Eddystone GitHub repository.

Test the Eddystone Beacon Application by performing the following steps:

1. Compile and program the application.
2. Observe that LED 1 on the board starts blinking and that the board starts broadcasting an Eddystone-URL. You can read the URL with any beacon scanner that is compatible with Eddystone.
3. Press Button 1 on the board to put the board (resp. beacon) in connectable mode for 60 seconds. Observe that LED 3 is turned on.
4. Open the nRF Beacon for Eddystone app, then tap the connect button on the update tab. The list of connectable Eddystone beacons is displayed.

5. Select your device. By default, it is advertising as "nRF5x_Eddystone".

6. Enter the beacon manufacturer lock code. By default, this code is FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF (32 F characters).

7. Observe that after entering the correct lock code, the application reads through all slots and displays the information for each slot. Also observe that LED 2 is turned on.

8. Test the supported features of the beacon. For example, configure all slots as shown in figure 27 .

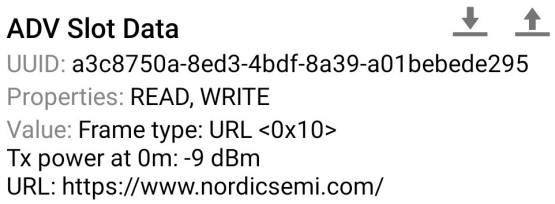


Figure 27: ADV Slot Data

Then perform a factory reset and ensure that the beacon has been reset to its default state.

9. Test disabling the automatic relocking capability:

   (a) In the app, select **Lock state** and set it to **Disable automatic relock**.

   (b) Disconnect from the device and repeat steps 3 to 5. Observe that you can now unlock the beacon without entering a lock code.

4 Beacon Transmitter Sample Application

The Beacon Transmitter Sample Application is an example that implements a transmitter beacon using the hardware delivered in the nRF5 Development Kit.

The beacon broadcasts information to all compatible devices in its range as Manufacturer Specific Data in the advertisement packets.

This information includes:

- A 128-bit UUID to identify the beacon's provider.
- An arbitrary Major value for coarse differentiation between beacons.
- An arbitrary Minor value for fine differentiation between beacons.
- The RSSI value of the beacon measured at 1 meter distance, which can be used for estimating the distance from the beacon.

Note

This application is not power optimized!

You can find the source code and the project file of the example in the following folder: <InstallFolder>\examples\ble_peripheral\ble_app_beacon

4.1 Configuring Major and Minor values

This example implements an optional feature which allows the change of Major and Minor values used in the advertisement packets without the need to recompile the application each time. To use this feature, the compiler define USE_UICR_FOR_MAJ_MIN_VALUES should be defined during compilation. If this is done, the application uses the value of the UICR (User Information Configuration Register) located at address 0x10001080 to populate the major and minor values. Whenever the values need to be updated, the user must set the UICR to a desired value using the nrfjprog tool and restart the application to bring the changes into effect. The byte ordering used when decoding the UICR value is shown in the image below.

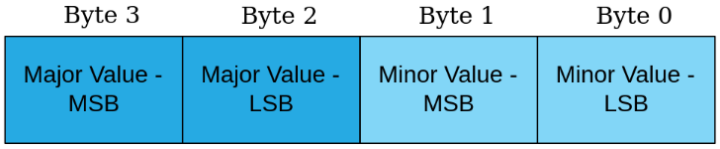


Figure 28: Byte ordering used while reading UICR

See Testing for more information about how to use this feature.

Note

This application can be used as a starting point to write an iBeacon application. The procedure for creating an iBeacon application and the specification should be available at mfi.apple.com. Once the specification is obtained, this application can be modified to fit the requirements of iBeacon.

4.2 Testing

Test the Beacon Transmitter Sample Application with nRF Connect by performing the following steps:

1. Compile and program the application. Observe that the BSP_INDICATE_ADVERTISING state is indicated.

2. After starting discovery in nRF Connect, observe that the beacon is advertising with its Bluetooth device address without a Device Name.

3. Click on Details under the beacon's address to view the full advertisement data.

4. Observe that the **Advertising Type** is 'Non-connectable' and the **Manufacturer Specific Data** field of the Advertising Data is as follows:
   59-00-02-15-01-12-23-34-45-56-67-78-89-9A-AB-BC-CD-DE-EF-F0-01-02-03-04-C3

1. Define the compiler define USE_UICR_FOR_MAJ_MIN_VALUES and recompile and flash.

2. Use the nrfjprog tool to write the value 0xabcd0102 to the UICR register as follows:

   ```
   nrfjprog -f nrf52 –snr <Segger-chip-Serial-Number> –memwr 0x10001080 –val 0xabcd0102
   ```

3. Reset the board and observe the bytes in bold below are seen in the **Manufacturer Specific Data** field of the Advertising Data. This indicates that the Major and Minor values have been picked up from the UICR register written in the above step.
   xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-xx-**AB-CD-01-02**-xx

5 Beaconing Example

This example shows how to do concurrent beaconing, which allows an application to advertise beacons (such as iBeacon or Eddystone beacons) while participating in the mesh network. Moreover, it demonstrates the usage of the RX callback.

5.1 Beacon Sending

To send beacons, the application uses the mesh-internal packet manager and advertiser structure directly.

The application first initializes the advertiser (initialization is needed only once). It then allocates and fills the fields of the packet. There is no need to set the packet type or advertisement address, because this is taken care of by the advertiser module.

After the initialization, the application schedules the packet for transmission by putting it in the TX queue of the advertiser, with a parameter indicating the number of repeats that the advertiser will do. In this example, the repeat count is set to BEARER_ADV_REPEAT_INFINITE, which causes the packet to be retransmitted forever or until replaced by a different packet.

5.2 RX Callback

The beaconing example also demonstrates the usage of the Packet RX callback. This functionality allows to receive all non-filtered, BLE-compliant advertisement packets in the application code. These packets are captured by Scanner.

The ble_packet_type_t type lists all advertisement packets that it is possible to receive. Once a new packet is captured by the Scanner, it is passed through the provided RX callback to the user application.

To listen to advertisement packets, the example registers the RX callback by calling nrf_mesh_rx_cb_set(). As input, the RX callback function takes a pointer to a parameter struct that contains all data available on the incoming packet.

The RX callback is invoked for all packets that are processed by the mesh after the mesh itself has processed them. The mesh assumes that all incoming packets adhere to the Bluetooth Low Energy advertisement packet format.

5.3 Software requirements

You need to install nRF Connect in one of the following versions:

- nRF Connect for Desktop
- nRF Connect for Mobile

5.4 Setup

You can find the source code of the beaconing example in the following folder: <InstallFolder>\examples\beaconing

5.5 Testing the example

To test the beaconing example:

1. Build the example by following the instructions in Building the mesh stack and examples
2. Program the board by following the instructions in Running examples using SEGGER Embedded Studio.
3. Connect RTT viewer to see the RTT output generated by the beaconing example in the RTT log.

Once the example is running, it outputs all incoming packets over RTT. Outgoing beacons can be observed with nRF Connect for Desktop or nRF Connect for Mobile.

6 Implementazione di beacon statici nell'esempio Light Switch Server

Verranno ora illustrate le varie prove effettuate per aggiungere dei beacon statici, cioè che non possono essere cambiati una volta che la scheda è stata programmata, all'interno dell'esempio Light Switch Server.

6.1 Integrazione dell'esempio Beacon Transmitter nel Light Switch Server

Nel file main.c dell'esempio Light Switch Server è stata aggiunta la funzione **advertising_init()** presa dall'applicazione Beacon Transmitter descritta nel capitolo 4 e definita come di seguito:

```
static void advertising_init(void)
{
  uint32_t       err_code;
  ble_advdata_t advdata;
  uint8_t        flags =
     BLE_GAP_ADV_FLAG_BR_EDR_NOT_SUPPORTED;

  ble_advdata_manuf_data_t manuf_specific_data;

  manuf_specific_data.company_identifier =
     APP_COMPANY_IDENTIFIER;

  manuf_specific_data.data.p_data = (uint8_t *)
     m_beacon_info;
  manuf_specific_data.data.size    = APP_BEACON_INFO_LENGTH;

  // Build and set advertising data.
  memset(&advdata, 0, sizeof(advdata));

  advdata.name_type               = BLE_ADVDATA_NO_NAME;
  advdata.flags                   = flags;
  advdata.p_manuf_specific_data = &manuf_specific_data;

  // Initialize advertising parameters (used when starting
     advertising).
  memset(&m_adv_params, 0, sizeof(m_adv_params));

  m_adv_params.properties.type =
     BLE_GAP_ADV_TYPE_NONCONNECTABLE_NONSCANNABLE_UNDIRECTED
     ;
  m_adv_params.p_peer_addr      = NULL;     // Undirected
     advertisement.
  m_adv_params.filter_policy    = BLE_GAP_ADV_FP_ANY;
  m_adv_params.interval         =
     NON_CONNECTABLE_ADV_INTERVAL;
  m_adv_params.duration         = 0;        // Never time out
     .

  err_code = ble_advdata_encode(&advdata, m_adv_data.
     adv_data.p_data, &m_adv_data.adv_data.len);
  ERROR_CHECK(err_code);
```

```
    err_code = sd_ble_gap_adv_set_configure(&m_adv_handle, &
        m_adv_data, &m_adv_params);
    ERROR_CHECK(err_code);
}
```

This function initialize the Advertising functionality. Encodes the required advertising data and passes it to the stack using **ble_advdata_encode**. Also builds a structure to be passed to the stack when starting advertising using **sd_ble_gap_adv_set**.

