
**UNIVERSITA' POLITECNICA DELLE MARCHE**

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Elettronica

# Implementazione su sistemi indossabili di algoritmi per il riconoscimento di attività motorie

# Implementation of algorithms for the recognition of human activities on wearable system

Relatore:

Prof. Biagetti Giorgio

Tesi di Laurea di:

Bocchini Dario

A.A. 2021 /2022

INDICE

INDICE DELLE FIGURE

# 1. Introduzione

Nell'utilizzo di dispositivi indossabili frequente è la richiesta di funzioni che permettano il riconoscimento di attività motorie: a tal fine si è sviluppato un algoritmo in grado di fornire dati utili all'identificazione; detto algoritmo è stato predisposto per sistemi con potenza di calcolo limitata come la scheda custom utilizzata basata su microcontrollore *nrf52840* della Nordic Semiconductor. La scheda in questione è osservabile in Figura 1. Il microcontrollore è basato su una CPU a 32 bit ARM Cortex M4 con un'unità di calcolo per operazioni a virgola mobile a 64 MHz. Inoltre, dispone: di una comunicazione bluetooth low energy, diverse periferiche e interfacce digitali. Infine, possiede 1 MiB di memoria flash e 256 KiB di RAM.

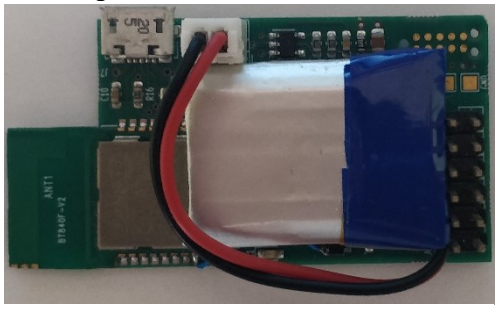

*Figura 1: Scheda custom utilizzata.*

I dati campionati necessari all'elaborazione dell'algoritmo verranno ottenuti tramite il sensore presente sulla scheda; quest'ultimo è il *lsm6dso* prodotto dalla STMicroelectronics. Nel sensore in questione sono presenti un accelerometro e un giroscopio; esso dispone di vari tipi di comunicazione, ma per la scheda utilizzata risulta essere connesso attraverso una trasmissione I2C. Dispone inoltre di diverse frequenze di campionamento: in questo utilizzo la frequenza di campionamento è 104 Hz. I dati utilizzati per l'algoritmo saranno quelli campionati dal giroscopio.

L'algoritmo utilizzato a tale scopo è ICHD [1] (Iterated Coherent Hilbert Decomposition), originariamente impiegato per l'elaborazione di segnali che risultano essere ripetitivi. Il lavoro svolto consiste nel miglioramento dell'algoritmo con l'obiettivo di essere utilizzato in tempo reale e, in questo caso specifico, con dati di natura inerziale; il fine ultimo è quello dell'implementazione sulla scheda ed ottenere così i dati via bluetooth. I dati ricavati possono essere utilizzati in lavori successivi per algoritmi volti al riconoscimento di attività motorie.

# 2. Richiami di teoria

## 2.1. Trasformata di Hilbert

La trasformata di Hilbert [2], a differenza di altre trasformate, non cambia il dominio di definizione. A causa di ciò può essere considerato un semplice filtro, detto filtro di Hilbert. Quest'ultimo ha una funzione di trasferimento del tipo:

$$H(f) = \begin{cases} -j & f > 0 \\ 0 & f = 0 \\ +j & f < 0 \end{cases}$$

Quindi si può notare che il filtro di Hilbert corrisponde ad uno sfasatore di 90°, mentre il modulo risulta essere unitario.

