UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI **INGEGNERIA**

Corso di Laurea triennale in Ingegneria Elettronica

Sviluppo di firmware per il monitoraggio e il provisioning di sensori Bluetooth indossabili basati su chip nRF52840.

Firmware development for monitoring and provisioning of wereable Bluetooth sensors based on nRF52840 chip.

Relatore: Chiar.mo/a

Prof. **Biagetti Giorgio**

Tesi di Laurea di:

Rettaroli Andrea

A.A. **2019/2020**

Indice:

1) Introduzione …… 2
    1.1 Obiettivo …… 2
2) Hardware e software utilizzati …… 2
    2.1 Microcontrollore …… 2
    2.2 Periferica…… 3
    2.3 Materiale di sviluppo…… 4
3) Bluetooth…… 5
    3.1 Invio dei dati…… 5
    3.2 Bluetooth mesh…… 5
    3.3 Bluetooth Address…… 5
    3.4 Processo di connessione…… 6
    3.5 Pairing e Bonding…… 6
    3.6 Protocolli Bluetooth…… 7
4) USB…… 8
    4.1 Specifiche del sistema……8
    4.2 Enumerazione……8
    4.3 Endpoints……9
    4.4 Trasmissione dati…… 9
    4.5 USB Descriptor…… 9
    4.6 Trasmissione dei dati…… 10
5) Programma……11
    5.1 #include e #define……11
    5.2 Main……13
    5.3 Inizializzazioni……14
    5.4 USB events……19
    5.5 Bluetooth events……23
    5.6 Altri eventi……24
6) Conclusioni……26
7) Appendice……27

1) Introduzione

I microcontrollori, da qualche decennio, sono entrati a far parte della vita di tutti i giorni. Infatti sono il cervello calcolatore che gestisce quasi tutti i dispositivi elettronici presenti ai giorni nostri. Il microcontrollore è un minuscolo computer compatto fabbricato all'interno di un chip, è costruito appositamente per il sistema embedded ed è composto generalmente da una memoria di lettura e scrittura (RAM), una memoria di sola lettura (ROM), porte di input/output (I/O ports), un processore e uno o più clock integrati. Il microcontrollore è un dispositivo molto adattabile e può essere utilizzato per una vasta gamma di applicazioni, ma gestirà solo quello per cui è stato programmato dall'utente.

Questo progetto è parte di un progetto più grande, che ha lo scopo di utilizzare il microcontrollore e vari sensori biometrici (e non) per il monitoraggio fisiologico dell'attività umana e dell'analisi del movimento utile nei campi di applicazione come sanità, sport e fitness.

1.1 Obiettivo

L'obiettivo di questo progetto è quello di realizzare un firmware per il microcontrollore della Nordic Semiconductor nRF52840 System-on-Chip (SoC) che faccia sì che quest'ultimo si accoppi con un PC tramite Bluetooth e condivida con lui informazioni attraverso l'USB.

2) Hardware e software utilizzati

Microcontrollore:	nRF52840 System-on-Chip
Periferica:	nRF52840 Development Kit
Ambiente di sviluppo:	Eclipse
Sistema operativo:	Kubuntu 20.04 (Linux)

2.1 Microcontrollore

L'nRF52840 SoC è completamente multiprotocollo e permette di eseguire programmi, algoritmi o problemi in maniera parallela senza influire sul risultato finale. In questo modo soddisfa le sfide di applicazioni sofisticate che richiedono l'esecuzione contemporanea di protocolli e ha un insieme ricco e vario di periferiche e funzionalità. Offre un'ampia disponibilità di memoria sia per Flash che per RAM, che sono prerequisiti per applicazioni impegnative.

Caratteristiche principali:

- 64 MHz Cortex-M4 with FPU
- 1 MB Flash, 256 KB RAM
- 2.4 GHz Transceiver
- 2 Mbps, 1 Mbps, Long Range
- Bluetooth 5, Bluetooth mesh
- ANT, 802.15.4, Thread, Zigbee
- +8 dBm TX Power
- 128-bit AES CCM, ARM CryptoCell
- UART, SPI, TWI, PDM, I2S, QSPI
- PWM
- 12-bit ADC
- NFC-A
- USB 2.0

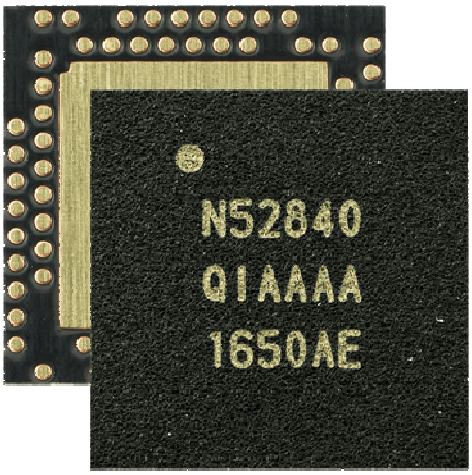


Foto dell'nRF52840 SoC

2.2 Periferica

L'nRF52840 Development Kit (DK) è una versatile scheda di sviluppo per applicazioni per Bluetooth Low Energy (BLE), Bluetooth mesh, Thread, Zigbee, ANT, facilitando lo sviluppo sfruttando tutte le caratteristiche dell'nRF52840 SoC. Tutti i GPIO (General Purpose Input/Output) sono raggiungibili tramite connettori montati sulla scheda e 4 pulsanti e 4 LED semplificano l'ingresso e l'uscita da e verso il SoC. Una memoria esterna integrata è collegata alla periferica QSPI (Queued Serial Peripheral Interface) nell'nRF52840 SoC. Il DK è tipicamente alimentato con USB, ma può essere alimentato da un'ampia gamma di sorgenti, nell'intervallo di alimentazione da 1,7 a 5,0 V.
Caratteristiche principali:

- Supports Bluetooth LE, Bluetooth mesh, NFC, Thread and Zigbee
- User-programmable LEDs(4) and buttons(4)
- 2.4 GHz and NFC antennas
- SWF RF connector for direct RF measurements
- On-board SEGGER J-Link debugger/programmer

- Arduino Uno Rev3 form factor
- Pins for measuring power consumption
- 1.7-5.0 V supply from USB, external, Li-Po or CR2032 coin cell battery