Nella funzione **advertising_init()** si sono presentati i seguenti errori:

1. unknown type name 'ble_advdata_t'
2. unknown type name 'ble_advdata_manuf_data_t'
3. 'APP_COMPANY_IDENTIFIER' undeclared (first use in this function)
4. 'm_beacon_info' undeclared (first use in this function)
5. 'APP_BEACON_INFO_LENGTH' undeclared (first use in this function)
6. 'BLE_ADVDATA_NO_NAME' undeclared (first use in this function)
7. 'm_adv_params' undeclared (first use in this function)
8. 'NON_CONNECTABLE_ADV_INTERVAL' undeclared (first use in this function)
9. 'm_adv_data' undeclared (first use in this function)
10. 'm_adv_handle' undeclared (first use in this function)

Gli errori 1 e 2 producono ulteriori errori sulle variabili advdata e manuf_specific_data in quanto le struct 'ble_advdata_t' e 'ble_advdata_manuf_data_t' non sono state dichiarate.
Per risolvere gli errori 1, 2 e 6 è stata aggiunta la libreria "ble_advdata.h" nel main:

```
#include "ble_advdata.h"
```

in questo modo si eliminano anche gli errori relativi alle variabili advdata e manuf_specific_data.
Per le costanti 3, 5 e 8 è stato aggiunto nel main:

```
#define NON_CONNECTABLE_ADV_INTERVAL MSEC_TO_UNITS(100,
    UNIT_0_625_MS) /**< The advertising interval for non-
    connectable advertisement (100 ms). This value can vary
    between 100ms to 10.24s). */

#define APP_BEACON_INFO_LENGTH 0x17 /**< Total length of
    information advertised by the Beacon. */
#define APP_COMPANY_IDENTIFIER 0x0059 /**< Company
    identifier for Nordic Semiconductor ASA. as per www.
    bluetooth.org. */
```

Per risolvere l'errore 4 è stato dichiarato nel main il vettore che costituisce il payload del beacon. I commenti al codice sono abbastanza esplicativi sulla struttura del dato.

```
static uint8_t m_beacon_info[APP_BEACON_INFO_LENGTH] =
                    /**< Information advertised by the
    Beacon. */
{
  APP_DEVICE_TYPE,      // Manufacturer specific information
      . Specifies the device type in this
  // implementation.
  APP_ADV_DATA_LENGTH, // Manufacturer specific information
      . Specifies the length of the
  // manufacturer specific data in this implementation.
  APP_BEACON_UUID,      // 128 bit UUID value.
  APP_MAJOR_VALUE,      // Major arbitrary value that can be
       used to distinguish between Beacons.
  APP_MINOR_VALUE,      // Minor arbitrary value that can be
       used to distinguish between Beacons.
  APP_MEASURED_RSSI     // Manufacturer specific information
      . The Beacon's measured TX power in
  // this implementation.
};
```

le cui costanti sono

```
#define APP_ADV_DATA_LENGTH   0x15 /**< Length of
    manufacturer specific data in the advertisement. */
#define APP_DEVICE_TYPE   0x02 /**< 0x02 refers to Beacon.
     */
#define APP_MEASURED_RSSI 0xC3 /**< The Beacon's measured
    RSSI at 1 meter distance in dBm. */
#define APP_MAJOR_VALUE   0x01, 0x02 /**< Major value used
```

```
    to identify Beacons. */
#define APP_MINOR_VALUE    0x03,    0x04   /**< Minor value
    used to identify Beacons. */
#define APP_BEACON_UUID 0x01, 0x12, 0x23, 0x34, \
  0x45, 0x56, 0x67, 0x78, \
  0x89, 0x9a, 0xab, 0xbc, \
  0xcd, 0xde, 0xef, 0xf0            /**< Proprietary UUID
     for Beacon. */
```

Per risolvere l'errore 7 è stata dichiarata globalmente la variabile

```
static ble_gap_adv_params_t m_adv_params; /**< Parameters
    to be passed to the stack when starting advertising. */
```

L'errore 9 è stato risolto dichiarando il buffer che contiene l'advertising codificato e la struttura che contiene il puntato al dato di advertising codificato.

```
static uint8_t                m_enc_advdata[
    BLE_GAP_ADV_SET_DATA_SIZE_MAX];  /**< Buffer for storing
     an encoded advertising set. */

/**@brief Struct that contains pointers to the encoded
    advertising data. */
static ble_gap_adv_data_t m_adv_data =
{
  .adv_data =
  {
    .p_data = m_enc_advdata,
    .len    = BLE_GAP_ADV_SET_DATA_SIZE_MAX
  },
  .scan_rsp_data =
  {
    .p_data = NULL,
    .len    = 0

  }
};
```

Per 10 è stato dichiarato

```
static uint8_t  m_adv_handle =
    BLE_GAP_ADV_SET_HANDLE_NOT_SET;  /**< Advertising handle
     used to identify an advertising set. */
```

Listing 1: Declaration of m_adv_handle

Successivamente viene aggiunta la funzione **advertising_start()** for starting advertising.

```
static void advertising_start(void)
{
  ret_code_t err_code;

  err_code = sd_ble_gap_adv_start(m_adv_handle,
     APP_BLE_CONN_CFG_TAG);
  ERROR_CHECK(err_code);

  err_code = bsp_indication_set(BSP_INDICATE_ADVERTISING);
  ERROR_CHECK(err_code);
}
```

Nella funzione **advertising_start()** si sono verificati i seguenti errori:

1. 'm_adv_handle' undeclared (first use in this function)
2. 'APP_BLE_CONN_CFG_TAG' undeclared (first use in this function)
3. 'BSP_INDICATE_ADVERTISING' undeclared (first use in this function)
4. undefined reference to 'bsp_indication_set'

L'errore 1 è stato risolto come riportato nel codice 1.
Per 2 è stata scritta la costante

```
#define APP_BLE_CONN_CFG_TAG  1 /**< A tag identifying the
    SoftDevice BLE configuration. */
```

Per 3 è stata inclusa la libreria "bsp.h"

```
#include "bsp.h"
```

Per il corretto funzionamento delle librerie occorre settare manualmente il percorso delle librerie stesse cliccando nell'IDE Segger **Project** → **Options** → **Preprocessor** → **User Include Directories** ed aggiungendo:
../components/libraries/bsp
../components/libraries/button
Per l'errore 4 occorre aggiungere il file sorgente "bsp.c" nel progetto. Per fare questo è stata creata la cartella "Board Support" nel progetto e vi è stato inserito il file sorgente. Il file "bsp.c" necessita anche di "boards.c" presente

in .../nRF5_SDK_16.0.0_98a08e2/components/boards per il corretto funzionamento. Per far partire il beaconing sono state richiamate le funzioni **advertising_init()** e **advertising_start()** all'interno del **main**().

```
int main(void)
{
  initialize();
  start();
  advertising_init();
  advertising_start();
  for (;;)
  {
    (void)sd_app_evt_wait();
  }
}
```

Con queste modifiche si ottiene l'errore Mesh error 4 at 0x00026901 causato dalla funzione **mesh_provisionee_prov_start** contenuta in **start**(). L'errore è stato risolto disabilitando il proxy: nel codice è stata portata a zero la macro MESH_FEATURE_GATT_ENABLED che non fa compilare la seguente parte di codice

```
#if MESH_FEATURE_GATT_ENABLED
  gap_params_init();
  conn_params_init();
#endif
```

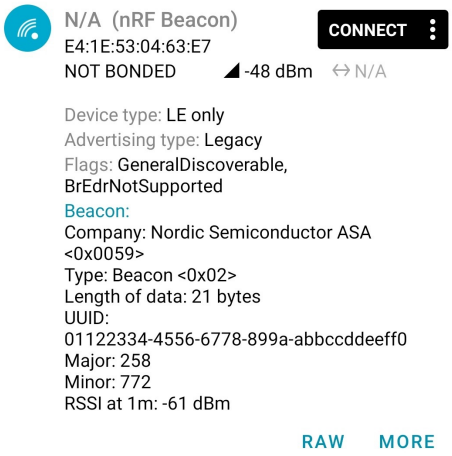
MESH_FEATURE_GATT_ENABLED è definita nella libreria "nrf_mesh_gatt.h" come

```
#define MESH_FEATURE_GATT_ENABLED (
    MESH_FEATURE_GATT_PROXY_ENABLED ||
    MESH_FEATURE_PB_GATT_ENABLED)
```

