UNIVERSITÀ POLITECNICA DELLE MARCHE
LAUREA TRIENNALE IN INGEGNERIA ELETTRONICA

TESI

Firmware per l'acquisizione e la trasmissione su reti Bluetooth LE di dati registrati da sensori indossabili

Firmware for acquiring and transmitting data from wearable sensors over Bluetooth LE networks

Studentessa
Giada ALIFFI

Relatore
Giorgio BIAGETTI

Correlatore
Paolo CRIPPA

ANNO ACCADEMICO 2018/2019

Indice

1 Introduzione

1.1 Obiettivo

L'obiettivo della presente tesi è quello di sviluppare un firmware in grado di effettuare l'acquisizione di segnali elettromiografici da sensori indossabili e la loro successiva trasmissione tramite il bluetooth low energy 5 (BLE 5). La necessità che ha portato allo sviluppo di tale progetto è quella di monitorare i movimenti di persone non autosufficienti, soprattutto anziani affetti da morbo di Alzheimer o Parkinson. In questo contesto, occorre individuare eventuali comportamenti anomali, potenzialmente pericolosi per il soggetto, in modo da prevenire danni.

1.2 Materiale utilizzato

1.2.1 Hardware

Per la prima parte del progetto, si è scelto di utilizzare il microcontrollore a 32-bit NRF52840 [1] della Nordic Semiconductor, dotato di processore ARM-CORTEX-M4F, data memory(RAM) da 256 kB e program-memory(ROM:FLASH) da 1 MB.
In questo contesto si è scelto di utilizzarlo montato sulla demoboard NRF52840 PDK (alias PCA10056) la quale supporta trasmissioni radio via BLE5.
Per l'acquisizione delle tensioni elettriche, inoltre, è stato impiegato il sensore INA226 della Texas Instruments, dotato d' interfaccia IIC.

Per la seconda parte del progetto, invece, la demoboard è stata abbandonata in favore di una nuova scheda, realizzata nel dipartimento di ingegneria dell'informazione dell'UNIVPM. Chiaramente, il microcontrollore inserito in quest'ultima è lo stesso di quello presente nella demoboard, ma cambiano le periferiche collegate ad esso. In particolare, per l'acquisizione dei dati, si è utilizzato il dispositivo ADS1293 della Texas Instruments, dotato d'interfaccia SPI.

1.2.2 Software

Per la parte software, invece, si è sfruttato l'SDK(Software Development kit)[2] disponibile sul sito della Nordic Semiconductor, al seguente indirizzo:
https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52840-DK/Download#infotabs
contenente tutte le librerie, gli header e i file di esempio necessari per un facile utilizzo della board.

Oltre a quest'ultimo, si è impiegato il SoftDevice S140 [3], ovvero uno stack Bluetooth 5 completo per l' NRF52840, disponibile anch'esso sul sito della Nordic Semiconductor, al seguente indirizzo:
https://www.nordicsemi.com/Software-and-Tools/Software/S140/Download#infotabs

Dovendo compilare programmi su un elaboratore con architettura diversa da quella del microcontrollore, si è dovuto installare anche un cross-compilatore, `arm-none-eabi-gcc`. Quest'ultimo è contenuto in un pacchetto chiamato gcc-arm-embedded [4], che si ottiene digitando su un terminale Linux i seguenti tre comandi, uno alla volta:

```
sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
sudo apt-get update
sudo apt-get install gcc-arm-embedded
```

In tale pacchetto sono presenti molti altri componenti tra cui il debugger `arm-none-eabi-gdb`. Ottenuto quest'ultimo, per debuggare programmi caricati sul microcontrollore, occorre potersi connettere al software SEGGER J-LINK [5] scaricabile dal sito:

https://segger.com/downloads/jlink

Infine,per caricare i programmi sul microcontrollore, ci sono diverse possibilità. Per esempio, è possibile sfruttare il programma `JFlashLite` contenuto nel pacchetto precedente.

2 Implementazione

2.1 Acquisizione

2.1.1 Breve descrizione del sensore INA226

Il sensore INA226 [6] della Texas Instruments è principalmente un dispositivo atto al monitoraggio della corrente che attraversa un carico (bipolo) e della potenza entrante nello stesso. A tal fine, esso effettua il rilevamento della caduta di potenziale ai capi di un resistore shunt noto e quella tra un bus (morsetto positivo del bipolo) e la massa.
In particolare, l' INA226 è in grado rilevare la corrente su tensioni di bus di modo comune che possono variare da 0 V a 36 V, indipendentemente dalla tensione di alimentazione.
Quest'ultima è normalmente compresa tra 2,7 e 5,5 V. Il consumo di corrente, invece, è tipicamente 330 μA.
Il funzionamento corretto del dispositivo è garantito in un range di temperatura tra $-40°$ e i $125°$.

Il suo schema elettrico di principio è mostrato di seguito:

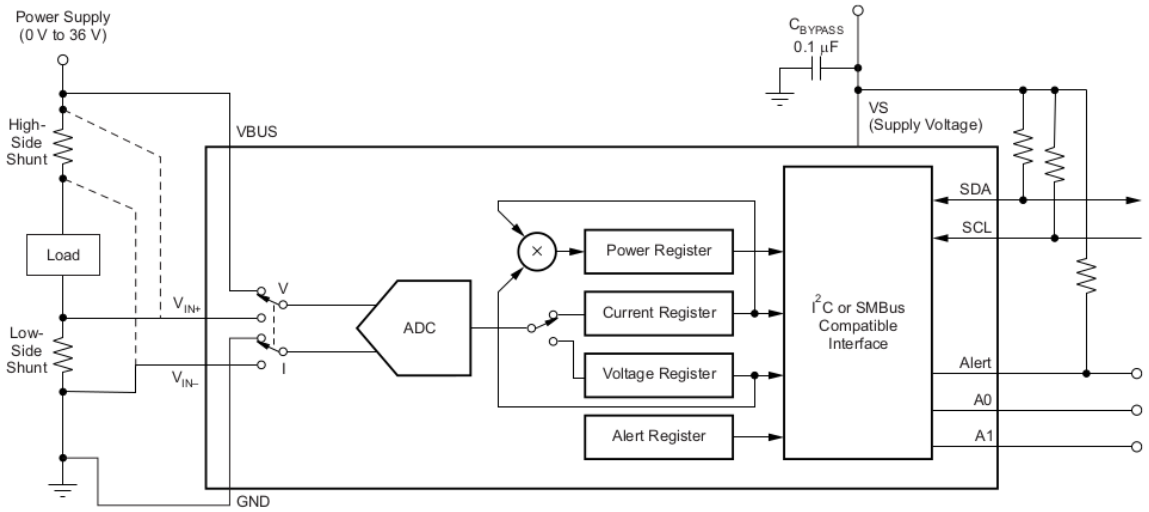

Figura 1: schema elettrico di principio dell'INA226

Come si può notare, i dati acquisiti vengono poi convertiti mediante un ADC di tipologia sigma-delta(avente una capacità intrinseca di reiezione ai disturbi), salvati in opportuni registri e poi, eventualmente, trasferiti mediante un interfaccia di tipo IIC. Quest'ultima, all'occorrenza, è dotata di 16 indirizzi programmabili.

2.1.2 Configurazione dei registri del sensore INA226

Il dispositivo INA226, per effettuare l'acquisizione di tensione, necessita due azioni fondamentali:

1. configurazione del **CONFIGURATION REGISTER** (registro 0x00) tramite la quale si decidono le impostazioni base di funzionamento. Nel caso corrente, si è deciso per le seguenti opzioni:

    - reset iniziale del sistema;
    - modalità bus continous → in modo da prelevare tensioni continuamente dal pin VBUS;
    - Ogni 16 valori prelevati e convertiti, viene effettuata una media;
    - Il tempo di conversione di ciascuno di questi valori è di 332 μs.

2. configurazione del **MASK-ENABLE REGISTER** (registro 0x06) tramite il quale è possibile attivare il cosiddetto "Alert Pin", in modo da ricevere un avviso ogni volta che è pronto un nuovo risultato nel **BUS VOLTAGE REGISTER**(registro 0x02);

Di seguito è mostrata la porzione di codice inserita nella funzione main per l'implementazione di quanto detto sopra, sfruttando l'interfaccia IIC dell'INA226:

```
//lista registri utili INA226 e relativi indirizzi
uint8_t configuration_reg_pointer = 0x00;
uint8_t bus_voltage_reg_pointer = 0x02;
uint8_t mask_enable_reg_pointer = 0x06;

//1 step --> CONFIGURAZIONE DEL CONFIGURATION REGISTER
//configurazione del TX BUFFER NRF52
uint8_t buffer0 [3];
size_t length0 = 3;
buffer0[0] = configuration_reg_pointer;
//system reset (gli altri bit non sono significativi)
buffer0[1] = 0b10000000;
buffer0[2] = 0b00000000;
transmission(buffer0,length0);

//reset off, 16 collected value, conversion time bus voltage = 332 us x 1024,
 ↪  mode: bus continous
buffer0[1] = 0b01000100;
buffer0[2] = 0b10100110;
transmission(buffer0,length0);

//2 step --> CONFIGURAZIONE DELL'ALERT PIN PER ATTIVAZIONE DELLA FUNZIONE
 ↪  "CONVERSION READY"
uint8_t buffer6 [3];
size_t length6 = 3;
buffer6[0] = mask_enable_reg_pointer;
buffer6[1] = 0b00000100;
buffer6[2] = 0b00000000;
transmission(buffer6,length6);
```

Come si può notare, nelle precedenti righe, sono stati dichiarati e inizializzati due array, buffer0 e buffer6 (il numero finale che compare nel nome sta a richiamare l'indirizzo del registro da scrivere) necessari come buffer di trasmissione per l'interfaccia IIC, utilizzata per configurare l'INA226. In particolare, la comunicazione avviene volta per volta mediante la funzione transmission(), la quale sarà descritta più nel dettaglio in seguito alla presentazione della periferica IIC(o TWIM).

2.1.3 Prelievo dei dati tramite sensore INA226

Fatto ciò, per poter iniziare ad effettuare ricezioni, occorre eseguire una scrittura fittizia del registro su cui verranno salvati i risultati delle conversioni, seguita dal settaggio del buffer di ricezione; in questo caso, un array di 2 uint8_t chiamato bus_voltage_val:

```
//3 step --> PREPARAZIONE ALLA LETTURA DELLA BUS VOLTAGE
   //scrittura fittizia su bus_voltage register per cambiare registro con cui
    ↪ comunicare tramite IIC
   change_reg2read(bus_voltage_reg_pointer);

   //configurazione buffer di ricezione IIC
   nrf_twim_rx_buffer_set(mytwim,(uint8_t
    ↪ *)bus_voltage_val,(size_t)length_bus_voltage_val);//scelgo dove scrivere
    ↪ il risultato della ricezione
```

La periferica TWIM, per funzionare, necessita di alcune configurazioni, o meglio:

1. settaggio di un paio di pin disponibili (nel caso corrente P0-26 e P0-27) come input e, quindi, per svolgere la funzionalità di SDA e SCL;
2. scelta della frequenza di clock, qui impostata come 100 kbps;
3. abilitazione degli interrupt su uno o più eventi, qui a fine comunicazione(trasmissione e ricezione)e in caso di errore;
4. abilitazione di shortcut utili al collegamento causa-effetto di un evento con un task, qui usate per mandare automaticamente la stop condition nel momento in cui si sta iniziando la trasmissione o ricezione dell'ultimo byte;
5. specificazione dell'indirizzo dello slave, qui pari a quello di default dell'INA226,ovvero 0x40;
6. abilitazione vera e propria della periferica.