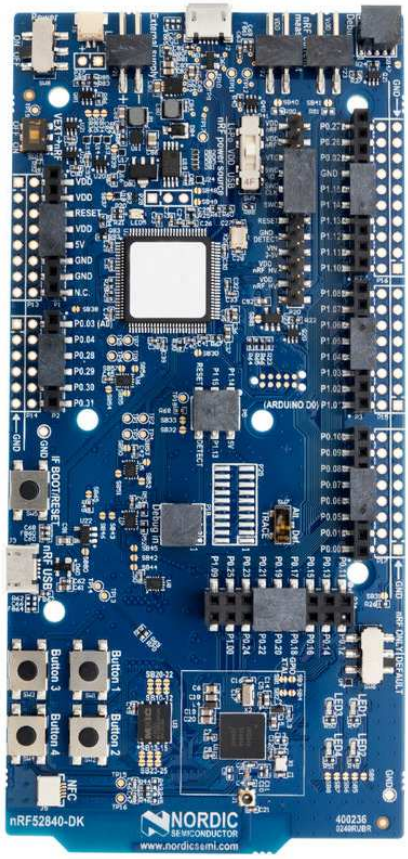


Foto dell'nRF52840 DK

2.3 Materiale di sviluppo

La Nordic Semiconductor fornisce il Software Development Kit (SDK) per la creazione di applicazioni complete, affidabili e sicure con le serie nRF52 e nRF51. Questo software offre agli sviluppatori una vasta gamma di moduli ed esempi diversi, includendo un'ampia selezione di driver, librerie ed esempi per periferiche. La versione dell'SDK utilizzato in questo progetto è la 17.0.2.
Per la realizzazione del codice in esame si è sfruttato inoltre il SoftDevice S140, che è uno stack di protocollo Bluetooth Low Energy (BLE) ricco di funzionalità per i System-on-Chips (SoC) nRF52811, nRF52820, nRF52833 e nRF52840. Supporta fino a 20 collegamenti simultanei in tutti i ruoli (emittente, centrale, osservatore, periferico), integra un BLE controller e host e fornisce un'API completa e flessibile per la creazione di programmi basati su BLE nRF5 SoC. È certificato Bluetooth 5.1 e supporta le seguenti funzionalità Bluetooth: 2 Mbps ad alta velocità, lungo raggio, Advertising Extensions e algoritmo di selezione del canale #2 (CSA #2). Il numero di connessioni e la larghezza di banda per connessione sono configurabili, offrendo ottimizzazione delle prestazioni e della memoria.

3) Bluetooth

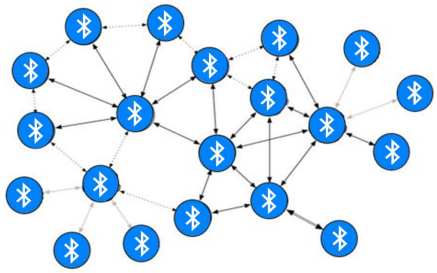
Il Bluetooth è uno standard per l'invio di dati binari su frequenze radio (2.4GHz circa) a breve distanza (<100m). I dispositivi Bluetooth contengono quindi mittenti e ricevitori radio, che inviano gli 0 e gli 1 attraverso l'aria in pacchetti. In termini semplici, il Bluetooth prende le informazioni normalmente trasportate dal cavo e le trasmette a una frequenza speciale a un altro dispositivo Bluetooth.

3.1 Invio dei dati

Le informazioni possono essere scambiate a quattro velocità differenti: 125 kbit/s, 500 kbit/s, 1 Mbit/s, 2 Mbit/s. I dati trasmessi vengono suddivisi in pacchetti e ciascun pacchetto viene trasmesso su uno dei 40 canali Bluetooth designati (da 2400Hz a 2480Hz). Ogni canale, quindi, ha una larghezza di banda di 2 MHz. I canali 37, 38 e 39 sono chiamati canali di advertisement, mentre gli altri sono canali per i dati. I canali di advertisement vengono utilizzati per il rilevamento dei dispositivi, durante la creazione della connessione, e per la trasmissione. I canali per i dati, invece, sono utilizzati per la comunicazione bidirezionale tra i dispositivi collegati.

3.2 Bluetooth mesh

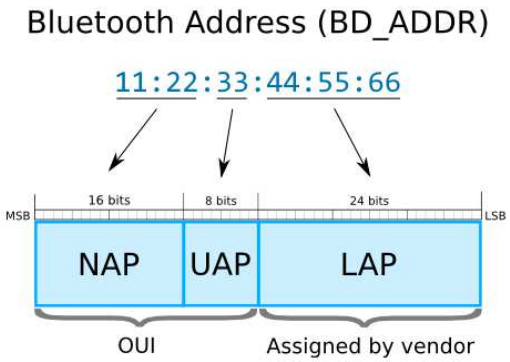
In precedenza, il protocollo Bluetooth era basato sulla trasmissione a pacchetti con struttura master-slave, dove un unico dispositivo fungeva da hub centrale per la gestione dei messaggi tra i vari dispositivi nella rete.



Nelle nuove versioni, il Bluetooth sfrutta una tecnologia utilizzata anche nei dispositivi WiFi, la cosiddetta rete mesh. Con questo termine ci si riferisce a una particolare topologia di rete locale in cui i nodi (cioè i dispositivi) dell'infrastruttura si connettono direttamente, dinamicamente e non gerarchicamente al maggior numero possibile di altri nodi e cooperano tra loro per instradare in modo efficiente i dati. Questa mancanza di dipendenza da un server centrale consente a ogni nodo di partecipare alla trasmissione delle informazioni. Nel caso in cui uno dei suoi nodi smettesse di funzionare, quindi, gli altri saranno comunque in grado di mantenere in vita la rete e continuare a comunicare tra di loro.

3.3 Bluetooth Address

Ogni singolo dispositivo Bluetooth ha un indirizzo univoco a 48 bit, comunemente abbreviato BD_ADDR. Di solito viene presentato sotto forma di un valore esadecimale di 12 cifre (esempio: 00:11:22:33:FF:EE). La metà più significativa (24 bit) dell'indirizzo è un identificatore univoco dell'organizzazione (OUI), che identifica il produttore. I prefissi OUI vengono assegnati dall'Institute of Electrical and Electronics Engineers (IEEE).



Il Bluetooth Address è formato da tre parti: NAP, UAP and LAP.
Il NAP (Non-significant Address Part) contiene i primi 16 bit dell'OUI. Il valore NAP viene utilizzato nei frame di sincronizzazione con salto di frequenza.
L'UAP (Upper Address Part) contiene gli 8 bit rimanenti dell'OUI. Il valore UAP viene utilizzato per il seeding in vari algoritmi di specifica Bluetooth.
Il LAP (Lower Address Part) viene assegnata dal fornitore del dispositivo. Il valore LAP identifica in modo univoco un dispositivo Bluetooth come parte del codice di accesso in ogni frame trasmesso.
Il LAP e l'UAP costituiscono la parte significativa dell'indirizzo (SAP) dell'indirizzo Bluetooth.

3.4 Processo di connessione

Ogni dispositivo Bluetooth in modalità discoverable (ovvero in grado di essere rilevato da altri dispositivi) trasmette, ad intervalli regolari, pacchetti contenenti: nome del dispositivo, classe del dispositivo, lista dei servizi offerti e altre informazioni (es. marca o altre caratteristiche del dispositivo). Questa operazione è detta advertising. Altri dispositivi possono effettuare la ricerca per rilevare device in modalità discoverable e ottenere informazioni su di essi tramite la ricezione dei messaggi di advertising.
La creazione di una connessione Bluetooth tra due dispositivi è un processo in più fasi che coinvolge tre stati progressivi:

- Inquiry: Se due dispositivi Bluetooth non sanno assolutamente nulla l'uno dell'altro, uno dei due deve eseguire una iniquiry per cercare di scoprire l'altro. Un dispositivo invia la richiesta di iniquiry e qualsiasi dispositivo in ascolto di tale richiesta risponderà con il suo indirizzo, ed eventualmente il suo nome e altre informazioni
- Paging: è il processo di creazione di una connessione tra due dispositivi Bluetooth. Prima che questa connessione possa essere avviata, ogni dispositivo deve conoscere l'indirizzo dell'altro (trovato nel processo di richiesta)
- Connection: dopo che un dispositivo ha completato il processo di paging, entra nello stato di connessione. Durante la connessione, un dispositivo può partecipare attivamente o può essere impostato in una modalità di sospensione a basso consumo

3.5 Pairing e Bonding

Due dispositivi possono creare un legame, detto bonding, per cui stabiliscono automaticamente una connessione ogni volta che sono abbastanza vicini.