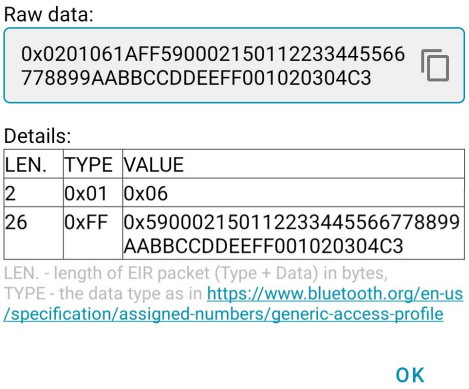
quindi sono state portate a zero le due macro MESH_FEATURE_GATT_PROXY_ENABLED e MESH_FEATURE_PB_GATT_ENABLED nella libreria "nrf_mesh_config_app.h".

```
/** PB-GATT feature. To be enabled only in combination with
    linking GATT files. */
#define MESH_FEATURE_PB_GATT_ENABLED                    (0)
/** GATT proxy feature. To be enabled only in combination
   with linking GATT proxy files. */
#define MESH_FEATURE_GATT_PROXY_ENABLED                 (0)
```

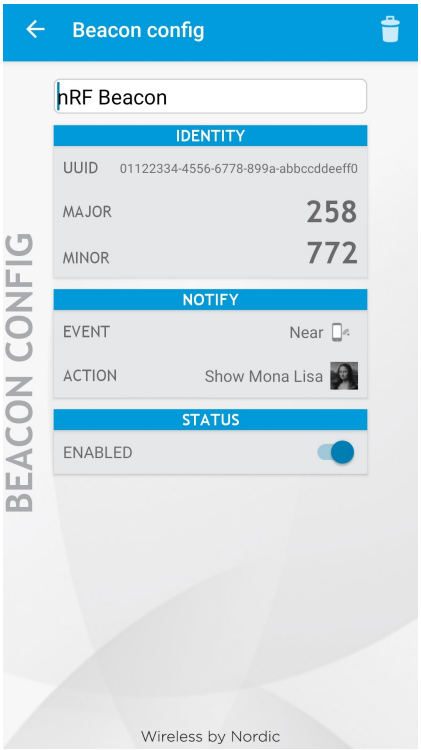
La disabilitazione del proxy non permette agli smartphone di partecipare alla rete mesh, ma possono comunque ricevere beacon come mostrato nelle figure 29a, 29b, e 29c.



(a) iBeacon



(b) Raw Data



(c) iBeacon nell'app nRF Beacon

Figure 29

6.2 Integrazione dell'esempio Mesh Beaconing nel Light Switch Server

Per risolvere il problema del proxy gatt incontrato nel capitolo precedente, ho usato le funzioni presenti nell'esempio Beaconing nell'SDK Mesh 4.0 descritto nel capitolo 5.

6.2.1 Implementazione di iBeacon

Nell'esempio Beaconing, nella funzione **adv_start**() viene definita una variabile chiamata adv_data che contiene i bit del payload del pacchetto di advertising da mandare.

Se nel campo Manufacturer ID del beacon si inserisce 0x004C si hanno beacon per dispositivi Apple. I valori di "Company Identifier" si possono trovare al seguente link Company Identifiers.

La variabile adv_data è stata modificata come di seguito per mandare iBeacon.

```
static const uint8_t adv_data[] =
{
  APP_FLAG_BEACON_LENGTH, /* Flag Beacon length (including
     type, but not itself) */
  BLE_GAP_AD_TYPE_FLAGS, /* Flags for discoverability. */
  BLE_GAP_ADV_TYPE_EXTENDED_CONNECTABLE_NONSCANNABLE_UNDIRECTED
     , /**< Connectable non-scannable undirected
     advertising events using extended advertising PDUs. */
  APP_ADVERTISER_LENGTH, /* Advertiser data length (
     including type, but not itself) */
  BLE_GAP_AD_TYPE_MANUFACTURER_SPECIFIC_DATA, /*
     Manufacturer Specific Data. */
  APP_COMPANY_IDENTIFIER, /* Company ID */
  APP_iBEACON_TYPE, /* iBeacon Type */
  APP_iBEACON_LENGTH, /* iBeacon Lenght */
  APP_BEACON_UUID,  /* UUID */
  APP_MAJOR_VALUE, /* Major Value*/
  APP_MINOR_VALUE, /* Minor Value */
  APP_MEASURED_RSSI  /* RSSI at 1 m */
};
```

Listing 2: Configurazione della variabile adv_data per iBeacon

avendo dichiarato nel main i seguenti valori costanti

```
#define APP_FLAG_BEACON_LENGTH  0x02  /*< Length of the
   discoverability Beacon. */
```

```
#define APP_ADVERTISER_LENGTH 0x1A  /* Total length of the
    iBeacon. */
#define APP_iBEACON_LENGTH  0x15  /* Length of manufacturer
     specific data in the advertisement. */
#define APP_iBEACON_TYPE  0x02  /**< iBeacon Type. */
#define APP_MEASURED_RSSI 0xC3  /* The Beacon's measured
    RSSI at 1 meter distance in dBm. */
#define APP_COMPANY_IDENTIFIER  0x4C, 0x00  /* Company
    identifier for Apple Inc. as per www.bluetooth.org. */
#define APP_MAJOR_VALUE 0x01, 0x02  /* Major value used to
    identify Beacons. */
#define APP_MINOR_VALUE 0x03, 0x04  /* Minor value used to
    identify Beacons. */
#define APP_BEACON_UUID 0x01, 0x12, 0x23, 0x34, \
                        0x45, 0x56, 0x67, 0x78, \
                        0x89, 0x9A, 0xAB, 0xBC, \
                        0xCD, 0xDE, 0xEF, 0xF0            /*<
                           Proprietary UUID for Beacon. */
```

Listing 3: Costanti pacchetto iBeacon

ed avendo usato i valori nella libreria "BLE_GAP.h":

- "BLE_GAP_AD_TYPE_FLAGS" = 0x01
- "BLE_GAP_ADV_TYPE_EXTENDED_CONNECTABLE_NONSCANNABLE_UND = 0x06
- "BLE_GAP_AD_TYPE_MANUFACTURER_SPECIFIC_DATA" = 0xFF.

Per trasmettere i beacon, nel firmware Light Switch Server sono state aggiunte delle funzioni dell'esempio Beaconing presente nel SDK Mesh. Nella funzione **mesh_init** è stata aggiunta **nrf_mesh_rx_cb_set** definita come di seguito. La funzione di callback è stata già descritta nel capitolo 5.2.

```
void nrf_mesh_rx_cb_set(nrf_mesh_rx_cb_t rx_cb)
{
  m_rx_cb = rx_cb;
}
```

Aggiungendo questa funzione si ottiene l'errore 'rx_cb' undeclared (first use in this function) risolto aggiungendo nel main il seguente codice

```
static void rx_cb(const nrf_mesh_adv_packet_rx_data_t *
    p_rx_data)
```

```
{
  LEDS_OFF(BSP_LED_0_MASK);   /* @c LED_RGB_RED_MASK on
      pca10031 */
  char msg[128];
  (void) sprintf(msg, "RX [@%u]: RSSI: %3d ADV TYPE: %x
     ADDR: [%02x:%02x:%02x:%02x:%02x:%02x]",
  p_rx_data->p_metadata->params.scanner.timestamp,
  p_rx_data->p_metadata->params.scanner.rssi,
  p_rx_data->adv_type,
  p_rx_data->p_metadata->params.scanner.adv_addr.addr[0],
  p_rx_data->p_metadata->params.scanner.adv_addr.addr[1],
  p_rx_data->p_metadata->params.scanner.adv_addr.addr[2],
  p_rx_data->p_metadata->params.scanner.adv_addr.addr[3],
  p_rx_data->p_metadata->params.scanner.adv_addr.addr[4],
  p_rx_data->p_metadata->params.scanner.adv_addr.addr[5]);
  __LOG_XB(LOG_SRC_APP, LOG_LEVEL_INFO, msg, p_rx_data->
      p_payload, p_rx_data->length);
  LEDS_ON(BSP_LED_0_MASK);   /* @c LED_RGB_RED_MASK on
      pca10031 */
}
```

Si ottengono i seguenti errori

1. 'm_advertiser' undeclared (first use in this function)
2. 'm_adv_buffer' undeclared (first use in this function)
3. 'ADVERTISER_BUFFER_SIZE' undeclared (first use in this function)

L'errore 1 è stato risolto aggiungendo nel main

```
static advertiser_t m_advertiser;
```

e la libreria "advertiser.h" che definisce il tipo di variabile advertiser_t.
L'errore 2 è stato risolto aggiungendo nel main

```
static uint8_t        m_adv_buffer[ADVERTISER_BUFFER_SIZE];
```