Quanto detto è implementato con il seguente codice:

```
/*CONFIGURAZIONE PERIFERICA TWIM*/

   //settaggio dei pin 26 e 27 come SDA e SCL
    nrf_twim_pins_set(mytwim,27,26);

   //configurazione pin SDA e SCL come INPUT --> IIC in modalità MASTER (TWIM)
   nrf_gpio_cfg_input(26,NRF_GPIO_PIN_NOPULL);//SDA input
   nrf_gpio_cfg_input(27,NRF_GPIO_PIN_NOPULL); //SCL input

   //scelta della frequenza del clock
   nrf_twim_frequency_set(mytwim,NRF_TWIM_FREQ_100K); //100 kbps

   //shortcut utile per mandare lo stop appena inizia la trasmissione dell'ultimo
    ↪ byte
   nrf_twim_shorts_enable(mytwim,NRF_TWIM_SHORT_LASTTX_STOP_MASK);

   //shortcut utile per mandare lo stop appena inizia la ricezione dell'ultimo
    ↪ byte
   nrf_twim_shorts_enable(mytwim,NRF_TWIM_SHORT_LASTRX_STOP_MASK);
```

```
//abilitazione degli interrupt TWIM
nrf_twim_int_enable(mytwim,NRF_TWIM_INT_STOPPED_MASK);//interrupt fine
 ↪  trasmissione o ricezione
nrf_twim_int_enable(mytwim,NRF_TWIM_INT_ERROR_MASK);//interrupt su errore -->
 ↪  nack dopo trasmissione di address o data

//abilitazione periferica TWIM
nrf_twim_enable(mytwim);

//impostazione dello slave address (INA226)
uint8_t slave_address = 0x40; //A0 e A1 sono posti a GND (vedi datasheet
 ↪  INA226)
nrf_twim_address_set(mytwim,slave_address);
```

In particolare, volendo descrivere brevemente il meccanismo di acquisizione dati mediante la periferica TWIM, è possibile affermare che, in caso le trasmissioni per la configurazione dei registri dell'INA226 vadano a buon fine (10 tentativi a disposizione), non appena sarà pronto un risultato all'interno del BUS VOLTAGE REGISTER (registro 0x02) dell'INA226, l'Alert Pin (nel caso corrente collegato alla gpio P0-3) si porterà basso e tale evento innescherà l'inizio di una nuova ricezione IIC.
Questo meccanismo di causa-effetto è permesso grazie a due periferiche particolari dell'NRF528540, ovvero:

1. **GPIOTE**, la quale consente di generare un evento in caso di un particolare cambiamento di stato di una gpio, in questo caso sul fronte di discesa dell'Alert Pin;

2. **PPI(PROGRAMMABLE PERIPHERAL INTERCONNECT)**, la quale è costituita da una rete di canali, ciascuno dei quali consente di porre in comunicazione due periferiche, senza bisogno dell'intervento della CPU. Nel caso corrente, GPIOTE e TWIM.

Queste ultime sono configurabili, per lo scopo corrente, con le seguenti righe:

```
//configurazioni periferica GPIOTE
   nrf_gpio_cfg_input(3,NRF_GPIO_PIN_NOPULL);//configuro l'alert pin come input
   nrf_gpiote_event_enable(0); //abilito l'evento su GPIOTE channel 0
   nrf_gpiote_event_configure(0, 3, NRF_GPIOTE_POLARITY_HITOLO); //genero un
    ↪  evento quando sull'alert pin ho un fronte di discesa

    nrf_ppi_channel_endpoint_setup(2,0x40006100,0x40004000);  // quando accade un
     ↪  evento su GPIOTE channel 0 --> parte la ricezione IIC
   nrf_ppi_channel_enable (2); // abilito il canale PPI 2
```

Tornando all'IIC, è possibile affermare che sia trasmissione che ricezione sono gestite mediante interrupt.
In particolare, per quanto riguarda la trasmissione, dopo aver invocato la funzione:

```
void transmission(uint8_t * buffer, size_t length)
{
while(nrf_twim_event_check(mytwim,NRF_TWIM_EVENT_TXSTARTED) == 1);
nrf_twim_tx_buffer_set(mytwim,buffer,length);//scelgo i byte da trasmettere
nrf_twim_task_trigger(mytwim,NRF_TWIM_TASK_STARTTX); //innesco lo STARTTX task
}
```

potrà verificarsi (al termine) l'evento NRF_TWIM_EVENT_STOPPED, il quale segnala l'avvenuta fine della comunicazione, oppure (nel mezzo) l'evento NRF_TWIM_EVENT_ERROR, il quale avvisa di eventuali errori che non consentono il normale svolgimento del programma. Un discorso analogo vale per la ricezione, la quale però, come si è già accennato prima, è innescata dal fronte di discesa dell'alert pin, mediante l'azione della periferica PPI.
Al verificarsi di uno dei due eventi suddetti, si entrerà nell'interrupt service routine(ISR) di TWIM1.
All'interno di quest'ultima, come prima cosa, verrà verificata la causa dell'interrupt.
In caso di fine comunicazione, verrà effettuato un ulteriore controllo per capire se ciò che ha avuto termine è stata una trasmissione o una ricezione.
In entrambi i casi, si procederà con i reset dei flag che si sono settati nel corso della comunicazione ma, se c'è stata una ricezione, quest'azione sarà accompagnata da due ulteriori azioni:

1. scrittura della variabile "bus_voltage" con il risultato della conversione. In particolare, tale variabile rappresenta un campo di una struttura denominata data2write del tipo seguente:

```
    typedef struct sd_data {

    volatile float bus_voltage;
    volatile uint32_t millisecondi;
    } my_sd_data;
```

2. settaggio di un flag "reception_complete" in modo da poter gestire le azioni successive alla ricezione fuori dalla ISR,nel main.

In caso di errore, invece, se ne controllerà la natura per poi salvare il valore identificativo nella variabile twim_flag e resettare il flag associato ad esso. In particolare, la variabile twim_flag è di tipo enum ed è definita come segue:

```
enum twim_err {
    NO_ERR = 0,
    ADDRESS_NACK = 1,
    DATA_NACK = 2,
};
```

Per sicurezza, si resetteranno anche i flag relativi ad un eventuale inizio ricezione o trasmissione. Infatti, una volta fuori dalla ISR, occorre evitare l'accadimento di eventi non corretti nel main.

La ISR della periferica TWIM1 è riportata di seguito:

```
void SPIM1_SPIS1_TWIM1_TWIS1_SPI1_TWI1_IRQHandler() // ISR della periferica TWI
{
    volatile uint32_t temp = 0;
    if ( nrf_twim_event_check(mytwim,NRF_TWIM_EVENT_STOPPED) == 1)
    {
      temp=nrf_twim_event_check(mytwim,NRF_TWIM_EVENT_TXSTARTED);
```

```
      if( temp == 1)
      {
              //cancellazione eventi avvenuti affinché possano ripetersi
        nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_TXSTARTED);
        nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_LASTTX);
        nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_STOPPED);

    }
    temp=nrf_twim_event_check(mytwim,NRF_TWIM_EVENT_RXSTARTED);
         if( temp == 1 ) // se è finita la ricezione
         {
            // modifica della variabile bus_voltage con il risultato della nuova
             ↪  conversione
            data2write.bus_voltage =
             ↪  ((float)(bus_voltage_val[0]*256+bus_voltage_val[1])*(0.00125));

            //cancellazione eventi avvenuti affinché possano ripetersi
            nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_RXSTARTED);
            nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_LASTRX);
            nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_STOPPED);
            reception_complete = 1;

        }
    }

    temp=nrf_twim_event_check(mytwim,NRF_TWIM_EVENT_ERROR);
    if ( temp == 1)
    {
     //controllo quale errore c'è stato
     twim_flag = nrf_twim_errorsrc_get_and_clear(mytwim);
     /* per sicurezza resetto eventi inizio trasmissione e ricezione
       perché poi nel main innesco la stop condition che mi riporterà
       nell' interrupt */
     nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_TXSTARTED);
     nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_RXSTARTED);
     nrf_twim_event_clear(mytwim,NRF_TWIM_EVENT_ERROR);
 }
```

Infine, una volta usciti dall'ISR, se c'è stata una ricezione, nel loop principale del main, il flag reception_complete settato porterà alla scrittura del risultato della conversione in una micro SD.
Tale azione sarà seguita da altre utili alla preparazione della periferica TWIM e del dispositivo INA226 per una nuova ricezione, ovvero reset del flag di avvenuta ricezione e di quello dell'alert pin (mediante riconfigurazione del CONFIGURATION REGISTER).
In termini di codice, dunque,si ha:

```
if (reception_complete == 1)
       {
          //scrittura su SD
          data2write.millisecondi = get_rtc();
          fatfs_write(data2write);
          //riconfiguro il registro 0x00 per resettare il flag dell'alert pin
          transmission(buffer0,length0);
          //mi preparo per un'altra ricezione
          change_reg2read(bus_voltage_reg_pointer);
          reception_complete = 0;
      }
```

Come si può notare, prima della chiamata della funzione fatfs_write() atta alla scrittura della SD, la quale verrà approfondita in seguito, viene chiamata un'altra funzione, get_rtc().
Per comprendere l'utilità di questa funzione, occorre riprendere la definizione della struttura data2write del tipo my_sd_data presentata precedentemente.
In questa, oltre al campo "bus_voltage" di tipo float, è presente un altro campo, cioè "millisecondi", che è di tipo uint32_t.
Quest'ultimo serve per associare a ciascuna delle tensioni scritte nella SD anche un'informazione di tipo temporale.
A tale scopo viene impiegata la periferica **RTC (REAL TIME COUNTER)**, la quale è sostanzialmente un timer che sfrutta la sorgente di clock più lenta del microcontrollore, ovvero LFCLK(LOW-FREQUENCY CLOCK SOURCE)a 32.768 kHz. In questo caso, il prescaler è stato fissato a 32 in modo da avere una precisione al millisecondo.
Quanto detto è rappresentato dal seguente codice:

```
 //configurazione periferica RTC
    nrf_rtc_prescaler_set(my_RTC,RTC_PRESCALER);
    nrf_rtc_task_trigger(my_RTC,NRF_RTC_TASK_START);

    uint32_t get_rtc ()
{
  float my_RTC_T = (RTC_PRESCALER+1)/32.768;
  uint32_t my_RTC_reg = nrf_rtc_counter_get(my_RTC);
  uint32_t millisecondi = my_RTC_reg * my_RTC_T;
  return millisecondi;
}
```

Come si può notare, per quanto riguarda LFCLK, non è necessario né selezionare una sorgente né farlo partire.
Il motivo di ciò è che questa periferica viene usata anche dal Soft Device, il quale provvede già alla sua configurazione e alla sua accensione.

Se, invece della fine di una comunicazione, c'è stato un errore e verrà settato twim_flag, risulterà verificata la condizione del seguente if, anch'esso contenuto nel loop principale del main:

```
if (twim_flag != 0)
     {
         //se c'è stato un errore in trasmissione innesco la stop condition
         nrf_twim_task_trigger(mytwim,NRF_TWIM_TASK_STOP);

         if (tx_attempts != 0)//se non ho superato il decimo tentativo
         {
           //diminuisco di un'unità la variabile che indica il numero di
            ↪  tentativi per la trasmissione
           tx_attempts--;
           //riprovo a fare le due configurazioni
           transmission(buffer0,length0);
           transmission(buffer6,length6);
       }
       else
       {
           //riscrivo la variabile tx_attempts
```

```
            tx_attempts = 10;
            //resetto comunque il flag di errore
            twim_flag = 0;
            __WFE();

        }


    }
    else
    {
        nrf_gpio_pin_write(14,0);//accendo il led2 prima di entrare in modalità
         ↪  wait for event
        __WFE();
        nrf_gpio_pin_write(14,1);//spengo il led2 dopo essere uscita dalla
         ↪  modalità wait for event
    }
```

In particolare, se twim_flag risulta diversa da 0 (NO_ERR), viene subito innescata una stop condition che pone fine alla comunicazione in corso. Successivamente, se non sono stati esauriti i 10 tentativi a cui accennato precedentemente, si riproverà ad effettuare nuovamente le trasmissioni IIC che non sono andate a buon fine.
Altrimenti, verrà rimessa a 10 la variabile tx_attempts, sarà azzerato il flag di errore e ci si porrà in modalità wait for event(WFE), esattamente come nel caso in cui non sono avvenuti errori.
Lo spegnimento del LED2 segnalerà l'entrata del microcontrollore in questa condizione di sostanziale risparmio energetico.