Questo legame viene creato tramite un singolo processo chiamato pairing (accoppiamento). Quando i dispositivi si accoppiano, condividono i loro indirizzi, nomi e profili e di solito li archiviano in memoria. Condividono anche una chiave segreta comune, che consente loro di legarsi ogni volta che sono insieme in futuro. L'associazione di solito richiede un processo di autenticazione in cui un utente deve convalidare la connessione tra i dispositivi. Il processo di autenticazione varia e solitamente dipende dalle capacità dell'interfaccia di un dispositivo o dell'altro.
In generale, la procedura di pairing si divide in tre fasi:

- Pairing Feature Exchange
- LE Legacy Pairing (LELP) oppure LE Secure Connections (LESC)
- Transport Specific Key Distribution

Nella prima fase vengono scambiate funzionalità di sicurezza che includono cose come capacità di input/output (IO), requisiti per la protezione Man-In-The-Middle, disponibilità di bonding, ecc….
Dopo la prima fase, entrambi i dispositivi possono selezionare quale metodo di generazione della chiave utilizzare:

- Just Works, in cui non si crea una chiave, per cui si può essere esposti ad attacchi esterni che mirano a leggere i dati scambiati dai dispositivi
- Out-of-Band(OOB), dove la chiave verrà poi scambiata tramite uno standard differente dal Bluetooth, ad esempio con antenna NFC
- Passkey, in cui la chiave è una parola di 6 cifre, e queste cifre devono essere tutti numeri compresi tra 0 e 9
- Numeric Comparison, dove entrambi i dispositivi generano indipendentemente un valore di conferma a 6 cifre sulla base di dati precedentemente scambiati. Questi valori sono poi visualizzati dall’utente, che dovrà controllare che entrambi siano uguali

I primi tre metodi vengono supportati sia dalla fase LELP che dalla fase LESC, mentre l’ultimo metodo è disponibile solo per LESC. Per rendere sicura la connessione, entrambi queste fasi generano anche una chiave per criptare i dati trasmessi nel canale di comunicazione.
Infine, nella fase Transport Specific Key Distribution vengono scambiate le chiavi tra i dispositivi.

3.6 Protocolli Bluetooth

Infine, è giusto ricordare che il Bluetooth non è un unico protocollo, ma ha al suo interno una varietà di altri protocolli. I protocolli Bluetooth possono essere divisi in due grandi stack: controller stack, contenente l'interfaccia radio per la temporizzazione, e host stack che si occupa di dati di alto livello. Tra i vari sotto-protocolli, una menzione importante va al protocollo GAP (Generic Access Profile) che definisce come i dispositivi BLE possono rendersi disponibili e come due dispositivi possono comunicare direttamente tra loro.

4) USB

Il nome stesso del protocollo USB, ovvero Universal Serial Bus, mette in evidenza uno degli scopi principali di questo standard, ovvero poter essere universale e indipendente dalla periferica. Una flessibilità fornita dal fatto che lo standard supporta diversi bit rate a seconda delle esigenze.
L'USB utilizza due fili per fornire alimentazione e due fili per trasferire i segnali, fornendo un meccanismo di trasferimento dati seriale sufficientemente veloce per la comunicazione e la possibilità di ottenere l'alimentazione tramite il connettore.

4.1 Specifiche del sistema

L'USB utilizza un protocollo del tipo master/slave, in cui il dispositivo "master" controlla la comunicazione con (e tra) i dispositivi "slave". Generalmente, un PC è al centro della rete mentre i vari dispositivi sono collegati al PC. In termini di definizioni delle entità in una rete USB, ci sono 5 elementi principali:

- Port: è la presa attraverso la quale si ottiene l'accesso alla rete USB. Può essere su un Host o su un Hub
- Hub: sono dispositivi che espandono una singola porta USB in modo che diversi dispositivi USB possano connettersi. È possibile collegare un Hub a un altro per espandere ulteriormente la capacità e la connettività
- Host: è il computer o l'elemento che funge da controller principale per il sistema USB. L'Host ha un Hub contenuto al suo interno e questo è chiamato Root Hub
- Function: queste sono le periferiche o gli elementi a cui è collegato il collegamento USB (mouse, tastiere, memorie Flash, ecc…)
- Device: questo termine viene utilizzato collettivamente per Hub e funzioni

La rete USB può supportare fino a 127 nodi esterni. A causa dei vincoli temporali per la propagazione del segnale, il numero massimo di livelli di Hub consentiti è 7. In termini di flusso di dati, i segnali viaggiano lungo il bus in modo che sia accessibile a tutte le funzioni, ma solo la funzione di destinazione lo accetta.

4.2 Enumerazione

Quando un dispositivo USB è collegato al bus USB, l'Host lo inizializza associandogli un indirizzo unico che verrà poi utilizzato per identificare in maniera univoca la periferica sul bus.
L'Host utilizza un processo noto come enumerazione del bus per identificare e configurare il dispositivo. L'Host USB invia le richieste di configurazione non appena il dispositivo si è connesso alla rete USB. Il dispositivo verrà istruito per selezionare una configurazione e un'interfaccia per soddisfare le esigenze dell'applicazione in esecuzione sull'Host USB.

4.3 Endpoints

Oltre all'indirizzo, il dispositivo contiene anche endpoints, che possono essere considerati dei buffer di memoria per mezzo dei quali è possibile scambiare informazioni tra l'Host e il dispositivo.
Gli endpoint possono funzionare solo in una direzione, ovvero input o output, ma non entrambi, e i dispositivi possono averne fino a 16, di cui uno deve essere riservato come 'Zero Endpoint' per quella direzione. Gli 'Zero Endpoint' vengono utilizzati per una varietà di attività tra cui il rilevamento automatico e la configurazione del dispositivo durante la fase di enumerazione.
Sebbene ogni dispositivo possa avere sedici endpoint di input e sedici di output, è molto raro che vengano utilizzati tutti.

4.4 Trasmissione dati

Il protocollo USB, al fine di supportare un'ampia tipologia di applicazioni, fornisce diverse modalità di trasmissione dati. In particolare:

- Control: viene generalmente utilizzata per la fase di inizializzazione, in cui l'Host invia comandi o parametri di query per leggere le informazioni di un dispositivo
- Interrupt: è un messaggio in polling (ciclico) dall'Host in cui viene interrogato il dispositivo sulla presenza o meno di dati specifici. Viene spesso utilizzato dai dispositivi che devono fornire dati in piccole quantità in maniera piuttosto rapida, ad esempio mouse o tastiere
- Bulk: viene utilizzato da dispositivi come le stampanti per cui sono necessarie quantità di dati molto maggiori. I blocchi di dati a lunghezza variabile vengono inviati o richiesti dall'Host, la loro integrità viene verificata utilizzando il controllo di ridondanza ciclico (CRC) e la loro ricezione viene confermata
- Isochronous: utilizzato per i dispositivi che trasmettono dati in tempo reale (come gli apparati audio o video) e non utilizza il controllo dei dati, poiché i dati persi possono essere adattati meglio dei ritardi subiti inviando nuovamente i dati

4.5 USB Descriptor

Si è visto precedentemente che quando un dispositivo viene collegato ad una rete USB, l'Host gli assegna un indirizzo unico e richiede delle informazioni per determinare quale driver debba essere caricato al fine di permettere il corretto funzionamento della periferica. Tutte queste informazioni sono contenute all'interno di vari Descriptor (descrittori) inclusi nel dispositivo. Quindi un Descriptor non è altro che un insieme di registri che contengono tutte le informazioni necessarie e vengono letti in successione. In particolare le varie informazioni sono distribuite in diversi Descriptor e ogni Descriptor è identificato da un codice.