L'errore 3 è stato risolto aggiungendo nel main

```
#define ADVERTISER_BUFFER_SIZE   (64)
```

Sempre nella funzione **mesh_init** è stata aggiunta **adv_init**() definita come di seguito.

```
static void adv_init(void)
{
  advertiser_instance_init(&m_advertiser, NULL,
      m_adv_buffer, ADVERTISER_BUFFER_SIZE);
}
```

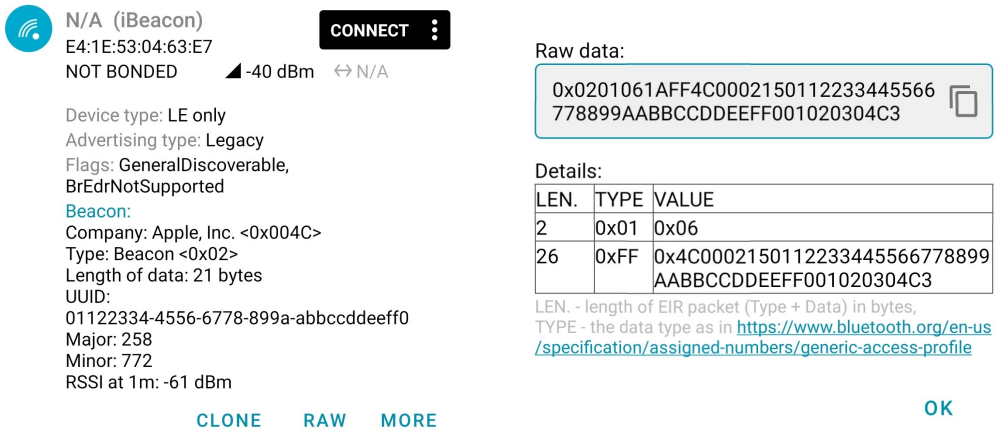
The function **advertiser_instance_init** can be called multiple times to initialize different advertiser.

Nella funzione **start** è stata aggiunta **adv_start** definita come di seguito.

```
static void adv_start(void)
{
  bearer_adtype_add(BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME);

  advertiser_enable(&m_advertiser);

  static const uint8_t adv_data[] =
  {
    APP_FLAG_BEACON_LENGTH, /* Flag Beacon length (
        including type, but not itself) */
    BLE_GAP_AD_TYPE_FLAGS, /**< Flags for discoverability.
        */
    BLE_GAP_ADV_TYPE_EXTENDED_CONNECTABLE_NONSCANNABLE_UNDIRECTED
        , /**< Connectable non-scannable undirected
        advertising
     events using extended advertising PDUs. */
    APP_ADVERTISER_LENGTH, /* Advertiser data length (
        including type, but not itself) */
    BLE_GAP_AD_TYPE_MANUFACTURER_SPECIFIC_DATA, /**<
        Manufacturer Specific Data. */
    APP_COMPANY_IDENTIFIER, /* Company ID */
    APP_iBEACON_TYPE, /* iBeacon Type */
    APP_iBEACON_LENGTH, /* iBeacon Lenght */
    APP_BEACON_UUID,   /* UUID */
    APP_MAJOR_VALUE, /* Major Value*/
    APP_MINOR_VALUE, /* Minor Value */
    APP_MEASURED_RSSI  /* RSSI at 1 m */
  };

    /* Allocate packet */
  adv_packet_t * p_packet = advertiser_packet_alloc(&
      m_advertiser, sizeof(adv_data));
  if (p_packet)
```

```
  {
    /* Construct packet contents */
    memcpy(p_packet->packet.payload, adv_data, sizeof(
        adv_data));
    /* Repeat forever */
    p_packet->config.repeats = ADVERTISER_REPEAT_INFINITE;

    advertiser_packet_send(&m_advertiser, p_packet);
  }
}
```

avendo definito nel main le macro come nel codice 3. The function **bearer_adtype_add** add the AD type BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME to the list of accepted AD types, while **advertiser_enable** enables the advertiser instance given. The functions **advertiser_packet_alloc** and **advertiser_packet_send** allocates a buffer, if available, from the advertiser instance m_advertiser and sends a packet p_packet using the advertiser m_advertiser, respectively. Il risultato è quello che si può vedere in figura 30a e 30b.



(a) iBeacon Apple (b) Raw Data

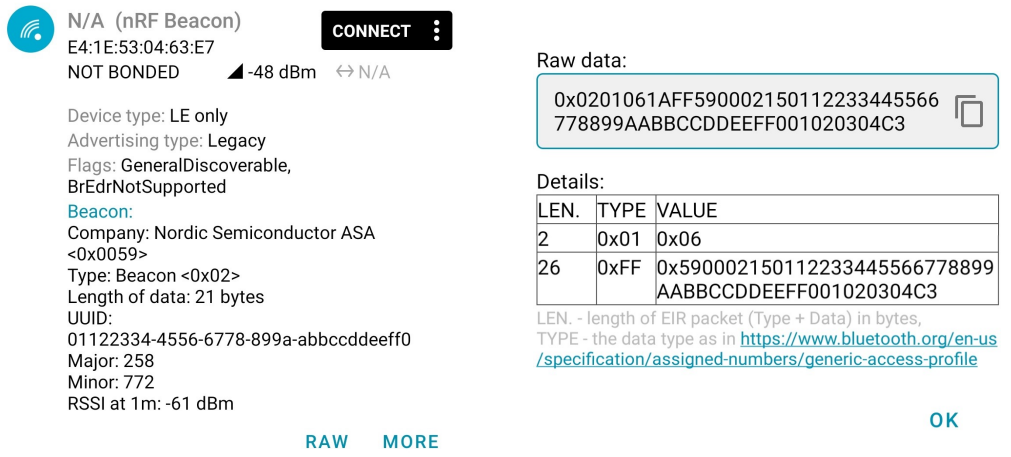
Figure 30: Beacon ricevuto dall'app nRF Connect

Se la macro APP_COMPANY_IDENTIFIER viene cambiata in 0x0059 nel campo Manufacturer ID, allora il produttore è Nordic Semiconductor.

```
#define APP_COMPANY_IDENTIFIER  0x59, 0x00  /* Company
    identifier for Nordic Semiconductor ASA. as per www.
    bluetooth.org. */
```

Listing 4: Manufacturer ID Nordic Semiconductor

I beacon trasmessi con Manufacturer ID pari a 0x0059 sono mostrati in figura 31a e 31b.



(a) iBeacon (b) Raw Data

Figure 31: Beacon ricevuto dall'app nRF Connect

6.2.2 Implementazione dei beacon Eddystone

Per costruire il beacon Eddystonè stata modificata la variabile adv_data precedentemente descritta nel seguente modo

```
static const uint8_t adv_data[] =
{
  APP_FLAG_BEACON_LENGTH, /* Flag Beacon length (including
     type, but not itself) */
  BLE_GAP_AD_TYPE_FLAGS, /**< Flags for discoverability. */
  BLE_GAP_ADV_TYPE_EXTENDED_CONNECTABLE_NONSCANNABLE_UNDIRECTED
     , /**< Connectable non-scannable undirected
     advertising events using extended advertising PDUs. */
  APP_SERVICE_UUID_LENGTH, /**< Server UUID advertising
     data length */
  BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE,   /**<
     Complete list of 16 bit service UUIDs. */
  APP_EDDYSTONE_UUID, /**< Eddystone UUID. */
  APP_EDDYSTONE_DATA_LENGTH, /**< Eddystone data length. */
  BLE_GAP_AD_TYPE_SERVICE_DATA, /**< Service Data - 16-bit
     UUID. */
  APP_EUUID, /**< Eddystone UUID. */
  APP_EDDYSTONE_URL_FRAME_TYPE, /**< Frame Type: Eddystone
     URL. */
```

```
  APP_MEASURED_RSSI, /**< Ranging data. */
  APP_URL_PREFIX, /**< URL prefix http://www. */
  'n', /**< URL. */
  'o',
  'r',
  'd',
  'i',
  'c',
  's',
  'e',
  'm',
  'i',
  '.',
  'c',
  'o',
  'm',
  '/'
};
```