2.2 Scrittura su SD

In seguito all'acquisizione di ciascun dato dall'INA226 mediante l'interfaccia IIC, avverrà la scrittura sulla SD.
Per realizzare ciò si è integrato nel programma principale l'esempio "fatfs" presente nell'SDK, cui nome riprende quello del filesystem utilizzato allo scopo, ovvero FAT32 [7].
Purtroppo le funzioni fornite dalla Nordic, che gestiscono l'interfaccia tra hardware e protocollo FATFS, non provvedono ad una bufferizzazione automatica dei dati nel corso della scrittura. In questo contesto, tale aspetto non è stato trattato, ma potrebbe essere uno spunto per uno sviluppo futuro.
Inizialmente l'esempio sopra citato, effettuava le seguenti azioni principali:

1. configurazione dei parametri del dispositivo di storage e della periferica SPI, necessaria per la comunicazione con quest'ultimo;

2. inizializzazione del dispositivo di storage, in modo da tenerlo pronto per un eventuale operazione di lettura o scrittura;

3. creazione una struttura (o oggetto) filesystem contenente i parametri necessari per effettuare le operazioni sui file o sulle directory;

4. apertura di una nuova directory e lettura del suo contenuto completo;

5. creazione di un nuovo file e scrittura di una stringa al suo interno;

Mentre il punto 1 veniva svolto principalmente mediante delle macro indipendenti, i punti successivi venivano tutti realizzati all'interno di una funzione denominata fatfs_example(). Quest'ultima è stata scissa per comodità in due funzioni: la prima fatfs_config(), per i punti 2 e 3, (eliminando la parte finale di lettura della directory) e la seconda fatfs_write, per il punto 5, modificata in base alle necessità del programma corrente.
Di seguito, sono riportate le funzioni di cui sopra:

```
static void fatfs_config()
{

    disk_state = STA_NOINIT;
    // Initialize FATFS disk I/O interface by providing the block device.
    static diskio_blkdev_t drives[] =
    {
        DISKIO_BLOCKDEV_CONFIG(NRF_BLOCKDEV_BASE_ADDR(m_block_dev_sdc,
         ↪  block_dev), NULL)

    };

    diskio_blockdev_register(drives, ARRAY_SIZE(drives));


    NRF_LOG_INFO("Initializing disk 0 (SDC)...");
    for (uint32_t retries = 3; retries && disk_state; --retries)//"retries"
     ↪  indica il numero di tentativi per l'inizializzazione
    {
        disk_state = disk_initialize(0); // se l'inizializzazione va a buon fine
         ↪  il flag STA_NOINIT si resetta
        // il parametro passato a disk_initialize è sempre "0" se si ha una sola
         ↪  sd card
    }
    if (disk_state) // se l'evento inizializzazione non va a buon fine, viene
     ↪  segnalato
    {
        NRF_LOG_INFO("Disk initialization failed.");
        return; // ha la funzione di "break" per uscire dalla procedura
         ↪  fatfs_example() in caso di fallimento
    }

    uint32_t blocks_per_mb = (1024uL * 1024uL) /
     ↪  m_block_dev_sdc.block_dev.p_ops->
     ↪  geometry(&m_block_dev_sdc.block_dev)->blk_size;
    uint32_t capacity = m_block_dev_sdc.block_dev.p_ops->
     ↪  geometry(&m_block_dev_sdc.block_dev)->blk_count / blocks_per_mb;
    NRF_LOG_INFO("Capacity: %d MB", capacity);

    NRF_LOG_INFO("Mounting volume...");
    ff_result = f_mount(&fs, "", 1);
    /*il FatFs richiede per ogni "logical drive" (FAT volume) una "working area"
     ↪  (flesystem object).
      Quest'ultimo va,dunque, registrato attraverso la funzione "f_mount" al
↪  logical drive.
      La funzione prende in ingresso: un puntatore al filesystem object, un
↪  puntatore al "logical drive"
      (se non viene specificato nulla s'intende quello di default), opzione per
↪  il mounting
```

```
      ( 0 se non lo si vuole montare ora ma al primo accesso, 1 se lo si vuole
↪  montare adesso
      per vedere se è pronto per lavorare)*/
    if (ff_result)
    {
        NRF_LOG_INFO("Mount failed."); // se "f_mount" non ritorna 0 ovvero FR_OK
         ↪  c'è stato qualche errore
        return; // ha la funzione di "break" per uscire dalla procedura
         ↪  fatfs_example() in caso di fallimento
    }

    NRF_LOG_INFO("\r\n Listing directory: /");
    ff_result = f_opendir(&dir, "/");
    /* la funzione "f_opendir" apre una directory e prende in ingresso: un
     ↪  puntatore ad una directory
       non ancora esistente con l'obiettivo di crearne una nuova, puntatore al
↪  nome della directory
       da aprire. */
    if (ff_result)
    {
        NRF_LOG_INFO("Directory listing failed!");// se il listaggio della
         ↪  directory fallisce si notifica tale evento
        return; // ha la funzione di "break" per uscire dalla procedura
         ↪  fatfs_example() in caso di fallimento
    }
    return;
}

static void fatfs_write(my_sd_data data2write)
{
    NRF_LOG_INFO("Writing to file " FILE_NAME "...");
    ff_result = f_open(&file, FILE_NAME, FA_READ | FA_WRITE | FA_OPEN_APPEND);
    /* la funzione "f_open" apre un file, dunque, prenderà in ingresso: un
     ↪  puntatore ad un "file object structure" vuoto
       (ovvero un identificatore del file usando quando occorre eseguire ripetute
↪  letture e/o scritture su di esso ),
        puntatore all'indirizzo dove è presente il nome del file, un parametro
↪  per specificare il tipo di accesso al file
        e il metodo di apertura. */
    if (ff_result != FR_OK)
    {
     NRF_LOG_INFO("Unable to open or create file: " FILE_NAME ".");
        // se il file non può essere aperto o creato, tale evento viene
         ↪  notificato
     return;
 }

 ff_result = f_write(&file,&data2write, sizeof(data2write), (UINT *)
  ↪  &bytes_written);
    /* la funzione "f_write" scrive un dato in un file, dunque, prenderà in
     ↪  ingresso: un puntatore ad un "file object structure" vuoto
       (ovvero un identificatore del file usando quando occorre eseguire ripetute
↪  letture e/o scritture su di esso ),
        un puntatore al dato da scrivere nel file, il numero di bytes del dato da
↪  scrivere nel file, un puntatore alla variabile
        che ritornerà il numero di bytes effettivamente scritti. */
 if (ff_result != FR_OK)
 {
```

```
        NRF_LOG_INFO("Write failed\r\n.");// se il file non può essere scritto,
         ↪  tale evento viene notificato
    }
    else
    {
        NRF_LOG_INFO("%d bytes written.", bytes_written);// altrimenti si informa
         ↪  riguardo al numero dei bytes scritti
    }

    file_dimension = f_size(&file);
    (void) f_close(&file);
    // la funzione "f_close" chiude il file precedentemente aperto.
    return;
}
```

Rispetto all'esempio iniziale, la sostanziale modifica che è stata effettuata su fatfs_write() è stata quella di sostituire la stringa inizialmente scritta sulla SD, ovvero "SD card example." con la struttura "data2write" contenente i risultati delle conversioni e l'informazione temporale associata a ciascuno di essi. Inoltre, come si può notare, prima della chiusura del file, viene invocata la funzione f_size, la quale consente di conoscere la dimensione del file e sarà utile all'interfaccia bluetooth.
Mentre la funzione fatfs_config() viene invocata quasi all'inizio del main, come si è visto, fatfs_write() viene chiamata solo al momento del bisogno, ovvero a ricezione completata.

2.3 Trasmissione

Per la trasmissione dei dati immagazzinati della SD sono state ideate due interfacce: una seriale (UARTE) e l'altra bluetooth (BLE 5).

2.3.1 Interfaccia seriale (UARTE)

La periferica UARTE,per funzionare, necessita delle seguenti configurazioni:

1. settaggio di un paio di pin disponibili (nel caso corrente P1-02 e P1-03) rispettivamente come output e input, per svolgere la funzionalità di TXD e RXD;
2. scelta del baud-rate, qui impostato a 9600 Hz;
3. disattivazione di altre opzioni come parity-check e flow control, non necessarie allo scopo corrente;
4. abilitazione dell'interrupt su fine ricezione;
5. impostazione del buffer di ricezione, qui definito come un array di 20 posizioni di uint8_t;
6. abilitazione vera e propria della periferica;

Quanto detto è implementato con il seguente codice:

```
/*CONFIGURAZIONE PERIFERICA UARTE*/
    //configurazione pin UARTE
    nrf_gpio_cfg_output(34); //TXD output P1-2
    nrf_gpio_cfg_input(35,NRF_GPIO_PIN_NOPULL);//RXD input P1-3
    nrf_uarte_txrx_pins_set(myuarte,34,35);
    //configurazione baud-rate UARTE
    nrf_uarte_baudrate_set(myuarte,NRF_UARTE_BAUDRATE_9600);
    //opzioni uarte
    nrf_uarte_parity_t parity = NRF_UARTE_PARITY_EXCLUDED;
    nrf_uarte_hwfc_t hwfc = NRF_UARTE_HWFC_DISABLED;
    nrf_uarte_configure(myuarte,parity,hwfc);
    //abilitazione interrupt su fine ricezione
    nrf_uarte_int_enable(myuarte, NRF_UARTE_INT_ENDRX_MASK);
    //configurazione buffer ricezione UARTE
    nrf_uarte_rx_buffer_set(myuarte,(uint8_t*)rx_buffer_uarte,20);
    //abilitazione periferica UARTE
    nrf_uarte_enable(myuarte);
```

Il meccanismo in base al quale è stato possibile effettuare delle trasmissioni dei dati via UARTE senza interrompere eccessivamente il normale flusso del programma, prevede l'utilizzo di altre due periferiche:

1. **TIMER** a 32-bit impostato con frequenza 250 kHz. Tramite il modulo capture-compare, si è fatto in modo che esso generasse un evento ogni 2 secondi una volta avviato. Grazie a delle short-cut utili, inoltre, tale evento è stato collegato a dei task di stop e di clear dello stesso timer, in modo che quest'ultimo potesse poi ripreparsi automaticamente per la fase successiva.

2. **PPI(PROGRAMMABLE PERIPHERAL INTERCONNECT)**, il quale qui metterà in comunicazione il TIMER con la UARTE.

Le configurazioni di queste ultime sono riportate di seguito:

```
//configurazione timer
    nrf_timer_mode_set(mytimer,NRF_TIMER_MODE_TIMER);
    nrf_timer_bit_width_set(mytimer,NRF_TIMER_BIT_WIDTH_32);
    nrf_timer_frequency_set(mytimer,NRF_TIMER_FREQ_250kHz);
    uint32_t ticks_2s = nrf_timer_ms_to_ticks(2000,NRF_TIMER_FREQ_250kHz);
    nrf_timer_cc_write(mytimer,NRF_TIMER_CC_CHANNEL0,ticks_2s);
    nrf_timer_shorts_enable(mytimer,NRF_TIMER_SHORT_COMPARE0_STOP_MASK);
    nrf_timer_shorts_enable(mytimer,NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK);

   //configurazioni periferica PPI
   nrf_ppi_channel_endpoint_setup(0,0x40028108,0x40009000); // arrivo byte da
    ↪  seriale --> partenza del timer
   nrf_ppi_channel_enable (0); // abilito il canale PPI 0

   nrf_ppi_channel_endpoint_setup(1,0x40009140,0x40028004); //  passati i 2 s del
    ↪  timer --> stop ricezione seriale
   nrf_ppi_channel_enable (1); // abilito il canale PPI 1
```

Come si può intuire da tale codice, il protocollo pensato per lo scopo corrente prevede l'avvio del timer non appena si riscontra l'arrivo di un byte dalla seriale (evento EVENTS_RXDRDY).
In seguito, l'utente potrà inviare anche altri caratteri. Se il tempo trascorso tra l'uno e l'altro è inferiore a 2 secondi, allora il timer non genererà l'evento COMPARE0, se non 2 secondi dopo l'invio dell'ultimo di essi.
A tal punto il timer si fermerà e si resetterà. Successivamente, mediante l'ausilio del PPI, verrà innescato il task di fine ricezione della UARTE, il quale porterà alla generazione di un secondo evento, ovvero "EVENTS_ENDRX".
Quest'ultimo determinerà l'entrata nell'interrupt service routine (ISR) della UARTE.