I Descriptor definiti dalle specifiche USB sono i seguenti:

- Device Descriptor: indica, tra le altre cose, quali sono il fornitore e l'ID prodotto
- Configuration Descriptor: contiene informazioni sui requisiti di alimentazione del dispositivo e sul numero di interfacce che può supportare. Un dispositivo può avere più configurazioni, quindi L'Host può selezionare la configurazione che meglio corrisponde ai requisiti del software applicativo
- Interface Descriptor: definisce la raccolta degli endpoint, ma non include mai lo 'Zero Endpoint' nella numerazione degli endpoint
- Endpoint Descriptor: specifica il tipo di trasferimento, la direzione, l'intervallo di polling e la dimensione massima del pacchetto per ogni endpoint. Lo 'Zero Endpoint' viene sempre considerato un endpoint di controllo e non ha mai un descrittore
- String Descriptor: sono facoltativi e aggiungono informazioni leggibili dagli utenti agli altri descrittori

Tutti quei dispositivi che possono cambiare la velocità di comunicazione possiedono dei Descriptor aggiuntivi che permettono di fornire le informazioni del dispositivo e relative configurazioni nella loro modalità.

4.6 Trasmissione dei dati

La comunicazione è controllata sempre dall'Host ed avviene per mezzo della lettura e scrittura degli Endpoint. Il trasferimento di dati avviene per mezzo di transazioni multiple di dati. Ogni transazione, a seconda della tipologia della stessa, può essere divisa in: Token Packet, Data Packet e Handshake Packet
Il Token Packet permette di trasmettere le seguenti informazioni:

- PID: Packed Identifier
- Address: Indirizzo del dispositivo
- Endpoint: Numero dell'Endpoint
- CRC: Cyclic Redundancy Check per l'identificazione di eventuali errori

Il Data Packet contiene i seguenti campi:

- PID: Packed Identifier
- Data: Byte dati in numero dipendente dal tipo di trasferimento
- CRC: Cyclic Redundancy Check per l'identificazione di eventuali errori

L'Handshake Packet contiene il solo campo PID.
Il modo con cui una transazione viene effettivamente suddivisa, viene a dipendere dal tipo di transazione. In particolare le transazioni Control, Bulk, Interrupt possiedono tutti e tre i packet, mentre la modalità Isochronous possiede solo i primi due poiché non è importante avere la conferma della correttezza dei dati. L'Handshake Packet non serve solo per segnalare eventuali errori e richiedere di spedire nuovamente i dati, ma anche per permettere di avvertire l'Host se il Device non è pronto.

5) Programma

Il programma si compone di quattro macroaree: le direttive al compilatore, la funzione “main”, le funzioni di inizializzazione e le funzioni di “interrupt”, che vengono eseguite solo quando accade un evento esterno al microcontrollore.

5.1 #include e #define

La parte iniziale del programma è sempre composta da linee di codice “#include” e “#define”. La direttiva “#include” aggiunge un testo all'inizio del codice sorgente del programma. Il testo è tratto da un file esterno, detto header, e generalmente è una libreria di funzioni. La direttiva “#define”, invece, è usata per ribattezzare delle costanti sotto forma di parole, più comprensibili per chi programma e utile per una miglior lettura del testo.

```
#include <stdbool.h>
#include <stdint.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include <nrf_nvic.h>

#include <nrf_clock.h>
#include <nrf_pwr_mgmt.h>
#include <nrf_delay.h>

#include "app_util_platform.h"
#include "app_error.h"
#include "app_timer.h"
#include "app_scheduler.h"
#include "app_usbd.h"

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"

#include "boards.h"

#include "nrf.h"
#include "nrf_drv_usbd.h"
#include "nrf_drv_clock.h"

#include "app_util.h"
#include "app_usbd_core.h"
#include "app_usbd_string_desc.h"
#include "app_usbd_serial_num.h"
#include "app_usbd_cdc_acm.h"

#include "nordic_common.h"
#include "ble.h"
#include "ble_hci.h"
#include "ble_srv_common.h"
#include "ble_advdata.h"
```

```
#include "ble_advertising.h"
#include "ble_conn_params.h"
#include "nrf_sdh.h"
#include "nrf_sdh_soc.h"
#include "nrf_sdh_ble.h"
#include "fds.h"
#include "peer_manager.h"
#include "peer_manager_handler.h"
#include "ble_conn_state.h"
#include "nrf_ble_gatt.h"
#include "nrf_ble_qwr.h"

#include "ble_bas.h"
#include "ble_dis.h"

#include "nrf_soc.h"
#include "nrf_nvic.h"

struct system_power_type usb_power_status;


#define CDC_ACM_COMM_INTERFACE          0
#define CDC_ACM_COMM_EPIN               NRF_DRV_USBD_EPIN2

#define CDC_ACM_DATA_INTERFACE          1
#define CDC_ACM_DATA_EPIN               NRF_DRV_USBD_EPIN1
#define CDC_ACM_DATA_EPOUT              NRF_DRV_USBD_EPOUT1

#define DEVICE_NAME                     "PROVABLE"
#define MANUFACTURER_NAME               "DII@UnivPM"

#define APP_BLE_OBSERVER_PRIO           3
#define APP_BLE_CONN_CFG_TAG            1

#define MIN_CONN_INTERVAL               MSEC_TO_UNITS(7.5, UNIT_1_25_MS)
#define MAX_CONN_INTERVAL               MSEC_TO_UNITS(7.5, UNIT_1_25_MS)
#define SLAVE_LATENCY                   0
#define CONN_SUP_TIMEOUT                MSEC_TO_UNITS(4000, UNIT_10_MS)

#define FIRST_CONN_PARAMS_UPDATE_DELAY  APP_TIMER_TICKS(5000)
#define NEXT_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(30000)
#define MAX_CONN_PARAMS_UPDATE_COUNT    3

#define SEC_PARAM_BOND                  1
#define SEC_PARAM_MITM                  0
#define SEC_PARAM_LESC                  0
#define SEC_PARAM_KEYPRESS              0
#define SEC_PARAM_IO_CAPABILITIES       BLE_GAP_IO_CAPS_NONE
#define SEC_PARAM_OOB                   0
#define SEC_PARAM_MIN_KEY_SIZE          7
#define SEC_PARAM_MAX_KEY_SIZE          16

#define DEAD_BEEF                       0xDEADBEEF
```

```
#define READ_SIZE 1

// Define a buffer for the scheduler:
#define SCHED_MAX_EVENT_DATA_SIZE MAX(2 * sizeof (uint32_t), APP_TIMER_SCHED_EVENT_DATA_SIZE)
#define SCHED_QUEUE_SIZE 8


// CDC_ACM class instance:
APP_USBD_CDC_ACM_GLOBAL_DEF (m_app_cdc_acm,
                             cdc_acm_user_ev_handler,
                             CDC_ACM_COMM_INTERFACE,
                             CDC_ACM_DATA_INTERFACE,
                             CDC_ACM_COMM_EPIN,
                             CDC_ACM_DATA_EPIN,
                             CDC_ACM_DATA_EPOUT,
                             APP_USBD_CDC_COMM_PROTOCOL_AT_V250
);

NRF_BLE_GATT_DEF(m_gatt);                                       /**< GATT module instance. */
NRF_BLE_QWR_DEF(m_qwr);                                         /**< Context for the Queued Write module.*/
BLE_ADVERTISING_DEF(m_advertising);                             /**< Advertising module instance. */

BLE_BAS_DEF(m_bas);


// YOUR_JOB: Use UUIDs for service(s) used in your application.
static ble_uuid_t m_adv_uuids[] =                               /**< Universally unique service identifiers. */
{
    {BLE_UUID_DEVICE_INFORMATION_SERVICE, BLE_UUID_TYPE_BLE},
//    {CUSTOM_SERVICE_UUID, BLE_UUID_TYPE_VENDOR_BEGIN}
};

static uint32_t scheduler_buffer[CEIL_DIV(
    APP_SCHED_BUF_SIZE(SCHED_MAX_EVENT_DATA_SIZE, SCHED_QUEUE_SIZE),
    sizeof (uint32_t)
)];


static void cdc_acm_user_ev_handler (app_usbd_class_inst_t const * p_inst, app_usbd_cdc_acm_user_event_t event);
void advertising_start (bool erase_bonds);
void advertising_stop (void);


static uint16_t m_conn_handle = BLE_CONN_HANDLE_INVALID;                /**< Handle of the current connection. */
static bool m_advertising_active = false;

static char tx_buffer[32];
static char m_rx_buffer[READ_SIZE];

static volatile bool bt_started = false;
```