Listing 5: Configurazione della variabile adv_data per beacon Eddystone

avendo definito nel main le seguenti macro

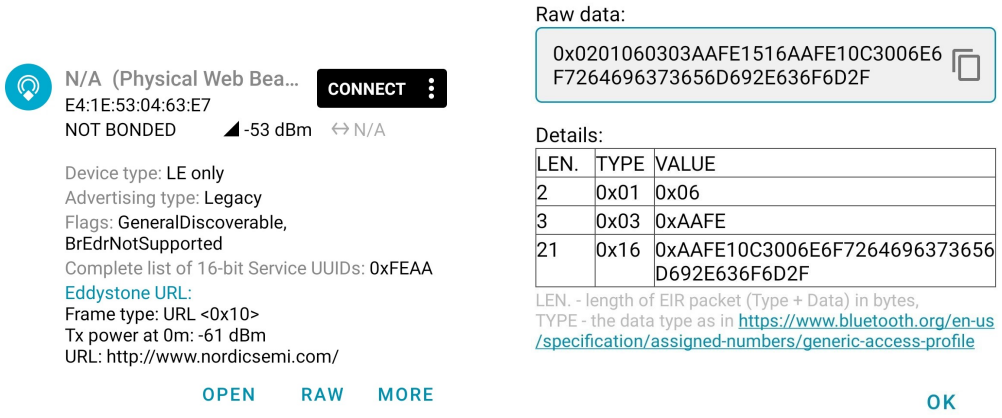
```
#define APP_SERVICE_UUID_LENGTH 0x03  /**< Complete list of
    UUID service advertising data length. */
#define APP_EDDYSTONE_DATA_LENGTH 0x15  /**< Eddystone
   Frame Length. */
#define APP_EDDYSTONE_UUID  0xAA, 0xFE  /**< Eddystone UUID
   . */
#define APP_EDDYSTONE_URL_FRAME_TYPE  0x10  /**< Eddystone
   URL Frame Type. */
#define APP_URL_PREFIX  0x00  /**< Eddystone URL Prefix. */
```

ed avendo usato i valori nella libreria "BLE_GAP.h":

- "BLE_GAP_AD_TYPE_FLAGS" = 0x01
- "BLE_GAP_ADV_TYPE_EXTENDED_CONNECTABLE_NONSCANNABLE_UND = 0x06
- "BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE" = 0x03
- "BLE_GAP_AD_TYPE_SERVICE_DATA" = 0x16.

Per implementare la trasmissione di beacon Eddystone nell'esempio Light Switch Server il procedimento è uguale a quello descritto sezione Implementazione di iBeacon con la variabile adv_data descritta sopra.

Il risultato è quello che si può vedere nelle due figure 32a e 32b.



(a) Eddystone Beacon (b) Raw Data

Figure 32: Eddystone Beacon ricevuto dall'app nRF Connect

6.3 Integrazione di iBeacon e beacon Eddystone in Light Switch Server

Per trasmettere sia gli iBeacon sia i beacon Eddystone sono state inserite due variabili chiamate iBeacon e eddystone, che hanno la stessa funzione di adv_data. Le due variabili sono state dichiarate rispettivamente come nei codici 2 e 5.

Con una sola istanza non è possibile mandare sia iBeacon sia Eddystone, quindi sono state dichiarate due istanze necessarie per trasmettere i due messaggi di advertising

```
static advertiser_t m_advertiser;
static advertiser_t m_advertiser2;
```

e due vettori che contengono i dati

```
static uint8_t       m_adv_buffer[ADVERTISER_BUFFER_SIZE];
/* */
static uint8_t       m_adv_buffer2[ADVERTISER_BUFFER_SIZE];
```

Nella funzione **adv_init**() è stato aggiunto

```
advertiser_instance_init(&m_advertiser2, NULL,
    m_adv_buffer2, ADVERTISER_BUFFER_SIZE);
```

mentre nella funzione **adv_start**() è stato aggiunto

```
advertiser_enable(&m_advertiser2);

adv_packet_t * p_packet_eddystone = advertiser_packet_alloc
    (&m_advertiser2, sizeof(eddystone));
if (p_packet_eddystone)
{
memcpy(p_packet_eddystone->packet.payload, eddystone,
    sizeof(eddystone));
p_packet_eddystone->config.repeats =
    ADVERTISER_REPEAT_INFINITE;
advertiser_packet_send(&m_advertiser2, p_packet_eddystone);
}
```

Il risultato finale che si può vedere nell'app nRF Connect è quello che si può osservare nell'animazione 33. [1]

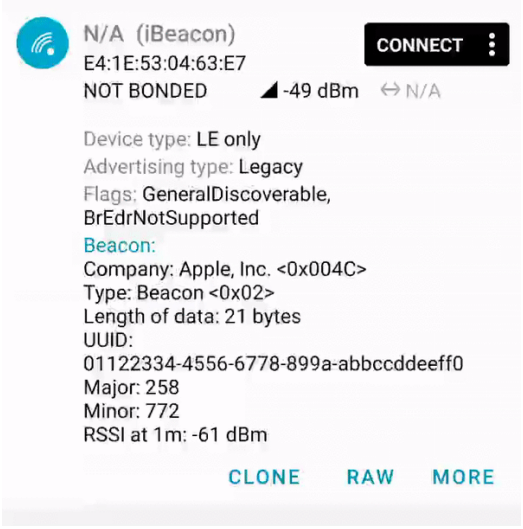


Figure 33: Beacon animation

[1] È possibile visualizzare correttamente l'animazione usando Adobe Acrobat Reader o un qualsiasi lettore PDF che abbia i plugin appropriati abilitati.

7 Integrating Mesh into nRF5 SDK examples

The nRF5 SDK for Mesh is compatible with Nordic's nRF5 SDK. This allows you to either include resources from nRF5 SDK in an existing mesh project or include nRF5 SDK for Mesh functionalities in an nRF5 SDK example.

7.1 Including nRF5 SDK for Mesh functionality in an nRF5 SDK example

1. Include the following source files from nRF5 SDK for Mesh in the nRF5 SDK example's project file:

    - All C files in mesh/core/src
    - All C files in mesh/bearer/src
    - All C files in mesh/prov/src except nrf_mesh_prov_bearer_gatt.c
    - All C files in mesh/access/src
    - All C files in mesh/dfu/src (not in this case)
    - All C files in mesh/stack/src
    - models/foundation/config/src/config_server.c
    - models/foundation/config/src/composition_data.c
    - models/foundation/config/src/packed_index_list.c
    - models/foundation/health/src/health_server.c
    - Any other mesh models that are used in your application (in this case Generic OnOff Model)
    - external/micro-ecc/uECC.c
    - examples/common/src/assertion_handler_weak.c
    - examples/common/src/mesh_provisionee.c
    - examples/common/src/app_onoff.c
    - examples/common/src/rtt_input.c
    - examples/common/src/simple_hal.c
    - examples/common/src/mesh_app_utils.c
    - examples/common/src/mesh_adv.c
    - examples/common/src/ble_softdevice_support.c
    - examples/nrf_mesh_weak.c

- config/sdk_config.h
- mesh/gatt/src/mesh_gatt.c
- mesh/gatt/src/proxy.c
- mesh/gatt/src/proxy_filter.c
- mesh/friend/src/friend.c
- mesh/friend/src/friend_queue.c
- mesh/friend/src/friend_sublist.c
- mesh/friend/src/core_tx_friend.c

Nota
If various mesh features are not needed (for example, DFU), the corresponding files may simply be omitted from the project file. However, add examples/nrf_mesh_weak.c in their place to provide stubs for the missing API functions.

2. Add the following folders to the project include path of the nRF5 SDK example:

   - mesh/core/api
   - mesh/core/include
   - mesh/bearer/api
   - mesh/bearer/include
   - mesh/prov/api ok
   - mesh/prov/include
   - mesh/access/api
   - mesh/access/include
   - mesh/dfu/api
   - mesh/dfu/include
   - mesh/stack/api
   - models/foundation/config/include
   - models/foundation/health/include
   - Path to include folder of any other mesh models that are used in your application, in this case models/model_spec/generic_onoff/include and models/model_spec/common/include
   - external/micro-ecc

- examples/common/include
- Path to any other resources in the mesh examples that are used in your application, like mesh/friend/api, mesh/friend/include, mesh/-gatt/api and mesh/gatt/include, examples/light_switch/include, examples/light_switch/server/include

3. Add the following preprocessor symbols to the project file of the nRF5 SDK example:

   - NRF52_SERIES
   - NRF_MESH_LOG_ENABLE=NRF_LOG_USES_RTT (because logging in the mesh stack relies on RTT)
   - CONFIG_APP_IN_CORE
   - NO_VTOR_CONFIG
   - USE_APP_CONFIG
   - NRF52832
   - NRF52832_XXAA
   - S132
   - SOFTDEVICE_PRESENT
   - NRF_SD_BLE_API_VERSION=7
   - BOARD_PCA10040
   - CONFIG_GPIO_AS_PINRESET