Il codice relativo a quest'ultima è mostrato di seguito:

```
void UARTE1_IRQHandler()
{
int k = 0;

for(k = 0; k < 20; k++)
{
if(rx_buffer_uarte[k] == '\r')//quando incontro il carattere terminatore
{
uarte_request = 1; // segnalo richiesta da UARTE valida
break;//esco dal ciclo for
}
}
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_RXDRDY);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_ENDRX);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_RXSTARTED);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_RXTO);
nrf_timer_event_clear(mytimer,NRF_TIMER_EVENT_COMPARE0);
nrf_uarte_task_trigger(myuarte,NRF_UARTE_TASK_STARTRX);
}
```

Come si può notare, in quest'ultima, oltre al reset dei flag associati agli eventi accaduti, non viene fatto altro che controllare se nei caratteri inviati dall'utente c'è o meno il carattere terminatore "\r".
In caso affermativo, viene settato il flag "uarte_request", il quale consentirà poi, nel main, di svolgere le azioni opportune.
Di seguito, vengono illustrate queste ultime:

```
if(uarte_request == 1)
{
nrf_gpio_pin_write(13,0);
ff_result = f_open(&file, FILE_NAME, FA_READ);//apro il file d'interesse
if (ff_result != FR_OK)
{
// se il file non può essere aperto o creato, tale evento viene notificato
NRF_LOG_INFO("FILE NOT OPEN");
uarte_request = 0;
continue;
}
char command;
char characteristic [20];
int option;
float buffer;
float_number value;
uint8_t amount = 0;
UINT byte_read;


if(rx_buffer_uarte[0] == 'r')
{
sscanf(rx_buffer_uarte,"%c %s\r",&command,characteristic);
//in base alla caratteristica fa qualcosa


if(strcmp(characteristic,"length") == 0) // se la caratteristica è "length"
{

file_length = f_size(&file);
int n = sprintf(tx_buffer_uarte,"file length:%d\r\n",file_length/4);
nrf_uarte_tx_buffer_set(myuarte,(uint8_t*)tx_buffer_uarte,n);
nrf_uarte_task_trigger(myuarte,NRF_UARTE_TASK_STARTTX);

while(nrf_uarte_event_check(myuarte,NRF_UARTE_EVENT_ENDTX) == 0);

nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXDRDY);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_ENDTX);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXSTARTED);
}


if(strcmp(characteristic,"values") == 0) // se la caratteristica è "values"
{
nrf_gpio_cfg_output(15);
nrf_gpio_pin_write(15,0);
f_lseek(&file,pos);//mi sposto all'ultima posizione indicata da "pos" (ultima
 ↪  modifica)
```

```
while (f_read(&file,(uint8_t*)(&buffer),sizeof(my_sd_data),&byte_read) == FR_OK
 ↪  && amount < 64)//finché la lettura di sizeof(my_sd_data) byte del file va a
 ↪  buon fine e i byte mandati sono meno di 64
{
amount ++;
value = float_split(buffer,3);
sprintf(tx_buffer_uarte,"%d) %d.%d
 ↪  V\r\n",pos++,value.integer_part,value.decimal_part);
nrf_uarte_tx_buffer_set(myuarte,(uint8_t*)tx_buffer_uarte,10);
nrf_uarte_task_trigger(myuarte,NRF_UARTE_TASK_STARTTX);

while(nrf_uarte_event_check(myuarte,NRF_UARTE_EVENT_ENDTX) == 0);

nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXDRDY);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_ENDTX);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXSTARTED);
  }
 }
}

else if (rx_buffer_uarte[0] == 'w')
{
sscanf(rx_buffer_uarte,"%c %19s %d\r",&command,characteristic,&option);
//in base alla caratteristica e all'opzione fa qualcosa

if(strcmp(characteristic,"position") == 0) // se la caratteristica è "position"
{
pos = 4*option;
if( pos  <= file_length)
{
f_lseek(&file,pos);// mi sposto alla posizione indicata da pos
}
else if (pos > file_length)
{
pos = file_length;
f_lseek(&file,pos);// mi sposto alla posizione indicata da pos
sprintf(tx_buffer_uarte,"EOF.\r\n");
nrf_uarte_tx_buffer_set(myuarte,(uint8_t*)tx_buffer_uarte,8);
nrf_uarte_task_trigger(myuarte,NRF_UARTE_TASK_STARTTX);

while(nrf_uarte_event_check(myuarte,NRF_UARTE_EVENT_ENDTX) == 0);

nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXDRDY);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_ENDTX);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXSTARTED);
}

int m = sprintf(tx_buffer_uarte,"new pos = %d\r\n",pos);
nrf_uarte_tx_buffer_set(myuarte,(uint8_t*)tx_buffer_uarte,m);
nrf_uarte_task_trigger(myuarte,NRF_UARTE_TASK_STARTTX);

while(nrf_uarte_event_check(myuarte,NRF_UARTE_EVENT_ENDTX) == 0);

nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXDRDY);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_ENDTX);
nrf_uarte_event_clear(myuarte,NRF_UARTE_EVENT_TXSTARTED);

}
```

```
}

//chiudo il file nella SD
(void) f_close(&file);

//svuoto il buffer di ricezione
int a = 0;
for(a=0;a<20;a++)
{
rx_buffer_uarte[a] = 0;
}
uarte_request = 0;
}
```

In breve, i passaggi effettuati in questa porzione del main, all'interno del loop principale, sono i seguenti:

1. apertura del file dentro la SD con i risultati delle conversioni e le informazioni temporali relative ad essi;
2. controllo del primo carattere inviato dall'utente;
3. controllo degli altri caratteri inviati dall'utente raggruppati in stringhe o numeri interi;
4. invio all'utente dei dati richiesti via seriale;
5. chiusura del file;
6. svuotamento del buffer di ricezione della UARTE.

Per realizzare l'interfaccia seriale, si è cercato di ricreare un protocollo il più possibile simile a quello utilizzato in caso di comunicazione tramite bluetooth.
Di fatto, il primo carattere che l'utente dovrà inviare corrisponderà al tipo di comando da eseguire,cioè lettura "r" o scrittura "w"; mentre la successiva stringa sarà indicativa dell'oggetto su cui verrà effettuata tale operazione. Da un altro punto di vista però, questi ultimi potrebbero essere l'analogo del tipo e del nome della caratteristica BLE che è associata al dato a cui è interessato l'utente.

Dal punto di vista pratico, nel caso di lettura, tramite un sscanf, viene isolato il primo carattere,"r",dalla successiva stringa.
Se quest'ultima coincide con "length", verrà inviata all'utente l'informazione riguardante la lunghezza del file.
Se, invece, la stringa è uguale a "values", allora verranno inviati all'utente 64 byte (o meno nel caso non vi fosse disponibilità per tutti) del file a partire dalla posizione attuale del cursore "pos".
In caso di scrittura, invece, oltre al primo carattere,"w", e alla successiva stringa, verrà prelevato anche un intero indicativo del dato che dovrà essere scritto. Se la stringa coincide con "position", allora l'intero successivo stabilirà in quale byte del file dovrà essere posizionato il cursore. Se però il numero specificato dovesse superare la lunghezza del file, allora il cursore si posizionerà alla fine del file, avvisando l'utente dell'accaduto mediante trasmissione della stringa "EOF".

2.3.2 Interfaccia Bluetooth Low Energy 5

Per quanto riguarda l'interfaccia bluetooth si è sfruttato il modello presentato dall'esempio "ble_app_lbs" presente nel SDK, il quale originariamente:

1. configurava lo stack bluetooth
2. creava un nuovo profilo con associate due caratteristiche (oltre quelle obbligatorie):
    - una di scrittura per cambiare lo stato di un led;
    - una di notifica per essere informati sul cambio di stato di un bottone.

In particolare, per profilo(o servizio) s'intende una sorta di "contenitore" di caratteristiche, le quali sono indici di particolari funzionalità del dispositivo.
Sia il profilo che le caratteristiche sono identificate da due valori, l'UUID e l'handle.
Ad ogni caratteristica, inoltre, è associata una funzione di callback, la quale viene invocata quando l'utente effettua una richiesta di lettura o scrittura della caratteristica stessa.

Sulla prima parte (punto 1) non è stata apportata nessuna modifica, in quanto generica per tutti gli esempi e, quindi, valida anche per il caso corrente.
Al contrario, sulla seconda parte (punto 2), sono state apportate modifiche tali da adattare l'esempio in allo scopo desiderato.
In particolar modo, la caratteristica del led è stata eliminata e ne sono state aggiunte altre 3:

- una di lettura per conoscere la lunghezza del file contenuto nella sd;
- una di scrittura per posizionare il cursore in uno dei byte del file sulla sd;
- una di lettura per prelevare 32 byte del file nella sd a partire dall'ultima posizione specificata.

La caratteristica associata al bottone è stata mantenuta, nel caso fosse utile per possibili sviluppi futuri.
Ad esempio, se l'utente preme il bottone perché necessita aiuto, questo viene notificato.

Dal punto di vista pratico, come prima cosa, si è modificato il file "myservice.h" (alias "ble_lbs.h").
Qui si sono definite, al posto di quelle esistenti, due nuove strutture:

1. "ble_myservice_t" contenente gli handle delle caratteristiche e dei riferimenti alle rispettive funzioni di callback;
2. "ble_myservice_init_t" contenente solo i riferimenti alle funzioni di callback.

Queste ultime sono riportate di seguito insieme a dei necessari typedef per poter considerare delle funzioni come campi di una struttura:

```
typedef struct ble_myservice_s ble_myservice_t;

typedef void (*ble_myservice_handler_t) (uint16_t conn_handle, ble_myservice_t *
 ↪  myservice, uint8_t new_state);
typedef void (*ble_myservice2_handler_t) (uint16_t conn_handle, ble_myservice_t *
 ↪  myservice);
typedef void (*ble_myservice3_handler_t) (uint16_t conn_handle, ble_myservice_t *
 ↪  myservice);

typedef struct
{
ble_myservice_handler_t myservice_handler;
ble_myservice2_handler_t myservice2_handler;
ble_myservice3_handler_t myservice3_handler;
} ble_myservice_init_t;


struct ble_myservice_s
{
uint16_t                    service_handle;
ble_gatts_char_handles_t    char1_handles; // handle vecchia caratteristica di
 ↪  notifica del bottone (personalizzabile in possibili sviluppi futuri)
ble_gatts_char_handles_t    char2_handles; // handle caratteristica di scrittura
 ↪  per impostare cursore nel file
ble_gatts_char_handles_t    char3_handles; // handle caratteristica di lettura
 ↪  per conoscere la lunghezza del file
ble_gatts_char_handles_t    char4_handles; // handle caratteristica di lettura
 ↪  per prelevare dati dal file
uint8_t                     uuid_type;
ble_myservice_handler_t myservice_handler;
ble_myservice2_handler_t myservice2_handler;
ble_myservice3_handler_t myservice3_handler;

};
```

Nel main.c poi, attraverso la macro, definita in "myservice.h", chiamata BLE_MYSERVICE_DEF (myservice), si dichiarerà una nuova struttura di tipo "ble_myservice_t" denominata "myservice", ovvero quella che verrà poi effettivamente utilizzata per identificare il profilo creato. Inoltre, sempre in "myservice.h" sono stati definiti gli UUID del profilo e delle caratteristiche come segue:

```
#define MYSERVICE_UUID_BASE        {0x23, 0xD1, 0xBC, 0xEA, 0x5F, 0x78, 0x23,
 ↪  0x15, \
0xDE, 0xEF, 0x12, 0x12, 0x00, 0x00, 0x00, 0x00}
#define MYSERVICE_UUID_SERVICE     0x1523
#define MYSERVICE_UUID_CHAR1 0x1524
#define MYSERVICE_UUID_CHAR2 0x1525
#define MYSERVICE_UUID_CHAR3 0X1526
#define MYSERVICE_UUID_CHAR4 0X1527
```