Qui vengono anche definite le variabili globali, cioè quelle variabili che sono visibili a tutte le funzioni presenti nel programma, e vengono anche implementate classi e strutture, utili per la creazione di oggetti che serviranno all'interno del programma.

5.2 Main

Il programma comprende poi una funzione "main", che è la prima parte di codice che viene eseguita dal microcontrollore. Questa funzione viene utilizzata, quindi, per la chiamata delle funzioni di inizializzazione e come loop principale, in cui al suo interno può essere eseguito il programma vero e proprio. Nel nostro caso, però, tutto quello che il "main" andrà a fare nel loop principale è aspettare la generazione di un evento esterno, per cui saranno poi le funzioni di "interrupt" a gestire la logica del programma.

```c
int main (void)
{
    // Initialize logging system:
    APP_ERROR_CHECK(NRF_LOG_INIT(NULL));
    NRF_LOG_DEFAULT_BACKENDS_INIT();

    NRF_LOG_INFO("Program started: %08X", NRF_POWER->RESETREAS);
    NRF_POWER->RESETREAS = (NRF_POWER->RESETREAS); // clear register
    NRF_LOG_FLUSH();

    // Initialize crystal:
    nrf_clock_lf_src_set(NRF_CLOCK_LFCLK_Xtal);
    nrf_clock_task_trigger(NRF_CLOCK_TASK_LFCLKSTART);

    // Initialize scheduler:
    ret_code_t err_code;
    err_code = app_sched_init(SCHED_MAX_EVENT_DATA_SIZE, SCHED_QUEUE_SIZE, scheduler_buffer);
    APP_ERROR_CHECK(err_code);

    // Initialize rest of application:
    err_code = nrf_pwr_mgmt_init();
    APP_ERROR_CHECK(err_code);

    usb_init();
    ble_init();

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

        while (app_usbd_event_queue_process()) ;

        if (NRF_LOG_PROCESS() == false)
            nrf_pwr_mgmt_run();
    }

    // Code shall never reach here!
    return 0;
}
```

5.3 Inizializzazioni

Le funzioni di inizializzazione servono ad attivare tutti quei registri del microcontrollore che sono utili per la realizzazione del programma. Di seguito verranno inserite tutti quelle funzioni chiamate nel codice, ma verranno spiegate solo quelle utili per capire la logica del programma.

Essendo un protocollo “semplice”, la funzione riguardante l’USB deve mettere in azione pochi registri. D’altra parte, il Bluetooth è composto da tanti e vari sotto-protocolli, quindi ha bisogno di richiamare molte funzioni di inizializzazione che andranno ad attivare tutti i registri utili per i suoi processi.

```c
int usb_init (void)
{
    ret_code_t ret;
    static const app_usbd_config_t usbd_config = {
        .ev_state_proc = usbd_user_ev_handler
    };
```

```
    app_usbd_serial_num_generate();

    ret = app_usbd_init(&usbd_config);
    APP_ERROR_CHECK(ret);

    app_usbd_class_inst_t const * class_cdc_acm = app_usbd_cdc_acm_class_inst_get(&m_app_cdc_acm);
    ret = app_usbd_class_append(class_cdc_acm);
    APP_ERROR_CHECK(ret);

    ret = app_usbd_power_events_enable();
    APP_ERROR_CHECK(ret);

    return ret;
}

void ble_init (void)
{
    ble_stack_init();
    gap_params_init();
    gatt_init();
    advertising_init();
    services_init();
    conn_params_init();
    peer_manager_init();

    advertising_start();

    bt_started = true;
}

/**@brief Function for initializing the BLE stack.
*
* @details Initializes the SoftDevice and the BLE event interrupt.
*/
static void ble_stack_init(void)
{
    ret_code_t err_code;

    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    // Configure the BLE stack using the default settings.
    // Fetch the start address of the application RAM.
    uint32_t ram_start = 0;
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    // Enable BLE stack.
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    // Register a handler for BLE events.
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}
```

Ci sono due meccanismi che un dispositivo BLE può utilizzare per comunicare con il mondo esterno: broadcasting o connecting. Questi meccanismi sono soggetti alle linee guida GAP (Generic Access Profile). GAP definisce come i

dispositivi abilitati BLE possono rendersi disponibili e come due dispositivi possono comunicare direttamente tra loro. Un dispositivo può unirsi a una rete BLE adottando questi ruoli specificati in GAP:

- Nel broadcasting i dispositivi non devono connettersi esplicitamente tra loro per trasferire i dati. I ruoli nel broadcasting sono due:
  - Broadcaster: un dispositivo che trasmette pacchetti di dati di advertising pubblici
  - Observer: un dispositivo che ascolta i dati nei pacchetti di advertising inviati dal Broadcaster
- In connecting i dispositivi devono connettersi in modo esplicito per trasferire i dati. Anche i ruoli nel connecting sono due:
  - Peripheral: un dispositivo che annuncia la sua presenza in modo che i dispositivi centrali possano stabilire una connessione. Dopo la connessione, le periferiche non trasmettono più dati ad altri dispositivi centrali e rimangono connesse al dispositivo che ha accettato la richiesta di connessione
  - Central: Un dispositivo che avvia una connessione con un dispositivo periferico ascoltando prima i pacchetti di advertising. Un dispositivo centrale può connettersi a molti altri dispositivi periferici. Quando il dispositivo centrale vuole connettersi, invia un pacchetto dati di connessione di richiesta al dispositivo periferico. Se il dispositivo periferico accetta la richiesta dal dispositivo centrale, viene stabilita una connessione.