7.2 Integrazione delle funzionalità Mesh nell'esempio Eddystone

Dopo aver eseguito i passaggi riportati nel capitolo precedente all'esempio Eddystone, ho iniziato ad aggiungere nel main le funzioni che gestiscono la Mesh. In particolare **start**(), presa dall'esempio Beaconing e definita come di seguito, permette di configurare lo stack mesh.

```
static void start(void)
{
  rtt_input_enable(app_rtt_input_handler,
     RTT_INPUT_POLL_PERIOD_MS);

  if (!m_device_provisioned)
  {
```

```
    static const uint8_t static_auth_data[NRF_MESH_KEY_SIZE
        ] = STATIC_AUTH_DATA;
    mesh_provisionee_start_params_t prov_start_params =
    {
      .p_static_data     = static_auth_data,
      .prov_complete_cb = provisioning_complete_cb,
      .prov_device_identification_start_cb =
          device_identification_start_cb,
      .prov_device_identification_stop_cb = NULL,
      .prov_abort_cb = provisioning_aborted_cb,
      .p_device_uri = EX_URI_LS_SERVER
    };
    ERROR_CHECK(mesh_provisionee_prov_start(&
        prov_start_params));
  }

  mesh_app_uuid_print(nrf_mesh_configure_device_uuid_get())
      ;

  ERROR_CHECK(mesh_stack_start());

  hal_led_mask_set(LEDS_MASK, LED_MASK_STATE_OFF);
  hal_led_blink_ms(LEDS_MASK, LED_BLINK_INTERVAL_MS,
      LED_BLINK_CNT_START);
}
```

Dopo aver aggiunto questa funzione, si verificano i seguenti errori:

1. 'STATIC_AUTH_DATA' undeclared (first use in this function)
2. unknown type name 'mesh_provisionee_start_params_t'

L'errore 1 è stato risolto aggiungendo la libreria light_switch_example_common.h

```
#include "light_switch_example_common.h"
```

mentre per 2 ho aggiunto la libreria mesh_provisionee.h.

```
#include "mesh_provisionee.h"
```

Per il corretto funzionamento della funzione **start**() ho dovuto aggiungere il file nrf_mesh_prov_bearer_gatt.c contrariamente a quanto indicato in Including nRF5 SDK for Mesh functionality in an nRF5 SDK example. Questo porta ad un malfunzionamento del programma durante l'esecuzione. I problemi riscontrati sono risolti nel capitolo 8.

8 Coexistence between Eddystone Example and Light Switch Server

In chapter 7.1 it's not suggested to add nrf_mesh_prov_bearer_gatt.c because adding PB-GATT provisioning to a BLE application aren't as straight forward and requires a bit of work. This thread might help more on this topic. As you can see, it's not possible to use (normal) BLE services and GATT Bearer at the same time. I need to first make the device be part of a mesh network, then reset the device and in the end add the BLE services.

If you want run both Mesh and regular BLE at the same time, I suggest you have a look at the coexistence example from our SDK. You will get a better idea of how things can be done. The coexistence examples would be a good starting point if you want to run BLE and BLE Mesh concurrently. You will be able to use nRF Connect at least, you will only see your device in the nRF Mesh if you have PB-GATT or proxy feature enabled. If you want to do provisioning with your phone, you need to add PB-GATT provisioning to your application. This will require a bit of work.

8.1 Operazioni preliminari per configurare l'ambiente di sviluppo

Per iniziare il nuovo progetto è stato copiato l'esempio Coexistence in una nuova cartella e sono state fatte diverse modifiche di seguito presentate.

8.1.1 SDK configuration header file

The SDK configuration header file (sdk_config.h) helps to manage the static configuration of an application that is built on top of nRF5 SDK. The configuration options included in this file can be quickly edited in a GUI wizard that is generated from the CMSIS Configuration Wizard Annotations. This annotations standard is supported natively by ARM Keil uVision (versions 4 and 5) or can be parsed using an open source Java tool - CMSIS Configuration Wizard. Every module in SDK contains at least one configuration option that enables this module. If the module is disabled, then even if source code is added to the project, it is not compiled because the module implementation is conditionally included. With only a quick evaluation of the sdk_config.h file, you can check which modules are used in the application:

```
example_module.c
#include "sdk_config.h"
#if EXAMPLE_MODULE_ENABLED
```

```
...
#endif //EXAMPLE_MODULE_ENABLED
```

For each supported device, there is a generic sdk_config.h file that contains all available configuration options.

- sdk/nrf5/config/nrf52810/config/sdk_config.h for nRF52810
- sdk/nrf5/config/nrf52832/config/sdk_config.h for nRF52832
- sdk/nrf5/config/nrf52840/config/sdk_config.h for nRF52840

For each supported device, there are generic linker files for Segger Embedded Studio and ARM GCC. These linker files contain all memory sections used by SDK modules.

- For ARM GCC: config/<device_name>/armgcc/generic_gcc_nrf52.ld
- For Segger Embedded Studio: config/<device_name>/ses/flash_placement.xml

These files are not actually used by any of the SDK projects, but they can serve as a template when developing a new application.

The open source Java tool CMSIS Configuration Wizard can be integrated with Segger Embedded Studio. The Java tool is bundled with the release and all examples have a reference to this tool (See "Project Macros" in project options).

The External Tools Configuration must be updated to enable integration with CMSIS Config Wizard.

1. Go to **File → Open Studio Folder... → External Tools Configuration**.
2. The tools.xml file will be opened in the editor.
3. Insert the following text before the </tools> tag:

```
<item name="Tool.CMSIS_Config_Wizard" wait="no">
<menu>&amp;amp;CMSIS Configuration Wizard</menu>
<text>CMSIS Configuration Wizard</text>
<tip>Open a configuration file in CMSIS Configuration
    Wizard</tip>
<key>Ctrl+Y</key>
<match>*config*.h</match>
<message>CMSIS Config</message>
<commands>
```

```
java -jar &quot;$(CMSIS_CONFIG_TOOL)&quot; &quot;$(
    InputPath)&quot;
</commands>
</item>
```

Ho integrato i file header sdk_config.h dell'esempio eddystone e coexistence. Le differenze sono elencate in tabella. Partendo dal file sdk_config.h dell'esempio coexistence, i moduli che sono abilitati nell'esempio Eddystone ma sono disabilitati o non definiti nel file originale, sono stati abilitati.

Configuration	Coexist	Eddystone
Queue size for BLE GATT Queue module	4	undefined
Database discovery module	enabled	undefined
BLE GATT Queue Module	enabled	undefined
Queued writes support module (prepare/execute write)	enabled	disabled
Peer Manager	enabled	disabled
central-specific Peer Manager functionality	disabled	enabled
Battery Service	enabled	disabled
Eddystone Configuration Service	undefined	enabled
Immediate Alert Service Client	enabled	disabled
Immediate Alert Service	enabled	disabled
Link Loss Service	enabled	disabled
TX Power Service	enabled	disabled
AES CBC mode using CC310	enabled	disabled
AES CTR mode using CC310	enabled	disabled
AES CBC_MAC mode using CC31	enabled	disabled
AES CMAC mode using CC310	enabled	disabled
AES CCM mode using CC310	enabled	disabled
AES CCM* mode using CC310	enabled	disabled
CHACHA-POLY mode using CC310	enabled	disabled
secp160r1 elliptic curve support using CC310	enabled	disabled
secp160r2 elliptic curve support using CC310	enabled	disabled
secp192r1elliptic curve support using CC310	enabled	disabled
secp224r1 elliptic curve support using CC310	enabled	disabled
secp256r1 elliptic curve support using CC310	enabled	disabled
secp384r1 elliptic curve support using CC310	enabled	disabled
secp521r1 elliptic curve support using CC310	enabled	disabled
secp160k1 elliptic curve support using CC310	enabled	disabled
secp192k1 elliptic curve support using CC310	enabled	disabled
secp224k1 elliptic curve support using CC310	enabled	disabled

secp256k1 elliptic curve support using CC310	enabled	disabled
Ed25519 curve support using CC310	enabled	disabled
CC310 backend implementation for hardware-accelerated SHA-256	enabled	disabled
CC310 backend implementation for SHA-512	enabled	disabled
CC310 backend implementation for HMAC using SHA-512	enabled	disabled
Cifra backend	disabled	enabled
mbed TLS backend	disabled	enabled
AES CBC mode mbed TLS	enabled	disabled
AES CTR mode using mbed TLS	enabled	disabled
AES CFB mode using mbed TLS	enabled	disabled
AES CBC MAC mode using mbed TLS	enabled	disabled
AES CMAC mode using mbed TLS	enabled	disabled
AES CCM mode using mbed TLS	enabled	disabled
AES GCM mode using mbed TLS	enabled	disabled
secp192r1 (NIST 192-bit) support using MBEDTLS	enabled	disabled
secp224r1 (NIST 224-bit) support using MBEDTLS	enabled	disabled
secp256r1 (NIST 256-bit) support using MBEDTLS	enabled	disabled
secp384r1 (NIST 384-bit) support using MBEDTLS	enabled	disabled
secp521r1 (NIST 521-bit) support using MBEDTLS	enabled	disabled
secp192k1 (Koblitz 192-bit) support using MBEDTLS	enabled	disabled
secp224k1 (Koblitz 224-bit) support using MBEDTLS	enabled	disabled
secp256k1 (Koblitz 256-bit) support using MBEDTLS	enabled	disabled
bp256r1 (Brainpool 256-bit) support using MBEDTLS	enabled	disabled
bp384r1 (Brainpool 384-bit) support using MBEDTLS	enabled	disabled
bp512r1 (Brainpool 512-bit) support using MBEDTLS	enabled	disabled
Curve25519 support using MBEDTLS	enabled	disabled