A questo punto si è modificata la funzione "ble_myservice_init", contenuta nel file "myservice.c"(alias ble_lbs.c), la quale crea effettivamente il nuovo profilo e le nuove caratteristiche, come mostrato di seguito:

```
uint32_t ble_myservice_init(ble_myservice_t * myservice, const
 ↪  ble_myservice_init_t * myservice_init)
{
uint32_t              err_code;
ble_uuid_t            ble_uuid;
ble_add_char_params_t add_char_params;

// Initialize service structure.
myservice->myservice_handler = myservice_init->myservice_handler;
myservice->myservice2_handler = myservice_init->myservice2_handler;
myservice->myservice3_handler = myservice_init->myservice3_handler;

// Add service.
ble_uuid128_t base_uuid = {MYSERVICE_UUID_BASE};
err_code = sd_ble_uuid_vs_add(&base_uuid, &myservice->uuid_type);
VERIFY_SUCCESS(err_code);

ble_uuid.type = myservice->uuid_type;
ble_uuid.uuid = MYSERVICE_UUID_SERVICE;

err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &ble_uuid,
 ↪  &myservice->service_handle);
VERIFY_SUCCESS(err_code);

// Add characteristic 1
memset(&add_char_params, 0, sizeof(add_char_params));
add_char_params.uuid              = MYSERVICE_UUID_CHAR1;
add_char_params.uuid_type         = myservice->uuid_type;
add_char_params.init_len          = sizeof(uint8_t);
add_char_params.max_len           = sizeof(uint8_t);
add_char_params.char_props.read   = 1;
add_char_params.char_props.notify = 1;

add_char_params.read_access       = SEC_OPEN;
add_char_params.cccd_write_access = SEC_OPEN;

err_code = characteristic_add
 ↪  (myservice->service_handle,&add_char_params,&myservice->char1_handles);

if (err_code != NRF_SUCCESS)
{
return err_code;
}

// Add characteristic 2 --> caratteristica di scrittura per impostare cursore nel
 ↪  file
memset(&add_char_params, 0, sizeof(add_char_params));
add_char_params.uuid              = MYSERVICE_UUID_CHAR2;
add_char_params.uuid_type         = myservice->uuid_type;
add_char_params.init_len          = sizeof(uint32_t);
add_char_params.max_len           = sizeof(uint32_t);
add_char_params.char_props.read  = 1;
add_char_params.char_props.write = 1;

//add_char_params.read_access  = SEC_OPEN;
add_char_params.write_access = SEC_OPEN;

err_code = characteristic_add(myservice->service_handle, &add_char_params,
 ↪  &myservice->char2_handles);
```

```
if (err_code != NRF_SUCCESS)
{
return err_code;
}

// Add characteristic 3 --> caratteristica di lettura per conoscere la lunghezza
 ↪  del file
memset(&add_char_params, 0, sizeof(add_char_params));
add_char_params.uuid              = MYSERVICE_UUID_CHAR3;
add_char_params.uuid_type         = myservice->uuid_type;
add_char_params.init_len          = sizeof(uint32_t);
add_char_params.max_len           = sizeof(uint32_t);
add_char_params.char_props.read   = 1;

add_char_params.read_access  = SEC_OPEN;
add_char_params.write_access = SEC_OPEN;

add_char_params.is_defered_read = 1;

err_code = characteristic_add(myservice->service_handle, &add_char_params,
 ↪  &myservice->char3_handles);

if(err_code != NRF_SUCCESS)
{
return err_code;
}

// Add characteristic 4 --> caratteristica di lettura per conoscere un pacchetto
 ↪  dati scritto nel file
memset(&add_char_params, 0, sizeof(add_char_params));
add_char_params.uuid              = MYSERVICE_UUID_CHAR4;
add_char_params.uuid_type         = myservice->uuid_type;
add_char_params.init_len          = sizeof(uint8_t)*32;
add_char_params.max_len           = sizeof(uint8_t)*32;
add_char_params.char_props.read   = 1;

add_char_params.read_access  = SEC_OPEN;
add_char_params.write_access = SEC_OPEN;

add_char_params.is_defered_read = 1;

return characteristic_add(myservice->service_handle, &add_char_params,
 ↪  &myservice->char4_handles);

}
```

Tra i passaggi di fondamentale importanza troviamo la chiamata alla funzione "sd_ble_gatts_service_add()", la quale è necessaria per aggiungere un nuovo profilo (o servizio).
In questo caso, come si può notare dal primo parametro passato alla funzione, il servizio sarà di tipo primario, in quanto sarà l'unico presente, quindi sicuramente svolgerà un ruolo principale.
L'UUID e l'handle, sono specificati come secondo e terzo parametro.

In seguito, troviamo l'aggiunta delle caratteristiche, la quale avviene volta per volta mediante la funzione "characteristic add()".

Quest'ultima richiede come parametri l'handle associato al servizio, una struttura denominata "add_char_params" e l'handle della caratteristica stessa.
In particolare, la struttura conterrà le informazioni principali associate alla caratteristica tra cui l'UUID,la lunghezza in byte, la tipologia (lettura, scrittura o notifica).

Nel normale flusso del programma, la funzione "ble_service_init()", sopra descritta, sarà chiamata a sua volta da un'altra funzione, ovvero "services_init", contenuta nel file main.c e riportata di seguito:

```
static void services_init(void)
{
ret_code_t         err_code;
ble_myservice_init_t     init     = {0};
nrf_ble_qwr_init_t qwr_init = {0};

// Initialize Queued Write Module.
qwr_init.error_handler = nrf_qwr_error_handler;

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

// Initialize LBS.
init.myservice_handler = myservice_handler;
init.myservice2_handler = myservice2_handler;
init.myservice3_handler = myservice3_handler;

err_code = ble_myservice_init(&myservice, &init);
APP_ERROR_CHECK(err_code);
}
```

Fatto ciò si è potuti passare alla fase successiva riguardante la modifica delle funzioni di callback e i relativi meccanismi di chiamata.
Come punto di partenza per la comprensione di questi ultimi, è possibile far riferimento alla funzione "ble_myservice_on_ble_evt", contenuta nel file "myservice.c" e mostrata già arricchita delle modifiche utili:

```
void ble_myservice_on_ble_evt(ble_evt_t const * p_ble_evt, void * p_context)
{
ble_myservice_t * myservice = (ble_myservice_t *)p_context;
switch (p_ble_evt->header.evt_id)
{
case BLE_GATTS_EVT_WRITE:
on_write(myservice, p_ble_evt);
break;
case BLE_GATTS_EVT_RW_AUTHORIZE_REQUEST:
on_read(myservice, p_ble_evt);
break;
default:// No implementation needed.
break;
}
}
```

Quest'ultima potrebbe essere definita come una sorta di gestore di una parte degli stati in cui può trovarsi il dispositivo bluetooth.
In particolare:

- BLE_GATTS_EVT_WRITE è lo stato in cui ci si trova quando l'utente inoltra una richiesta di scrittura;
- BLE_GATTS_EVT_RW_AUTHORIZE_REQUEST è lo stato in cui ci si trova quando l'utente inoltra una richiesta di lettura o scrittura che deve essere prima "autorizzata".

Se, per esempio, dunque, l'utente vorrà effettuare una scrittura della caratteristica che posiziona il cursore del file nella sd su uno specifico byte, allora il gestore rileverà lo stato BLE_GATTS_EVT_WRITE, il quale porterà alla chiamata della funzione "on_write()" qui riportata:

```
tatic void on_write(ble_myservice_t * myservice, ble_evt_t const * p_ble_evt)
{
ble_gatts_evt_write_t const * p_evt_write =
 ↪  &p_ble_evt->evt.gatts_evt.params.write;

if (   (p_evt_write->handle == myservice->char2_handles.value_handle) &&
 ↪  (p_evt_write->len == 1) && (myservice->myservice_handler != NULL))
{
myservice->myservice_handler(p_ble_evt->evt.gap_evt.conn_handle, myservice,
 ↪  p_evt_write->data[0]);
}
}
```

Dapprima, nella precedente, viene verificato che l'handle coincida con "char2_handles"(ovvero quello della caratteristica suddetta), che il valore che vuole scrivere l'utente sia lungo 1 byte e che la funzione di callback esista. Successivamente, se queste tre condizioni sono verificate, allora viene effettivamente chiamata la funzione di callback "myservice_handler" riportata di seguito:

```
static void myservice_handler(uint16_t conn_handle, ble_myservice_t * myservice,
 ↪  uint32_t state)
{

  f_seek_cursor = state;
  f_seek_flag = 1;
}
```

Tutto ciò che viene fatto qui è scrivere il valore che l'utente vorrebbe attribuire alla caratteristica, ovvero il parametro "state", su una variabile globale denominata f_seek_cursor e poi settare il f_seek_flag.
Una volta che il programma tornerà nel loop principale del main, infatti, questo flag porterà all'esecuzione delle righe di codice contenute nel seguente if:

```
if(f_seek_flag == 1)
{
NRF_LOG_INFO("f_seek_flag");
ff_result = f_open(&file, FILE_NAME, FA_READ);//apro il file d'interesse
if (ff_result != FR_OK)
{
// se il file non può essere aperto o creato, tale evento viene notificato
NRF_LOG_INFO("FILE NOT OPEN");
}
// posiziono il cursore sul byte scelto dall'utente tramite l'interfaccia ble
```

```
f_lseek(&file,f_seek_cursor);

//leggo 32 byte dopo la posizione selezionata e li salvo in una variabile globale
read_once = 0;
read_total = 0;
i = 0;
k = 0;


while (f_read(&file,buffer_once,sizeof(my_sd_data),&read_once) == FR_OK &&
 ↪  read_total < 4)//finché la lettura di sizeof(my_sd_data) byte del file va a
 ↪  buon fine e i byte mandati sono meno di 64
{
for(i=0;i<8;i++)
{
data_after_cursor[k] = buffer_once[i];
k++;
}
read_total ++;

}


f_seek_flag = 0;
(void) f_close(&file);

}
```

Qui, in maniera simile a quanto visto per l'interfaccia seriale, verrà posizionato il cursore nella posizione indicata dall'utente.
Successivamente però, verranno anche letti i 32 byte del file a partire da quella posizione e salvati in una variabile globale.
Ciò perché, in questo modo, alla prossima richiesta di lettura dei valori nella sd, si avranno già i dati pronti senza bisogno di dover rallentare (o addirittura bloccare) il programma in fase di esecuzione della funzione di callback.
Fatto ciò, chiaramente, la variabile f_seek_flag dovrà essere resettata in vista delle prossime richieste.

Per quanto riguarda le altre due caratteristiche, analogamente, nel momento in cui l'utente inoltra una richiesta di lettura, si entrerà nello stato BLE_GATTS_EVT_RW_AUTHORIZE_REQUEST. Ciò per via del valore 1 attribuito al campo "is_defered_read" della relativa struttura "add_char_params", in fase di aggiunta delle caratteristiche in questione. In entrambi i casi, verrà chiamata la funzione "on_read()", la quale è riportata di seguito:

```
static void on_read(ble_myservice_t * myservice, ble_evt_t const * p_ble_evt)
{
NRF_LOG_INFO("on_read");
ble_gatts_evt_rw_authorize_request_t const * p_evt_rw_authorize_request =
 ↪  &p_ble_evt->evt.gatts_evt.params.authorize_request;

if (   (p_evt_rw_authorize_request->request.read.handle ==
 ↪  myservice->char3_handles.value_handle)
&& (myservice->myservice2_handler != NULL))
```

```
{
myservice->myservice2_handler(p_ble_evt->evt.gap_evt.conn_handle, myservice);
}

if (   (p_evt_rw_authorize_request->request.read.handle ==
 ↪  myservice->char4_handles.value_handle)
&& (myservice->myservice3_handler != NULL))
{
myservice->myservice3_handler(p_ble_evt->evt.gap_evt.conn_handle, myservice);
}


}
```

Qui, analogamente a quanto visto per la funzione "on_write", si verificherà l'handle della caratteristica e l'esistenza (o validità) della funzione di callback. In particolare, se la richiesta inoltrata riguarda la caratteristica associata alla lettura dei dati dalla sd, verrà chiamata la funzione "myservice3_handler", la quale è riportata di seguito:

```
static void myservice3_handler(uint16_t conn_handle, ble_myservice_t * myservice)
{

ble_gatts_rw_authorize_reply_params_t add_params;
memset(&add_params,0,sizeof(add_params));
add_params.type = 1;
add_params.params.read.update = 1;
add_params.params.read.offset = 0;
add_params.params.read.len = sizeof(uint8_t)*32;
add_params.params.read.p_data = data_after_cursor;

sd_ble_gatts_rw_authorize_reply(conn_handle,&add_params);

}
```

Come prima cosa, qui viene creata una nuova struttura "add_params" nella quale verranno specificate informazioni sulla caratteristica quali il suo valore, il suo tipo(lettura o scrittura)e la sua lunghezza.
Successivamente tale struttura verrà passata come parametro alla funzione "sd_ble_gatts_rw_authorize_reply()", la quale andrà effettivamente ad aggiornare il valore della caratteristica, in questo caso con i dati contenuti in "data_after_cursor".
Qui dentro vi saranno, come accennato prima, i 32 byte prelevati nel loop principale del main, in seguito al settaggio di un flag nella callback "myservice_handler".
Ciò però non significa che, prima di effettuare una lettura dei dati nella sd occorre sempre posizionare prima il cursore.
Infatti, tra le prime azioni svolte nel main, in seguito alla configurazione dello stack bluetooth, vi è anche una prima apertura del file per prelevare l'informazione sulla lunghezza e sui primi 32 byte a partire dalla posizione 0. È chiaro che se, invece, si vogliono 32 byte localizzati in un'altra posizione, occorrerà prima posizionare il cursore.