  Questi ruoli sono più comunemente usati rispetto ai ruoli di broadcasting

```
/**@brief Function for the GAP initialization.
*
* @details This function sets up all the necessary GAP (Generic Access Profile) parameters of the
*          device including the device name, appearance, and the preferred connection parameters.
*/
static void gap_params_init(void)
{
    ret_code_t              err_code;
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

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

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

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

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

Quando due dispositivi sono connessi fra loro, entra in gioco la linea guida GATT (Generic Attribute Profile) che descrive in dettaglio come vengono trasferiti i dati una volta che i dispositivi hanno una connessione dedicata. Simile a GAP, ci sono alcuni ruoli che i due dispositivi possono adottare:

- Client: in genere invia una richiesta al server GATT. Il client può leggere e/o scrivere dati trovati nel server
- Server: uno dei ruoli principali del server è memorizzare i dati. Una volta che il client effettua una richiesta, il server deve renderli disponibili.

```
/**@brief Function for initializing the GATT module.*/
static void gatt_init(void)
{
    ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for initializing the Advertising functionality.*/
static void advertising_init (void)
{
    ble_advertising_init_t init;
    memset(&init, 0, sizeof(init));

    init.advdata.name_type               = BLE_ADVDATA_FULL_NAME;
    init.advdata.include_appearance      = true;
    init.advdata.flags                   = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
    init.advdata.uuids_complete.uuid_cnt = sizeof m_adv_uuids / sizeof m_adv_uuids[0];
    init.advdata.uuids_complete.p_uuids  = m_adv_uuids;

    init.config.ble_adv_fast_enabled  = true;
    init.config.ble_adv_fast_interval = 244;   // in units of  0.625 ms.
    init.config.ble_adv_fast_timeout  = 3000;  // in units of 10.000 ms.
    init.config.ble_adv_slow_enabled  = true;
    init.config.ble_adv_slow_interval = 2056;  // in units of  0.625 ms.
    init.config.ble_adv_slow_timeout  = 3000;  // in units of 10.000 ms.

    init.evt_handler = on_adv_evt;

    ret_code_t err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);

    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}

/**@brief Function for initializing services that will be used by the application.*/
static void services_init (void)
{
    ret_code_t err_code;

    // Initialize Queued Write Module.

    nrf_ble_qwr_init_t qwr_init;
    memset(&qwr_init, 0, sizeof qwr_init);

    qwr_init.error_handler = nrf_qwr_error_handler;

    err_code = nrf_ble_qwr_init(&m_qwr, &qwr_init);
    APP_ERROR_CHECK(err_code);
```

```
    // Initialize Battery Service.
    ble_bas_init_t bas_init;
    memset(&bas_init, 0, sizeof bas_init);

    bas_init.evt_handler          = NULL;
    bas_init.support_notification = true;
    bas_init.p_report_ref         = NULL;
    bas_init.initial_batt_level   = 0;
    bas_init.bl_rd_sec        = SEC_OPEN;
    bas_init.bl_cccd_wr_sec   = SEC_OPEN;
    bas_init.bl_report_rd_sec = SEC_OPEN;

    err_code = ble_bas_init(&m_bas, &bas_init);
    APP_ERROR_CHECK(err_code);

    // Initialize Device Information Service.
    ble_dis_init_t dis_init;
    memset(&dis_init, 0, sizeof dis_init);

    ble_srv_ascii_to_utf8(&dis_init.manufact_name_str, (char *)MANUFACTURER_NAME);
    dis_init.dis_char_rd_sec = SEC_OPEN;

    err_code = ble_dis_init(&dis_init);
    APP_ERROR_CHECK(err_code);

}

/**@brief Function for initializing the Connection Parameters module.*/
static void conn_params_init(void)
{
    ret_code_t             err_code;
    ble_conn_params_init_t cp_init;

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

    cp_init.p_conn_params                  = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail             = false;
    cp_init.evt_handler                    = on_conn_params_evt;
    cp_init.error_handler                  = conn_params_error_handler;

    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
}
```

Il Peer Manager può essere utilizzato dalle applicazioni per gestire la sicurezza BLE (crittografia, associazione e collegamento). Utilizza l'archiviazione flash per archiviare in modo persistente le informazioni sul legame e i dati GATT per ogni dispositivo peer con cui è collegato. Per far si che il nostro dispositivo avvii la procedura di bonding, dall'infocenter della Nordic Semiconductor dobbiamo andare a vedere quali parametri di sicurezza dobbiamo impostare.

```
/**@brief Function for the Peer Manager initialization.*/
static void peer_manager_init(void)
{
    ret_code_t err_code;

    err_code = pm_init();
    APP_ERROR_CHECK(err_code);

    err_code = pm_register(pm_evt_handler);
    APP_ERROR_CHECK(err_code);

    ble_gap_sec_params_t sec_param;
    memset(&sec_param, 0, sizeof(ble_gap_sec_params_t));

    // Security parameters to be used for all security procedures.
    sec_param.bond           = SEC_PARAM_BOND;
    sec_param.mitm           = SEC_PARAM_MITM;
    sec_param.lesc           = SEC_PARAM_LESC;
    sec_param.keypress       = SEC_PARAM_KEYPRESS;
    sec_param.io_caps        = SEC_PARAM_IO_CAPABILITIES;
    sec_param.oob            = SEC_PARAM_OOB;
    sec_param.min_key_size   = SEC_PARAM_MIN_KEY_SIZE;
    sec_param.max_key_size   = SEC_PARAM_MAX_KEY_SIZE;
    sec_param.kdist_own.enc  = 1;
    sec_param.kdist_own.id   = 1;
    sec_param.kdist_peer.enc = 1;
    sec_param.kdist_peer.id  = 1;

    err_code = pm_sec_params_set(&sec_param);
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for starting advertising.*/
void advertising_start (void)
{
    if (!m_advertising_active) {
        ret_code_t err_code = ble_advertising_start(&m_advertising, BLE_ADV_MODE_FAST);
        APP_ERROR_CHECK(err_code);
        m_advertising_active = true;
    }
}
```

5.4 USB events

All’interno del codice possiamo andare a gestire due tipologie di eventi legati all’USB: quelli relativi alla parte di potenza, in cui dovremmo solo andare a controllare se alla porta USB sta arrivando la giusta alimentazione, ed agire di conseguenza (cioè disattivare l’USB in caso non ci fosse potenza), e la parte di lettura e scrittura. In quest’ultima parte dovremmo andare a coordinare, tramite l’evento “APP_USBD_CDC_ACM_USER_EVT_RX_DONE”, le richieste dell’utente identificate dalla ricezione di specifici caratteri precedentemente concordati.

I caratteri e le relative richieste sono i seguenti:

- ‘2’: inizio procedura di advertising
- ‘3’: interruzione di advertising
- ‘5’: invio del Bluetooth address del microcontrollore
- ‘D’: cancellazione di tutti i peer nel microcontrollore
- ‘L’: invio del Bluetooth address di tutti i peer nel microcontrollore
- ‘R’: hardware reset