mbed TLS backend implementation for SHA-256	enabled	disabled
mbed TLS backend implementation for SHA-512	enabled	disabled
mbed TLS backend implementation for HMAC using SHA-256	enabled	disabled
mbed TLS backend implementation for HMAC using SHA-512	enabled	disabled
nRF HW RNG backend	disabled	enabled
Oberon backend	disabled	enabled
CHACHA-POLY mode using Oberon	enabled	disabled
secp256r1 curve support using Oberon library	enabled	disabled
Ed25519 signature scheme	enabled	disabled
Oberon backend implementation for SHA-256	enabled	disabled
Oberon backend implementation for SHA-512	enabled	disabled
Oberon backend implementation for HMAC using SHA-512	enabled	disabled
Use static memory buffers for context and temporary init buffer	undefined	enabled
Initialize the RNG module automatically when nrf_crypto is initialized	undefined	enabled
RNG peripheral driver	disabled	enabled
RNG peripheral driver - legacy layer	disabled	enabled
SAADC_CONFIG_LP_MODE Enabling low power mode	disabled	enabled
Enable scheduling app_timer events to app_scheduler	disabled	enabled
Enable RTC always on	disabled	enabled
Number of virtual flash pages to use	3	10
Size of the internal queue	4	10
Dynamic memory allocator	disabled	enabled
Log RTT backend	enabled	disabled

Because more than one backend module are enabled I disabled the following modules

```
#define NRF_CRYPTO_BACKEND_OBERON_ECC_SECP256R1_ENABLED 0

#define NRF_CRYPTO_BACKEND_MBEDTLS_ECC_CURVE25519_ENABLED 0

#define NRF_CRYPTO_BACKEND_MBEDTLS_HASH_SHA256_ENABLED 0

#define NRF_CRYPTO_BACKEND_MBEDTLS_HASH_SHA512_ENABLED 0
```

```
#define NRF_CRYPTO_BACKEND_OBERON_HASH_SHA512_ENABLED 0

#define NRF_CRYPTO_BACKEND_MBEDTLS_HMAC_SHA256_ENABLED 0

#define NRF_CRYPTO_BACKEND_MBEDTLS_HMAC_SHA512_ENABLED 0
```

Then I changed

```
#define FDS_VIRTUAL_PAGES 3

#define FDS_OP_QUEUE_SIZE 4
```

Debugging the code, it gets the following error <error> app: ERROR 9 [NRF_ERROR_INVALID_LENGTH] at 0x00066647. It has been solved writing in sdk_config.h

```
#ifndef APP_TIMER_CONFIG_USE_SCHEDULER
#define APP_TIMER_CONFIG_USE_SCHEDULER 0
#endif
```

A questo punto il file header sdk_config.h è stato configurato correttamente.

8.1.2 Directory del preprocessore

In **Options** → **Prepocessor** → **User Include Directories** sono state aggiunte le seguenti directory:

- /components/ble/ble_services/eddystone
- /components/ble/ble_services/ble_escs
- /components/libraries/crypto/backend/cc310
- /components/libraries/crypto/backend/cc310_bl
- /components/libraries/crypto/backend/cifra
- /components/libraries/crypto/backend/mbedtls
- /components/libraries/crypto/backend/micro_ecc
- /components/libraries/crypto/backend/nrf_hw
- /components/libraries/crypto/backend/nrf_sw
- /components/libraries/crypto/backend/oberon

- /components/libraries/crypto/backend/optiga
- /components/libraries/stack_info
- /external/mbedtls/include
- /external/micro-ecc/micro-ecc
- /external/nrf_cc310/include
- /external/nrf_oberon
- /external/nrf_oberon/include
- /external/nrf_tls/mbedtls/nrf_crypto/config

8.1.3 Macro del preprocessore

In **Options → Prepocessor → Preprocessor Definitions** sono state aggiunte le macro MBEDTLS_CONFIG_FILE="nrf_crypto_mbedtls_config.h" e DEBUG, quest'ultima necessaria per abilitare il debug.

8.1.4 File sorgente

. Ho aggiunto le seguenti cartelle (e il rispettivo contenuto) presenti nel progetto Eddystone al nuovo progetto

- nRF_Crypto
- nRF_Crypto backend cifra
- nRF_Crypto backend mbed TLS
- nRF_Crypto backend nRF HW
- nRF_Crypto backend Oberon
- nRF_Crypto backend uECC
- nRF_micro-ecc
- nRF_Oberon_Crypto
- nRF_TLS

Ho aggiunto il file sorgente nrf_ble_escs.c nella cartella nRF_BLE_Services e i seguenti file nella cartella nRF_Drivers:

- nrf_drv_rng.c
- nrf_nvmc.c
- nrfx_rng.c

Nella cartella nRF_Libraries ho aggiunto i seguenti file contenuti in /components/ble/ble_services/eddystone/

- es_adv.c
- es_adv_frame.c
- es_adv_timing.c
- es_adv_timing_resolver.c
- es_battery_voltage_saadc.c
- es_flash.c
- es_gatts.c
- es_gatts_read.c
- es_gatts_write
- es_security.c
- es_slot.c
- es_slot_reg.c
- es_stopwatch.c
- es_tlm.c
- nrf_ble_es.c

e i seguenti file contenuti in /external/cifra_AES128-EAX/

- cifra_eax_aes.c
- modes.c
- eax.c
- blockwise.c

- cifra_cmac.c
- cifra_AES128-EAX/gf128.c

e il file mem_manager.c contenuto in /components/libraries/mem_manager/.

A questo punto le operazioni preliminari per fare coesistere il servizio Eddystone con la Mesh sono completate.

8.2 Implementazione delle funzionalità Eddystone

In questa sezione sarà illustrato come implementare le funzioni che gestisono il servizio Eddystone nel progetto. Nel file main.c è stato inserita la libreria nrf_ble_es.h

```
#include "nrf_ble_es.h"
```

e le macro

```
#define NON_CONNECTABLE_ADV_LED_PIN      BSP_BOARD_LED_0
        //!< Toggles when non-connectable advertisement is
    sent.
#define CONNECTED_LED_PIN                BSP_BOARD_LED_1
        //!< Is on when device has connected.
#define CONNECTABLE_ADV_LED_PIN          BSP_BOARD_LED_2
        //!< Is on when device is advertising connectable
    advertisements.
```

Nella funzione **bsp_event_handler** è stato aggiunta **nrf_ble_es_on_start_connectable_** necessaria per mandare "connectable advertising" quando si preme il pulsante 2 (BSP_EVENT_KEY_2).

```
switch (event)
{
  case BSP_EVENT_KEY_2:
  {
    nrf_ble_es_on_start_connectable_advertising();
  } break;
  default:
  break;
}
```

Inoltre è stata implementata la funzione **on_es_evt** che gestisce gli eventi Eddystone accendendo dei led per dare un segnale visivo sullo stato degli advertising trasmessi.

```
static void on_es_evt(nrf_ble_es_evt_t evt)
{
  switch (evt)
  {
    case NRF_BLE_ES_EVT_ADVERTISEMENT_SENT:
      bsp_board_led_invert(NON_CONNECTABLE_ADV_LED_PIN);
    break;

    case NRF_BLE_ES_EVT_CONNECTABLE_ADV_STARTED:
      bsp_board_led_on(CONNECTABLE_ADV_LED_PIN);
    break;

    case NRF_BLE_ES_EVT_CONNECTABLE_ADV_STOPPED:
      bsp_board_led_off(CONNECTABLE_ADV_LED_PIN);
    break;

    default:
      break;
  }
}
```

A questo punto, nel **main**() dopo la funzione **services_init**() ho aggiunto la chiamata alla funzione **nrf_ble_es_init**(on_es_evt) la cui definizione è presente nel file nrf_ble_es.c precedentemente aggiunta come descritto nel capitolo 8.1.4. Tutte le funzionalità presenti nell'esempio Eddystone sono ora state implementate.

Per trasmettere gli iBeacon, nel file mesh_main.c sono state aggiunte le funzioni **adv_init**() e **adv_start**() per implementare e trasmettere iBeacon come descritto nel capitolo 6.2.1.