Infine, se l'utente inoltra una richiesta di lettura riguardamte la caratteristica associata alla lunghezza del file, allora la "on_read()" chiamerà la funzione di callback denominata "myservice2_handler()", la quale è riportata di seguito:

```
static void myservice2_handler(uint16_t conn_handle, ble_myservice_t * myservice)
{
NRF_LOG_INFO("myservice2_handler");

ble_gatts_rw_authorize_reply_params_t add_params;
uint32_t my_data = file_dimension;
memset(&add_params,0,sizeof(add_params));
add_params.type = 1;
add_params.params.read.update = 1;
add_params.params.read.offset = 0;
add_params.params.read.len = sizeof(uint32_t);
add_params.params.read.p_data = (uint32_t*)&my_data;

sd_ble_gatts_rw_authorize_reply(conn_handle,&add_params);

}
```

Come si può notare, quest'ultima è strutturata molto simile alla precedente.
Infatti le uniche differenze sono il valore della caratteristica, qui rappresentato dalla variabile "my_data", la quale contiene il valore della variabile globale "file_dimension", e la sua lunghezza, qui uint32_t.
In particolare, la variabile "file_dimension" viene scritta agli inizi del programma, come accennato in precedenza, per poi essere aggiornata ad ogni chiamata della funzione "fatfs_write", ovvero in seguito ad ogni scrittura sulla sd.

3 Porting sulla nuova board

3.1 Introduzione

L'ultima fase del progetto è consistita in un tentativo di porting del lavoro fatto su una nuova board realizzata proprio nel dipartimento d'ingegneria dell'informazione dell'UNIVPM.
Tale board è caratterizzata dalla presenza di un modulo BT480 [8] descrivibile mediante il seguente schema:

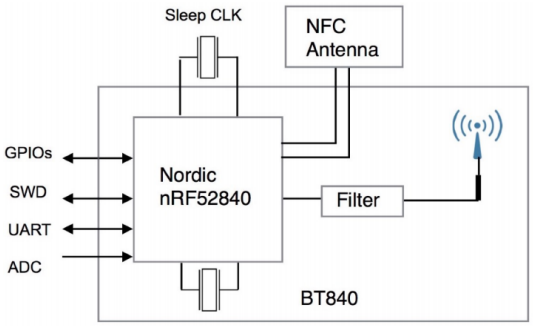

Figura 2: modulo bt480

ovvero montante lo stesso microcontrollore Nordic nRF52840 usato in precedenza ma avente in più l'antenna bluetooth e la rete di adattamento per la stessa.

Nella board in questione troviamo collegate ai pin del micro diverse periferiche e componenti, come mostrato nel seguente schema elettrico:

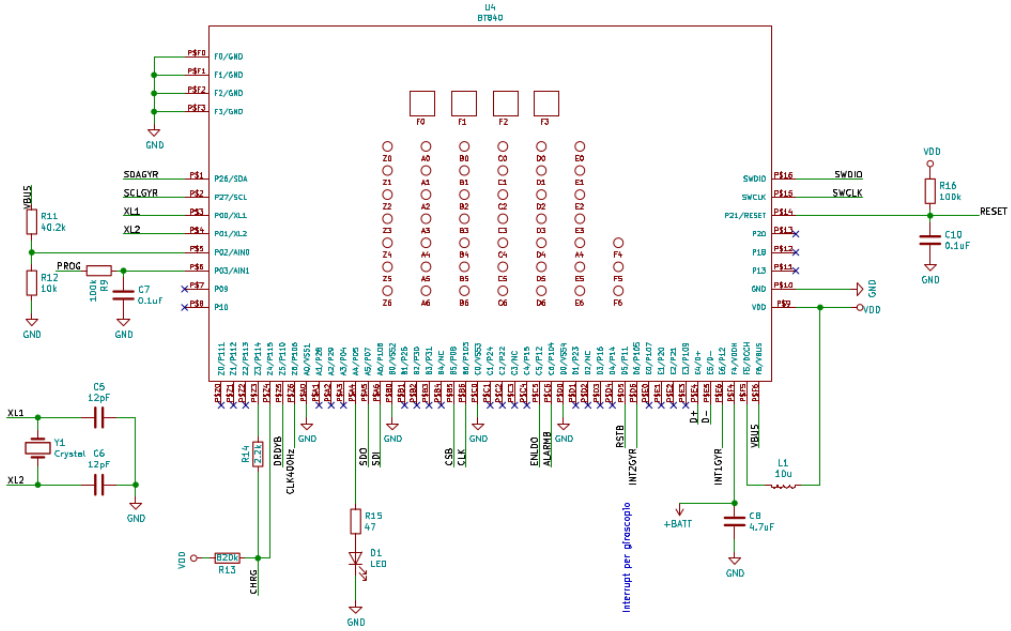

Figura 3: schema elettrico board

3.2 Breve descrizione dell'ADS1293

Tra le periferiche suddette, troviamo l'ADS1293 [9] della Texas Instrument, un convertitore analogico-digitale con 24 bit di risoluzione, adatto per l'acquisizione di biopotenziali e dotato di interfaccia SPI.
Lo schema elettrico di principio di tale componente è mostrato di seguito:

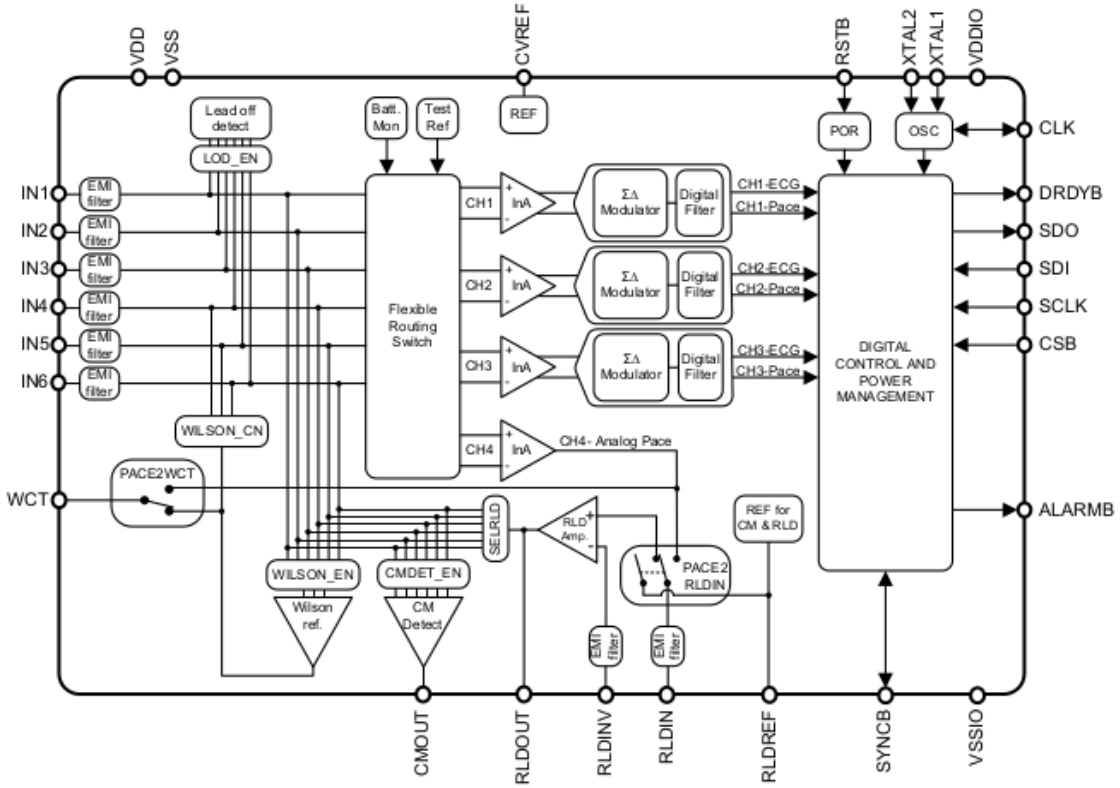

Figura 4: schema elettrico ads

Come si può notare dal grafico precedente, tale ads presenta 3 ingressi differenziali, ovvero al suo ingresso possono essere applicati massimo 6 segnali analogici. Questi ultimi poi, tramite l'interfaccia denominata "flexible routing switch" vengono indirizzati in 4 amplificatori differenziali per strumentazione (Ina) con guadagno fisso x3.5.
Uno di questi è dedicato all'analog pace che necessita di alte velocità ma non di estrema precisione, viene utilizzato per monitorare il ritmo del muscolo cardiaco.
Gli alti 3 InA, invece, vengono utilizzati per amplificare il segnale differenziale in ingresso proveniente dagli elettrodi oppure per testare delle tensioni di riferimento o quella della batteria. L'uscita degli InA, infine, viene applicata prima all' ingresso di un modulatore sigma-delta e poi di un filtro digitale configurabile tramite i registri interni.
Ogni singolo canale può essere programmato a sé, cioè ognuno può essere caratterizzato da una banda e un sample diverso da quello degli altri. In tal modo, si possono ottenere miglioramenti a livello di performance e consumi.

Di seguito, vengono mostrate le connessioni dell'ads con gli altri componenti della board:

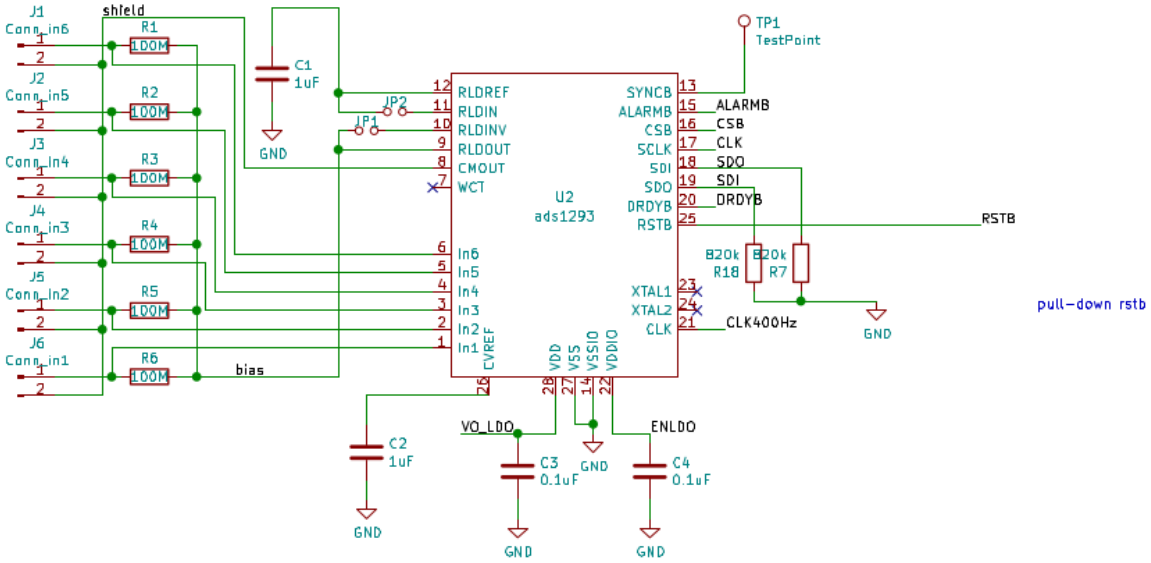

Figura 5: schema connessioni ads

3.3 Accensione dell'ads1293

L'obiettivo è quello di sfruttare l'interfaccia bluetooth di cui sopra per trasferire i dati acquisiti tramite i 3 canali differenziali dell'ADS1293 ad un utente generico.
Prima di tutto, dunque, è necessario rendere operativo il convertitore.
A tale scopo, occorre, in condizioni di reset (pin RSTB basso):

1. portare alto il pin ENLDO;
2. fornirgli un clock a 400 KHz (con una tolleranza sulla frequenza del 20% circa);
3. attendere 10-20 ms l'accensione;

In particolare, a differenza della parte digitale dell' ADS1293 che è alimentata tramite il pin VDDIO, la parte analogica deve essere alimentata ad una tensione costante di 3.3 V circa.
Per ottenere tale tensione si sfrutta un dispositivo chiamato TPS78833 [10] che è un "low-dropout regulator"(LDO), ovvero un regolatore di tensione lineare capace di funzionare correttamente anche con una piccola differenza tra tensione d'ingresso e d'uscita.Il suo schema elettrico è il seguente:

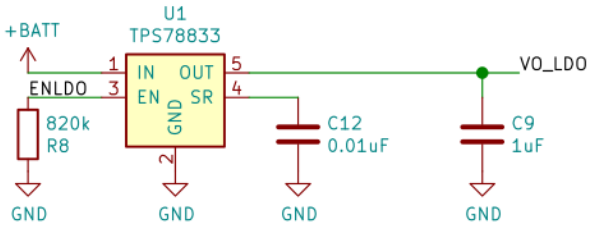

Figura 6: schema elettrico ldo

All'LDO è applicata in ingresso una tensione variabile dai 3.4 V ai 4.2 V (a seconda dello stato di carica della batteria agli ioni di litio che alimenta il micro); in uscita, invece, è fornita una tensione circa costante e pari a 3.3 V (il condensatore C9 serve in qualità di filtro passa basso).
Portando alto il pin ENLDO si rende possibile all'ADS1293 l'utilizzo di quest'ultima; dunque dopo un tempo opportuno, quando il condensatore C9 sarà carico, il dispositivo sarà operativo.