```
static void usbd_user_ev_handler (app_usbd_event_type_t event)
{
    switch (event)
    {
        case APP_USBD_EVT_DRV_SUSPEND:
            NRF_LOG_INFO("USB_SUSPEND");
            usb_power_status.usb_active = false;
            break;
        case APP_USBD_EVT_DRV_RESUME:
            NRF_LOG_INFO("USB_RESUME");
            usb_power_status.usb_active = true;
            break;
        case APP_USBD_EVT_STARTED:
            break;
        case APP_USBD_EVT_STOPPED:
            app_usbd_disable();
            break;
        case APP_USBD_EVT_POWER_DETECTED:
            NRF_LOG_INFO("USB power detected");
            usb_power_status.usb_active   = false;
            usb_power_status.usb_powered  = true;
            usb_power_status.usb_response = false;
            if (!nrf_drv_usbd_is_enabled())
            {
                app_usbd_enable();
            }
            break;
        case APP_USBD_EVT_POWER_REMOVED:
            NRF_LOG_INFO("USB power removed");
            usb_power_status.usb_active   = false;
            usb_power_status.usb_powered  = false;
            usb_power_status.usb_response = false;
            app_usbd_stop();
            break;
        case APP_USBD_EVT_POWER_READY:
            NRF_LOG_INFO("USB ready");
            app_usbd_start();
            break;
        case APP_USBD_EVT_DRV_RESET:
            NRF_LOG_INFO("USB reset");
            usb_power_status.usb_response = true;
            break;
        default:
            break;
    }
}
```

```
static void cdc_acm_user_ev_handler (app_usbd_class_inst_t const * p_inst, app_usbd_cdc_acm_user_event_t event)
{
    app_usbd_cdc_acm_t const *p_cdc_acm = app_usbd_cdc_acm_class_get(p_inst);

    switch (event)
    {
        case APP_USBD_CDC_ACM_USER_EVT_PORT_OPEN:
        {
            NRF_LOG_INFO("CDC_ACM_OPEN");

            /*Setup first transfer*/
            ret_code_t ret = app_usbd_cdc_acm_read(&m_app_cdc_acm,
                                                   m_rx_buffer,
                                                   READ_SIZE);
            UNUSED_VARIABLE(ret);
            break;
        }
        case APP_USBD_CDC_ACM_USER_EVT_PORT_CLOSE:
            NRF_LOG_INFO("CDC_ACM_CLOSE");
            break;
        case APP_USBD_CDC_ACM_USER_EVT_TX_DONE:
            break;
        case APP_USBD_CDC_ACM_USER_EVT_RX_DONE:
        {
            ret_code_t ret;
            NRF_LOG_INFO("Bytes waiting: %d", app_usbd_cdc_acm_bytes_stored(p_cdc_acm));
            do
            {
                /*Get amount of data transfered*/
                size_t size = app_usbd_cdc_acm_rx_size(p_cdc_acm);
                NRF_LOG_INFO("RX: size: %lu char: %c", size, m_rx_buffer[0]);
                switch (m_rx_buffer[0]) {
                    case 2: // STX
                        advertising_start(false);
                        break;
                    case 3: // ETX
                        advertising_stop();
                        break;
                    case 5: // ENQ
                        bt_list_info();
                        break;
                    case 'D':
                        bt_clear_bonds();
                        break;
                    case 'L':
                        bt_list_bonds();
                        break;
                    case 'R':
                        // hardware reset:
                        NRF_GPIO->OUTCLR = 1 << RESET_OUT;
                        NRF_GPIO->DIRSET = 1 << RESET_OUT;
                        break;
                }
                /* Fetch data until internal buffer is empty */
                ret = app_usbd_cdc_acm_read(&m_app_cdc_acm,
                                            m_rx_buffer,
                                            READ_SIZE);
            } while (ret == NRF_SUCCESS);

            break;
        }
        default:
            break;
    }
}

void advertising_stop (void)
{
    m_advertising_active = false;
    sd_ble_gap_adv_stop(m_advertising.adv_handle);
}
```

```
void bt_list_info (void)
{
    ble_gap_addr_t ble_addr;
    uint32_t err_code = pm_id_addr_get(&ble_addr);
    // "pm_id_addr_get" is a Peer Manager function
    APP_ERROR_CHECK(err_code);
    send_ble_address("PM BT address: ", ble_addr);

    // Instead of a Peer Manager function, we can use a GAP function
    err_code = sd_ble_gap_addr_get(&ble_addr);
    // "sd_ble_gap_addr_get" is a GAP function
    APP_ERROR_CHECK(err_code);
    send_ble_address("SD BT address: ", ble_addr);
}

void bt_clear_bonds (void)
{

    ret_code_t err_code = pm_peers_delete();
    APP_ERROR_CHECK(err_code);

}

void bt_list_bonds (void)
{
    int num_peers = 0;
    for (
        pm_peer_id_t current_peer_id = PM_PEER_ID_INVALID;
        (current_peer_id = pm_next_peer_id_get(current_peer_id)) != PM_PEER_ID_INVALID;
        ++num_peers
    ) {
        pm_peer_data_bonding_t peer_data;
        uint32_t err_code = pm_peer_data_bonding_load(current_peer_id, &peer_data);
        APP_ERROR_CHECK(err_code);
        size_t size = sprintf(tx_buffer, "Peer ID=%04X ", (unsigned) current_peer_id);
        usb_send(tx_buffer, size);
        send_ble_address("BT=", peer_data.peer_ble_id.id_addr_info);
    }

    if (!num_peers) {
        size_t size = sprintf(tx_buffer, "There are no peers\r\n");
        usb_send(tx_buffer, size);
    }

}
```

Per completezza, aggiungo anche la funzione per inviare i dati tramite USB da microcontrollore:

```
int usb_send (const char *data, size_t size)
{
    int ret = app_usbd_cdc_acm_write(&m_app_cdc_acm, data, size);
    return ret;
}
```

5.5 Bluetooth events

Lo scopo della funzione della gestione degli eventi del Bluetooth è quello di rispondere alle richieste di un dispositivo esterno di accoppiarsi (tramite bonding in questo caso) con il nostro microcontrollore. Gli eventi che vengono creati per questo processo sono “BLE_GAP_EVT_SEC_PARAMS_REQUEST” e “BLE_GATTS_EVT_SYS_ATTR_MISSING”. Però possiamo usare un “trucchetto” per inviare i parametri di bonding senza dover rispondere a questi eventi. Quando questi due eventi sono mancanti nel codice creato dal programmatore, il microcontrollore risponderà di default con i parametri presenti all’interno del Peer Manager. Se noi impostiamo i parametri del Peer Manager per connettersi con altri dispositivi nel modo in cui vogliamo (in questo caso come bonding, come è stato fatto precedentemente), il microcontrollore eseguirà in autonomia tutte le varie procedure.