Debuggando il codice si presentano alcuni errori. Il primo è ERROR 4 [NRF_ERROR_NO_MEM] quando il programma ritorna dalla chiamata alla funzione **nrf_ble_escs_init**() che contiene **sd_ble_uuid_vs_add**(). Quest'ultima ritorna NRF_ERROR_NO_MEM quando non ci sono slot liberi per "vendor UUID". Per risolvere questo errore in sdk_config.h ho impostato Per risolverlo, nel file sdk_config.h si deve mettere

```
#define NRF_SDH_BLE_VS_UUID_COUNT 1
```

Dopo aver modificato questa macro, ho cambiato le dimensioni della ram dedicata al programma come suggerito dalle informazioni di debug. I valori da impostare nell'esempio sono riportati in tabella 5. Ulteriori informazioni sulle impostazioni della RAM e della FLASH sono disponibili nella discussione Adjustment of RAM and Flash memory. Per cambiare le dimen-

Softdevice	Version	Minimum RAM Start	FLASH Start
S132	7.0.0	0x20001668	0x26000

Table 5: RAM e FLASH start

sioni della RAM occorre andare in **Project** → **Edit Options** → **Common Configuration** → **Linker** → **Section Placement Macros**. Nel progetto corrente è composto da

- FLASH_PH_START=0x0
- FLASH_PH_SIZE=0x80000
- RAM_PH_START=0x20000000
- RAM_PH_SIZE=0x10000
- FLASH_START=0x26000
- FLASH_SIZE=0x5a000
- RAM_START=0x20002430
- RAM_SIZE=0xdbd0

sono state cambiate le macro RAM_START e RAM_SIZE precedentemente uguali a 0x20002420 e 0xdbe0 rispettivamente. Infatti per ogni nuovo UUID che si aggiunge occorre aumentare di 16 byte RAM_START e sottrarre 16 byte da RAM_SIZE, come indicato nella discussione sd_ble_uuid_vs_add error code 4. Si noti che **FLASH_SIZE** non deve essere più grande di (**FLASH_PH_SIZE - FLASH_START**), e che (**RAM_SIZE** + **RAM_START**) non deve essere più grande di (**RAM_PH_START** + **RAM_PH_SIZE**).

Successivamente si presenta l'errore ERROR 8 [NRF_ERROR_INVALID_STATE] dovuto alla chiamata alla funzione **es_battery_voltage_init**() in **nrf_ble_es_init**() perché inizializza il convertitore analogico digitale a successive approssimazioni (SAACD) che è stato giù inizializzato precedemente nel main.c dentro la funzione **adc_configure**(), già presente nell'esempio coexistence. Rimuovendo quest'ultima funzione si può risolvere l'errore perché l'ADC sarà inizializzato in **es_battery_voltage_init**().

Risolto l'errore precedente, si ha ERROR 4 [NRF_ERROR_NO_MEM] nella funzione **sd_ble_gap_adv_set_configure**() contenuta in**adv_start**, nel file es_adv.c, che è richiamata da **es_adv_start_non_connectable_adv**() in nrf_ble_es.c dovuto al fatto che non c'è abbasta memoria per configurare

un nuovo handle di advertising. Per eliminare l'errore si possono rimuovere le funzioni **advertising_init()** e **advertising_start()**, già presenti nell'esempio coexistence, che aggiungono handle di advertising non necessari in questo progetto.

Continuando a debuggare il codice, quando si preme il pulsante 2 per mandare "connectable advertising" si ottiene ERROR 5 [NRF_ERROR_NOT_FOUND] quando viene chiamata la funzione **sd_ble_gap_adv_start** dichiarata come

```
sd_ble_gap_adv_start(uint8_t adv_handle, uint8_t
    conn_cfg_tag));
```

Dalla documentazione si può osservare che se il parametro conn_cfg_tag non è stato trovato si ottiene questo errore. Poiché la funzione viene precedentemente chiamata per mandare iBeacon con conn_cfg_tag pari a 1, occorre che le macro MESH_SOFTDEVICE_CONN_CFG_TAG e BLE_CONN_CFG_TAG_DEFAULT siano uguali. Questo non è cosi di default perché nella libreria mesh_adv.h si ha

```
#define MESH_SOFTDEVICE_CONN_CFG_TAG 1
```

e in ble.h

```
#define BLE_CONN_CFG_TAG_DEFAULT  0
```

Risolto questo errore, si possono usare le funzionalità dell'esempio Eddystone nella rete Mesh.

8.3 Eliminazione delle funzionalità dell'esempio coexistence non necessarie

Per alleggerire il codice si possono eliminare le funzionalità dell'esempio coexistence che non sono necessarie. Sono stati rimmosse le seguenti funzioni:

- **saadc_event_handler**
- **adc_configure**
- **tps_init**
- **ias_init**
- **lls_init**
- **bas_init**

- **ias_client_init**
- **alert_signal**
- **on_ias_evt**
- **on_lls_evt**
- **on_ias_c_evt**
- **on_bas_evt**

queste abilitano il convertitore ADC e servizi GATT non necessari e gestiscono i relativi eventi. Una volta rimosse queste funzioni, si possono eliminare i seguenti file dal progetto:

- ble_bas.c
- ble_ias.c
- ble_ias_c.c
- ble_lls.c
- ble_tps.c

e le seguenti librerie:

- ble_tps.h
- ble_ias.h
- ble_lls.h
- ble_bas.h
- ble_ias_c.h

Una volta rimosse le funzionalità non necessarie, le risorse utilizzate dal programma sono 373.8 KB di memoria flash su 512 KB disponibile e 44.6 KB di memoria RAM su 64 KB disponibili come si può vedere dalla figura 34.

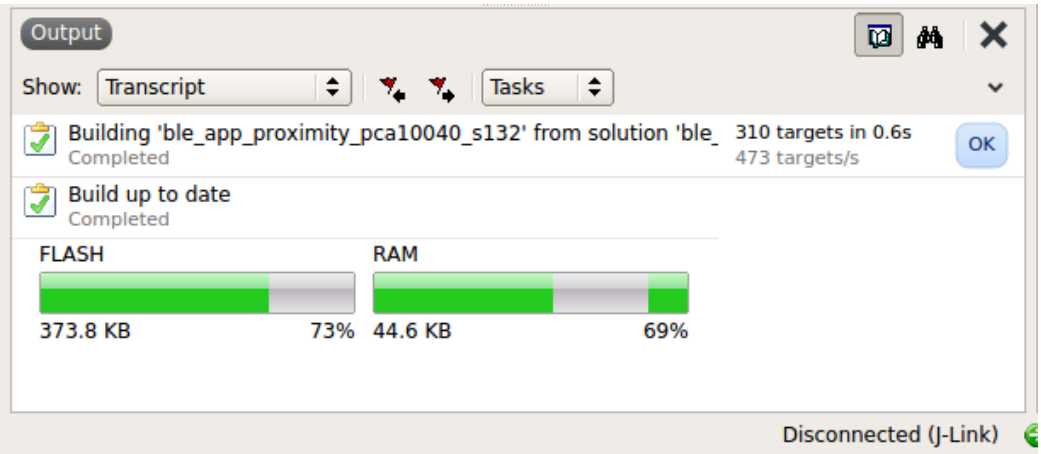


Figure 34: Risorse utilizzate

9 Conclusions

L'applicazione realizzata nel progetto funziona correttamente. Nella relazione vengono descritte diverse modalità di implementazione delle funzionalità tra cui l'ultimo capitolo descrive l'implementazione finale.

Caricando il firmware nella board, usando l'applicazione nRF Connect si può anche notare che viene inviato un iBeacon, inoltre è possibile connettersi alla scheda e tramite il servizio GATT Eddystone Configuration Service si può configurare il beacon Eddystone trasmesso. Con questo programma non è possibile usare l'app nRF Mesh per fare il provisioning del nodo in quanto non è abilitato il Proxy GATT.

Per modificare l'intervallo di advertising per i beacon Eddystone è possibile cambiare le macro APP_CFG_NON_CONN_ADV_INTERVAL_MS e PP_CFG_CONN_ADV_INTERVAL_MS nella libreria es_app_config.h. Per cambiare l'intervallo di advertising degli iBeacon si può usare la macro BEARER_ADV_INT_DEFAULT_MS nella libreria nrf_mesh_config_bearer.h.

References

[1] Ericsson, Lenovo, Intel Corporation, Microsoft Corporation, Motorola, Nokia Corporation, and Toshiba Corporation. *Specification of the Bluetooth System*. Vol. 4.0. 2010. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification.

[2] Dixys Hernandez, Tiago Fernández-Caramés, Paula Fraga-Lamas, and Carlos Escudero. "Design and Practical Evaluation of a Family of Lightweight Protocols for Heterogeneous Sensing through BLE Beacons in IoT Telemetry Applications". In: *Sensors* 18 (Dec. 2017), p. 57. DOI: 10.3390/s18010057.

[3] *Introduction to BLE IoT*. 2019. URL: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsdk_nrf5_v16.0.0%2Fiot_intro.html&cp=6_1_1_7.

[4] *Light switch example*. 2019. URL: https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.meshsdk.v4.0.0/md_examples_light_switch_README.html?cp=7_2_3_0.

[5] Joakim Lindh. *Bluetooth low energy Beacons*. Texas Instruments. 2015.

[6] K. Townsend, C. Cufi, Akiba, and R. Dividson. *Getting Started with Bluetooth Low Energy*. O'Reilly, 2014.

[7] Martin Woolley. *Bluetooth Mesh Networking*. 2017. URL: https://www.bluetooth.com/specifications/mesh-specifications.