Purtroppo, già in questa prima fase del lavoro, si è dovuta affrontare una problematica legata ad un difetto di progetto della board sopra descritta. Infatti, sebbene il segnale di enable dell'LDO dovrebbe essere attivo basso, per accendere l'ads, è necessario che questo vada alto. Per risolvere tale questione, è stato modificato l'hardware tagliando una pista e inserendo un cavo tra una delle gpio libere e l'enable dell'ldo. Dal punto di vista firmware poi, chiaramente, tale gpio è stata sempre scritta con un valore complementato rispetto a quello del vecchio enable,ancora presente.

Per creare l'attesa necessaria alla carica del condendatore sopra citato è stata utilizzata la periferica TIMER2 del micro a frequenza 250 KHz e configurata in modo tale da generare un evento ogni millisecondo, per poi farla ciclare 20 volte.
Per quanto riguarda il clock, infine, si è sfruttata la periferica PWM0 configurata in modo tale da avere un'onda quadra a frequenza 400 KHz (frequenza base di 4 MHz divisa per un fattore 10) con duty cycle del 50%. La porzione di codice che descrive quanto sopra è:

```
 //configurazione PWM a 400 kHz per ads
nrf_gpio_cfg_output(38);
//configurazione PWM a 400 kHz per ads
nrf_pwm_enable(mypwm);
uint32_t pwm_pins [4] = {38,NRF_PWM_PIN_NOT_CONNECTED,NRF_PWM_PIN_NOT_
↪  CONNECTED,NRF_PWM_PIN_NOT_CONNECTED};
nrf_pwm_pins_set(mypwm,pwm_pins);
nrf_pwm_configure(mypwm,NRF_PWM_CLK_4MHz,NRF_PWM_MODE_UP,10);
nrf_pwm_seq_cnt_set(mypwm,0,1);
static uint16_t pwm_sequence[1] = {5};
nrf_pwm_sequence_t const sequence =
{
```

```
.values.p_common = pwm_sequence,
.length          = sizeof(pwm_sequence)/sizeof(uint16_t),
.repeats         = 0,
.end_delay       = 0
};
nrf_pwm_sequence_set(mypwm,0,&sequence);

//configurazione ads e timer per attesa alimentazione LDO
//configurazione timer per attesa di 1 ms circa ( poi da ripetere in loop 20
 ↪  volte )
nrf_timer_mode_set(mytimer2,NRF_TIMER_MODE_TIMER);
nrf_timer_bit_width_set(mytimer2,NRF_TIMER_BIT_WIDTH_32);
nrf_timer_frequency_set(mytimer2,NRF_TIMER_FREQ_250kHz);
uint32_t ticks_1ms = nrf_timer_ms_to_ticks(1,NRF_TIMER_FREQ_250kHz);
nrf_timer_cc_write(mytimer2,NRF_TIMER_CC_CHANNEL1,ticks_1ms);
nrf_timer_shorts_enable(mytimer2,NRF_TIMER_SHORT_COMPARE1_STOP_MASK);
nrf_timer_shorts_enable(mytimer2,NRF_TIMER_SHORT_COMPARE1_CLEAR_MASK);

//CONFIGURAZIONE ADS
nrf_gpio_cfg_output(33);//RSTB (reset)
nrf_gpio_cfg_output(34);//ENLDO n.1 (alimentazione)
nrf_gpio_cfg_output(10);//ENLDO n.2 (alimentazione)
nrf_gpio_pin_write(33,0);//RSTB low (acceso)
nrf_gpio_pin_write(34,0);//ENLDO n.1 low
nrf_gpio_pin_write(10,1);//ENLDO n.2 high

nrf_gpio_pin_write(34,1);//ENLDO n.1 high
nrf_gpio_pin_write(10,0);//ENLDO n.2 low

nrf_pwm_task_trigger(mypwm,NRF_PWM_TASK_SEQSTART0);//faccio partire il clock 400
 ↪  kHz per l'ads

int a = 0;
while (a != 20)//innesco il timer di 20 ms
{
nrf_timer_task_trigger(mytimer2,NRF_TIMER_TASK_START);
if(nrf_timer_event_check(mytimer2,NRF_TIMER_EVENT_COMPARE1) == 1)
{
a++;
nrf_timer_event_clear(mytimer2,NRF_TIMER_EVENT_COMPARE1);
}
}
nrf_gpio_pin_write(33,1);//RSTB high (spento)
```

3.4 Configurazione dei registri dell'ads e acquisizione dei dati tramite SPI

Una volta acceso il convertitore, affinché si possano acquisire correttamente dati, è necessario configurare in modo opportuno alcuni dei suoi registri mediante comunicazione SPI. Di seguito, viene mostrata innanzitutto la porzione di codice utile per configurare la periferica SPIM3(SPI-MASTER)del micro:

```
//settaggio dei pin p1-03 come clock,p0-07 come miso e p0-08 come mosi
nrf_spim_pins_set^^I(myspim,35,7,40);

//configurazione dei pin spim come input/output
nrf_gpio_cfg_output(35); //clk -> output
nrf_gpio_cfg_output(7); //mosi ->output
```

```
nrf_gpio_cfg_output(8); //cs -> output
nrf_gpio_cfg_input(40,NRF_GPIO_PIN_NOPULL); //miso->input, no pull perché c'è
 ↪  hardware verso il basso

//configurazione modalità operativa spim
nrf_spim_configure^^I(myspim,NRF_SPIM_MODE_0, NRF_SPIM_BIT_ORDER_MSB_FIRST );

//configurazione chip select
nrf_spim_csn_configure(myspim,8,NRF_SPIM_CSN_POL_LOW,1);


//scelta della frequenza di clock
nrf_spim_frequency_set^^I(myspim,NRF_SPIM_FREQ_8M); // 8 MHz ( l'ads può arrivare
 ↪  massimo a 10 MHz )


//abilitazione degli interrupt SPIM
nrf_spim_int_enable^^I(myspim, NRF_SPIM_INT_END_MASK); //interrupt a fine
 ↪  transazione
```

Come si può notare, tutto ciò che è stato fatto è:

- configurare i pin P1-03,PO-07,P0-08 e P1-08 rispettivamente come CLK (output), MOSI (output), CSN (output)e MISO (input with no pull);
- selezionare la modalità operativa con CPOL=0 e CPHA=0 (modalità 0);
- scegliere l'endianess in modo tale da trasmettere/ricevere prima il byte più significativo (MSB first);
- settare la polarità del chip select come low (CSN attivo basso);
- determinare il tempo che deve trascorrere tra la fine di una transazione (stop del vecchio clock)e l'istante in cui il chip select può essere riportato basso prima dell'inizio della nuova, ovvero l'anticipo dell'abbassamento del chip select prima della ripartenza del clock. Tale intervallo temporale è stato impostato di 15 ms circa in quanto minimo possibile, nonostante l'ads lo richiedesse minimo di 5 ms;
- scegliere la frequenza di lavoro dell'SPI come 8 MHz (l'ads può lavorare fino a 10 MHz);
- abilitare l'interrupt sull'evento END che indica la fine di una transazione generica.

Una volta pronta la periferica SPIM3, sono stati configurati i registri dell'ADS1293 e si è dato inizio alla conversione mediante la seguente porzione di codice:

```
//abilitazione periferica SPIM
nrf_spim_enable(myspim);

//inizializzazione registri ads:
uint8_t ads_reg_addr[] = {0x00, 0x01, 0x02, 0x03, 0x05, 0x0a, 0x0c, 0x12, 0x13,
 ↪  0x14, 0x15, 0x21, 0x22, 0x23, 0x24, 0x25, 0x27, 0x2f};
uint8_t ads_reg_data[] = {0x00, 0x0a,  065, 0x00, 0x00, 0x00, 0x05, 0x06, 0x3f,
 ↪  0x04, 0x04, 0x08, 0x01, 0x01, 0x01, 0x00, 0x08, 0x71};

int i = 0;
for(i = 0; i < sizeof ads_reg_addr; i++)
```

```
{
uint8_t temp[2]={ads_reg_addr[i], ads_reg_data[i]};
transaction_spi(temp,2,(unsigned int*)0x00,0);
}

//inizio conversione
uint8_t start_data_conversion[2]={0x00, 0x01};
transaction_spi(start_data_conversion,2,(unsigned int*)0x00,0);
```

Per i dettagli riguardanti le scelte dei valori con i quali sono stati impostati i vari registri si rimanda alla tesi di Alessandro Carra.[11]
Ogni qual volta una transazione termina, la periferica SPIM genererà l'evento END che permetterà l'entrata nella seguente ISR:

```
void SPIM1_SPIS1_TWIM1_TWIS1_SPI1_TWI1_IRQHandler()
{
nrf_spim_event_clear(myspim,NRF_SPIM_EVENT_STARTED);
nrf_spim_event_clear(myspim,NRF_SPIM_EVENT_ENDTX);
nrf_spim_event_clear(myspim,NRF_SPIM_EVENT_ENDRX);
nrf_spim_event_clear(myspim,NRF_SPIM_EVENT_END);
nrf_gpio_pin_write(8,1);//rimetto il chip select alto

if ( *((unsigned int*)0x4002F538) != 0)//se la lunghezza del buffer di ricezione
 ↪  è diversa da 0, ovvero c'è stata una ricezione.
{
reception_complete=1;
}
}
```

Come si può notare, in quest'ultima saranno resettati i flag relativi agli eventi accaduti nel corso della transazione ma, siccome non si sta volutamente realizzando una ricezione, il flag reception_complete non sarà messo a 1.
Nel contesto corrente, è importante sapere che, in seguito all'ultima configurazione che consiste nel settaggio a 1 del primo bit del registro 0x00, inizierà la conversione.
A tal punto, si effettuerà l'acquisizione di 3 byte di dato da ciascuno dei 3 canali ECG (dunque non dai canali PACE), ogni qual volta andrà basso il pin DRDYB(DATA READY BAR).
Affinché ciò avvenga, quest'ultimo è stato configurato come GPIOTE ed è stato abilitato un interrupt sul suo fronte di discesa mediante il seguente codice:

```
//configurazioni periferica GPIOTE
nrf_gpio_cfg_input(42,NRF_GPIO_PIN_NOPULL);//configuro il pin DRDYB(data ready
 ↪  bar:P1_10 = 42) come input
nrf_gpiote_event_enable(0); //abilito l'evento su GPIOTE channel 0
nrf_gpiote_event_configure(0,42, NRF_GPIOTE_POLARITY_HITOLO); //genero un evento
 ↪  quando sul pin DRDYB ho un fronte di discesa
nrf_gpiote_int_enable(NRF_GPIOTE_INT_IN0_MASK);//attivo l'interrupt quando il pin
 ↪  DRDYB ha un fronte di discesa
```

In tal modo, infatti, pronti i dati da acquisire, il pin DRDYB genererà un evento NRF_GPIOTE_EVENTS_IN_0 il quale permetterà l'entrata nella seguente ISR:

```
void GPIOTE_IRQHandler()
{
uint8_t start_reception = 0x37 | 0b10000000; //indirizzo del primo registro ecg
 ↪  da leggere + bit di ricezione
transaction_spi(&start_reception,1,ads_receveid_data,10);
}
```

Da quest'ultima viene avviata la lettura di 10 byte di dato a partire dal registro del convertiore numero 0x37, dove risiede il primo byte del primo canale ECG da leggere, in poi.
Il fatto che tali byte siano 10 anziché 9 (3 byte x canali ECG) deriva dal fatto che l'SPI è un protocollo di comunicazione seriale full-duplex; di conseguenza, nel momento in cui si trasmette l'indirizzo dove leggere sul MOSI, si riceve contemporaneamente sul MISO un garbage byte, il quale poi sarà trascurato.