```
/**@brief Function for handling BLE events.
*
* @param[in]   p_ble_evt   Bluetooth stack event.
* @param[in]   p_context   Unused.
*/
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    ret_code_t err_code = NRF_SUCCESS;

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_DISCONNECTED:
            NRF_LOG_INFO("Disconnected.");
            // LED indication will be changed when advertising starts.
            sd_clock_hfclk_release();
            break;

        case BLE_GAP_EVT_CONNECTED:
            NRF_LOG_INFO("Connected.");
            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
            err_code = nrf_ble_qwr_conn_handle_assign(&m_qwr, m_conn_handle);
            APP_ERROR_CHECK(err_code);
        {
            ble_gap_phys_t const phys = {
                .rx_phys = BLE_GAP_PHY_2MBPS,
                .tx_phys = BLE_GAP_PHY_2MBPS,
            };
        sd_ble_gap_phy_update(m_conn_handle, &phys);
        }
            break;

        case BLE_GAP_EVT_PHY_UPDATE:
        {
            NRF_LOG_INFO("PHY: %d %d %d",
                    p_ble_evt->evt.gap_evt.params.phy_update.status,
                    p_ble_evt->evt.gap_evt.params.phy_update.rx_phy,
                    p_ble_evt->evt.gap_evt.params.phy_update.tx_phy
            );
        } break;
```

```
        case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
        {
            NRF_LOG_INFO("PHY update request.");
            ble_gap_phys_t const phys =
            {
                .rx_phys = BLE_GAP_PHY_AUTO,
                .tx_phys = BLE_GAP_PHY_AUTO,
            };
            err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GATTC_EVT_TIMEOUT:
            // Disconnect on GATT Client timeout event.
            NRF_LOG_DEBUG("GATT Client Timeout.");
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
                                             BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_GATTS_EVT_TIMEOUT:
            // Disconnect on GATT Server timeout event.
            NRF_LOG_DEBUG("GATT Server Timeout.");
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
                                             BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_GATTS_EVT_HVN_TX_COMPLETE:
            break;

        default:
            // No implementation needed.
            break;
    }
}
```

5.6 Altri eventi

Oltre agli eventi riguardanti l'USB e il Bluetooth, possiamo gestire anche quegli eventi che scaturiscono da quei registri che abbiamo precedentemente inizializzato, come ad esempio il registro per l'advertising.

```
/**@brief Callback function for asserts in the SoftDevice.
*
* @details This function will be called in case of an assert in the SoftDevice.
*
* @warning This handler is an example only and does not fit a final product. You need to analyze
*          how your product is supposed to react in case of Assert.
* @warning On assert from the SoftDevice, the system can only recover on reset.
*
* @param[in] line_num   Line number of the failing ASSERT call.
* @param[in] file_name  File name of the failing ASSERT call.
*/
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
    app_error_handler(DEAD_BEEF, line_num, p_file_name);
}
```

```
/**@brief Function for handling Peer Manager events.
*
* @param[in] p_evt  Peer Manager event.
*/
static void pm_evt_handler(pm_evt_t const * p_evt)
{
    pm_handler_on_pm_evt(p_evt);
    pm_handler_flash_clean(p_evt);

    switch (p_evt->evt_id)
    {
        case PM_EVT_PEERS_DELETE_SUCCEEDED:
            usb_send("Peers deleted\r\n", 16);
            break;

        default:
            break;
    }
}

/**@brief Function for handling Queued Write Module errors.
*
* @details A pointer to this function will be passed to each service which may need to inform the
*          application about an error.
*
* @param[in]   nrf_error   Error code containing information about what went wrong.
*/
static void nrf_qwr_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for handling the Connection Parameters Module.
*
* @details This function will be called for all events in the Connection Parameters Module which
*          are passed to the application.
*          @note All this function does is to disconnect. This could have been done by simply
*                setting the disconnect_on_fail config parameter, but instead we use the event
*                handler mechanism to demonstrate its use.
*
* @param[in] p_evt  Event received from the Connection Parameters Module.
*/
static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
{
    ret_code_t err_code;

    if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
    {
        err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UNACCEPTABLE);
        APP_ERROR_CHECK(err_code);
    }
}

/**@brief Function for handling a Connection Parameters error.
*
* @param[in] nrf_error  Error code containing information about what went wrong.
*/
static void conn_params_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}
```

```
/**@brief Function for handling advertising events.
*
* @details This function will be called for advertising events which are passed to the application.
*
* @param[in] ble_adv_evt  Advertising event.
*/
static void on_adv_evt (ble_adv_evt_t ble_adv_evt)
{
    switch (ble_adv_evt) {
    case BLE_ADV_EVT_FAST:
        NRF_LOG_INFO("Fast advertising.");
        break;
    case BLE_ADV_EVT_SLOW:
        NRF_LOG_INFO("Slow advertising.");
        break;
    case BLE_ADV_EVT_IDLE:
        NRF_LOG_INFO("Idle advertising.");
        if (m_advertising_active)
            ble_advertising_start(&m_advertising, BLE_ADV_EVT_SLOW);
        else
            sleep_mode_enter();
        break;
    default:
        break;
    }
}


/**@brief Function for putting the chip into sleep mode.
*
* @note This function will not return.
*/
static void sleep_mode_enter( void)
{
    ret_code_t err_code;

    // Go to system-off mode (this function will not return; wakeup will cause a reset).
    err_code = sd_power_system_off();
    APP_ERROR_CHECK(err_code);
}
```

6) Conclusioni

Testando il codice è emerso che durante l'invio tramite USB dal microcontrollore al PC dei Bluetooth address relativi ai peers, venivano persi dei pacchetti dati. La causa è, molto probabilmente, la chiamata della funzione che inizia la scrittura su microcontrollore. Se non si aspetta del tempo tra due chiamate successive, i dati della seconda chiamata possono sovrascrivere la prima, causando errori.
In secondo luogo, sarebbe opportuno inserire un PIN quando si inizia la procedura di bonding, dato che in assenza di ciò chiunque potrebbe connettersi. In più, si potrebbe aggiungere un modo per cambiare il PIN attraverso l'USB, dato che l'unica modalità con cui viene implementato all'inizio è attraverso un PIN statico che può essere cambiato solo mettendo di nuovo mano al codice.
Provando ad implementare queste procedure, il software non funzionava. Data la mancanza di tempo, non si è potuto correggere l'errore nel nuovo software.

7) Appendice

- nRF52840 SoC:
  https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840

- nRF52840 DK:
  https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52840-DK

- Realizzazione di un'interfaccia universale in tecnologia Bluetooth Low Energy per sensori integrati, Prof. Mirko Viroli, Piero Biagini:
  https://amslaurea.unibo.it/9241/1/Tesi.pdf

- The Basics of Bluetooth Low Energy (BLE)
  https://www.novelbits.io/basics-bluetooth-low-energy/

- Bluetooth Protocol (Part 1): Basics and Working:
  https://www.engineersgarage.com/tech-articles/bluetooth-protocol-part-1-basics-and-working/

- Bluetooth Protocol (Part 2): Types, Data Exchange, Security:
  https://www.engineersgarage.com/tech-articles/bluetooth-protocol-part-2-types-data-exchange-security/

- Bluetooth Basics:
  https://learn.sparkfun.com/tutorials/bluetooth-basics/all

- A Basic Introduction to BLE Security:
  https://www.digikey.com/eewiki/display/Wireless/A+Basic+Introduction+to+BLE+Security

- What is Bluetooth Address (BD_ADDR):
  https://macaddresschanger.com/what-is-bluetooth-address-BD_ADDR

- USB Operation: Protocol, Data Transfer & Packets:
  https://www.electronics-notes.com/articles/connectivity/usb-universal-serial-bus/protocol-data-transfer.php

- USB Concepts:
  https://www.keil.com/pack/doc/mw/USB/html/_u_s_b__concepts.html

- Embedded USB - a brief tutorial:
  https://computer-solutions.co.uk/info/Embedded_tutorials/usb_tutorial.htm

- Nordic Semiconductor Infocenter:
  https://infocenter.nordicsemi.com/index.jsp