Una volta terminata la ricezione dei dati tramite SPIM, tale periferica genererà l'evento END il quale consentirà nuovamente l'entrata nella già citata ISR. Qui, questa volta, verrà anche settata anche la variabile reception_complete. Di conseguenza, una volta tornati nel loop infinito del main sarà possibile copiare i 9 byte utili all'interno di un buffer più grande, mediante la seguente porzione di codice:

```
while(1)
{

if (reception_complete == 1)
{
NRF_LOG_INFO("reception complete");

//copio i dati ricevuti su un buffer più grande
uint8_t pos = 9*(occupied_pos_buffer);

for(int k = pos; k < pos+9; k++)
{
ads_buffer[k] = ads_receveid_data[k-pos+1];
}

if (occupied_pos_buffer < 9)
{
occupied_pos_buffer++;
}
else
{
occupied_pos_buffer = 1;
}
reception_complete = 0;
}
```

3.5 Trasmissione dei dati dell'ads tramite BLE

In seguito alla fase di acquisizione e bufferizzazione, i dati potranno poi essere trasmessi tramite un'interfaccia BLE identica a quella utilizzata nel programma descritto nella precedente sezione,fatta eccezione per le funzioni di callback, le quali sono riportate di seguito:

```
static void myservice_handler(uint16_t conn_handle, ble_myservice_t * myservice,
 ↪  uint32_t state)
{
//salvo il valore passato dall'utente in una variabile globale
buffer_cursor = state;

}
static void myservice3_handler(uint16_t conn_handle, ble_myservice_t * myservice)
{

ble_gatts_rw_authorize_reply_params_t add_params;
memset(&add_params,0,sizeof(add_params));
add_params.type = 1;
add_params.params.read.update = 1;
add_params.params.read.offset = 0;
add_params.params.read.len = sizeof(uint8_t)*9;
add_params.params.read.p_data = ads_buffer+buffer_cursor;

sd_ble_gatts_rw_authorize_reply(conn_handle,&add_params);

}
```

Come si può notare, nella prima di esse viene semplicemente salvata la posizione del cursore sul buffer in una variabile globale.
Nella seconda, invece,vengono letti i 9 byte (singola acquisizione) contenuti nel buffer a partire dalla posizione specificata dal cursore.
La funzione di callback "myservice2_handler", in questo nuovo contesto è stata ignorata, in quanto, avendo sostituito la SD con un buffer ram di lunghezza fissa, non occorreva più una caratteristica atta ad informare l'utente sul numero di dati salvati.

3.6 Misure

Precedentemente, è stato specificato che l'interfaccia SPI utilizzata è stata SPIM3. Inizialmente, tale scelta sembrava quasi obbligata, in quanto le altre interfacce non sono dotate di gestione del CSN (chip-select).

Tuttavia, nel test finale del programma, sono state riscontrate delle anomalie nel comportamento di tale periferica, per cui si è deciso di riscrivere la parte di codice dedicata all'SPI utilizzando i driver nrfx. In tal caso, il programma è risultato funzionante. Come prova di ciò, è stata condotta una misura cortocircuitando il primo canale con il terzo e lasciando aperto il secondo. Di seguito, sono mostrati gli andamenti nel tempo dei segnali acquisiti sui 3 canali:

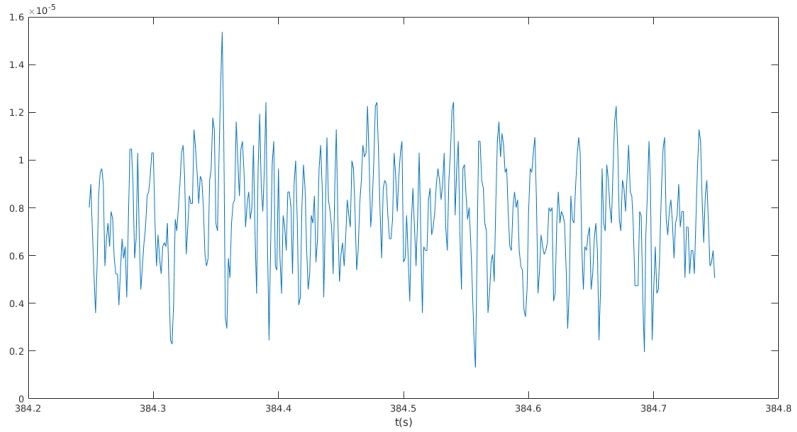

Figura 7: andamento temporale del segnale sul canale 1

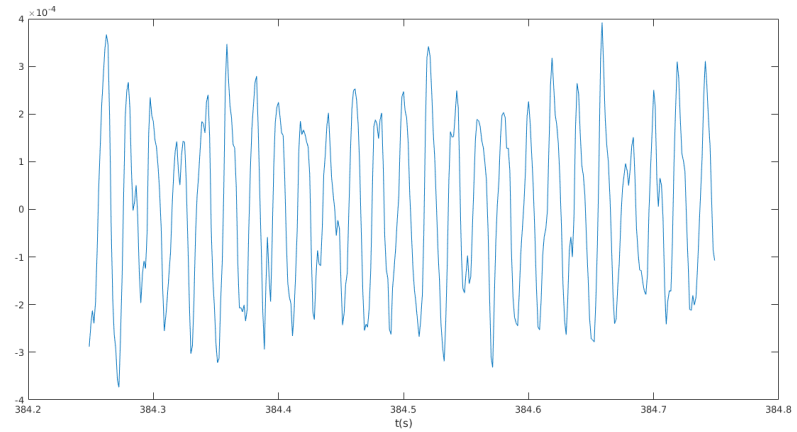

Figura 8: andamento temporale del segnale sul canale 2

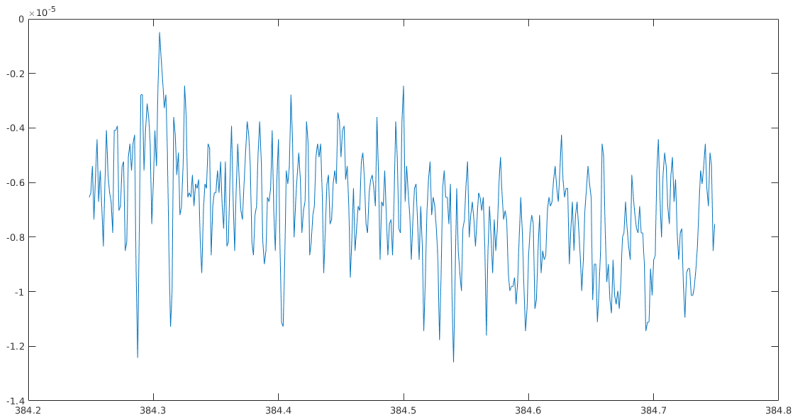

Figura 9: andamento temporale del segnale sul canale 3

Come si può notare, mentre sui canali 1 e 3 c'è solo rumore dovuto all'amplificatore/convertitore; sul canale 2 si riesce ad intravedere un andamento sinusoidale a 50 Hz (frequenza di rete).
Per visualizzare meglio ciò, di seguito è stata riportata la DFT del segnale sul canale 2:

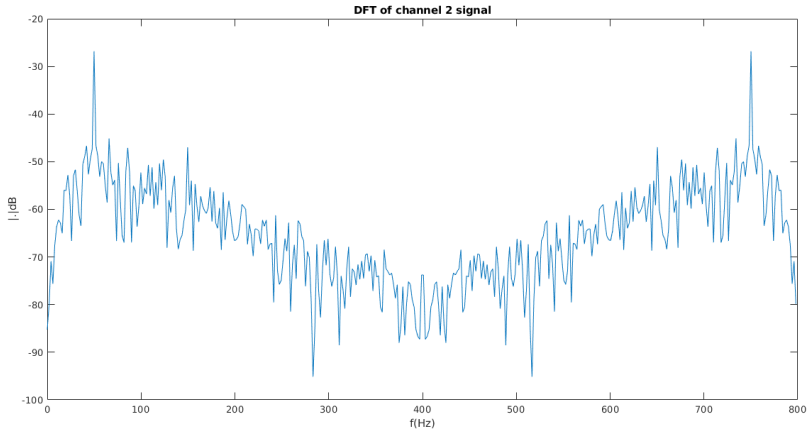

Figura 10: DFT del segnale sul canale 2

In quest'ultima, infatti, tra la grande varietà di componenti armoniche presenti, si può osservare una componente preponderante in corrispondenza dei 50 Hz e per periodicità, in corrispondenza dei 750 Hz (prima replica).

4 Conclusioni

In concordanza con quanto specificato nella sezione 1.1, l'obiettivo della tesi si può considerare raggiunto.

In particolare, volendo considerare la prima fase, è possibile affermare che nella parte di acquisizione, si è riusciti sia nell'intento di far funzionare correttamente l' interfaccia IIC del sensore INA226, sia in quello di effettuare una bufferizzazione intelligente dei dati ricevuti. Le varie casistiche di trasmissione e ricezione, infatti, sono state gestite mediante uno schema di interrupt efficiente, permettendo al dispositivo di restare in uno stato di risparmio energetico, quando possibile.
I valori di tensione prelevati, inoltre, sono stati salvati con un buon grado di precisione, affiancati da un riferimento temporale utile ad una migliore comprensione dello stato corrente.

Nella parte di trasmissione, analogamente, si possono considerare riuscite in modo ottimo sia l'interfaccia seriale tramite UART, sia quella bluetooth.
La prima è stata creata, come prova, pensando ad un algoritmo tale da poter simulare l'ambiente bluetooth con le sue richieste di lettura e scrittura delle caratteristiche. Di conseguenza, una volta sviluppata correttamente la sezione di codice sulla UART, il passaggio dall'una all'altra modalità di comunicazione non ha comportato grossi cambiamenti dal punto di vista del codice, eccetto alcuni formalismi riguardanti il funzionamento base del BLE.

Per essere più chiari, le azioni da svolgere in seguito ad una richiesta dell'utente da seriale, tramite l'invio di uno o più caratteri, sono state riadattate senza troppe variazioni come funzioni di callback associate alle caratteristiche del profilo BLE creato, raggiungendo in entrambi i casi il successo.

Un discorso simile si potrebbe fare sulla seconda parte, in quanto questa è consistita nel semplice porting del lavoro svolto su una scheda diversa da una demoboard.
L' unica variazione nell'acquisizione, rispetto alla situazione precedente, è stata la sostituzione del sensore INA226 con l'ADS1293 come dispositivo dal quale acquisire i dati. Ciò ha comportato la sostituzione dell'interfaccia IIC gestita manualmente tramite interrupt con una SPI gestita sempre ad interrupt ma tramite driver, per la configurazione dei registri e la gestione dei meccanismi di rice-trasmissione.
Dal punto di vista della bufferizzazione, invece, non avendo la nuova scheda uno slot per la micro SD, si è effettuato, provvisoriamente, un salvataggio dei dati in RAM.
Per quanto riguarda, infine, la trasmissione, si è deciso di conservare solo la comunicazione via BLE e non ci sono state variazioni rilevanti rispetto al caso precedente.

Riferimenti bibliografici

[1] *nRF52840 Product Specification*, v. 1.0, Nordic Semiconductor, mar. 2018.

[2] Nordic Semiconductor. (2019). NRF5 SDK documentation, indirizzo: https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v15.3.0/index.html (visitato il 11/11/2019).

[3] ——, (2019). NRF5 SOFTDEVICE documentation, indirizzo: https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.s140.api.v7.0.1/index.html (visitato il 11/11/2019).

[4] A. Vieira. (2019). GNU Arm Embedded Toolchain PPA, indirizzo: https://launchpad.net/~team-gcc-arm-embedded/+archive/ubuntu/ppa (visitato il 11/11/2019).

[5] Segger. (2019). Segger JLink, indirizzo: https://www.segger.com/downloads/jlink (visitato il 11/11/2019).

[6] *High-Side or Low-Side Measurement, Bi-Directional Current and Power Monitor*, INA226, Rev. A, Texas Instruments, giu. 2011.

[7] ChaN. (2019). FATFS module, indirizzo: http://elm-chan.org/fsw/ff/00index_e.html (visitato il 11/11/2019).

[8] *BLE 5, Thread, Zigbee Modules*, BT480, Rev. 1.10, FANSTEL, ago. 2018.

[9] *Low-Power, 3-Channel, 24-Bit Analog Front-End for Biopotential Measurements*, ADS1293, Rev. C, Texas Instruments, feb. 2013.

[10] *150-mA LOW-NOISE LDO WITH IN-RUSH CURRENT CONTROL FOR USB APPLICATION*, TPS78833, Rev. 07/2001, Texas Instruments, giu. 2001.

[11] A. Carra, "Studio e realizzazione di sistemi per l'acquisizione di biopotenziali elettrici", Università Politecnica delle Marche, 2019.