## 2.2. Decomposizione multicomponente AM-FM

La decomposizione multicomponente AM-FM [1] nasce allo scopo di essere usata per estrarre varie componenti di segnali ripetitivi. Potendo considerare il segnale come:

$$x(t) = \sum_{i=1}^{N} a_i(t) \cos(\varphi_i + \int_0^t \omega_i(\tau) d\tau)$$

Il metodo si compone di diversi passaggi elencati di seguito:

- Dato il segnale originale *x(t)* calcolare la sua controparte analitica *z(t)* dove $z(t) = x(t) + j\,H[x(t)]$, *H[]* rappresenta la trasformata di Hilbert;
- Calcolare la frequenza istantanea del segnale $\omega(t) = \frac{d\varphi(t)}{dt}$ dove $\varphi(t) = Im(\log(z))$;
- Stimare le frequenze istantanee medie effettuando un filtraggio passa basso della frequenza istantanea;
- Compiere una prima stima della fase della componente identificata: $\bar{\varphi}(t)$: $\int_0^t \bar{\omega}(\tau)d\tau$;
- Aggiustare la fase iniziale attraverso l'errore di fase e mantenerla sincronizzata, si sottrae la stima alla fase iniziale e il risultato ottenuto viene filtrato con un filtro passa basso. Il risultato di quest'ultima operazione viene sommata alla fase stimata;
- Effettuare la coherent demodulation, essa corrisponde a considerare $c(t) = z(t)e^{-j\bar{\varphi}(t)}$ e filtrare quest'ultimo. Dal risultato del filtraggio $\bar{c}(t)$ calcolandone il modulo si ottiene l'ampiezza $\hat{a}(t) = |\bar{c}(t)|$;
- Calcolare la componente selezionata $\hat{x}(t)$, essa sarà calcolata in tal modo: $\hat{x}(t) = \hat{a}(t)\cos(\hat{\varphi}(t))$;
- Sottrarre la componente selezionata al segnale originale.

Uno schema a blocchi riassuntivo delle operazioni è osservabile in Figura 2.

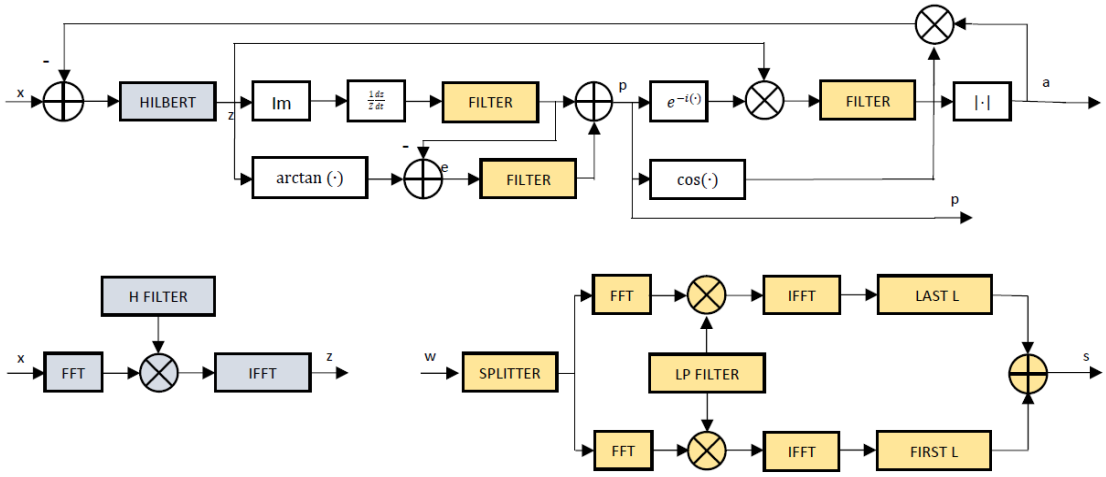


*Figura 2: Schema a blocchi algoritmo. Riprodotta con licenza CC BY-NC-ND [3].*

# 3. Matlab

## 3.1. Codice Matlab

Il codice originale consiste nella trasformazione dei passaggi presenti nel paragrafo 2.2 in codice Matlab. Quest'ultimo è rappresentato nelle Figure 3 e 4; la prima consiste nei filtraggi da effettuare, la seconda rappresenta il corpo principale del codice.

```
function s = trend (w)
global bf;
global bd;
    s = filter(bf, 1, [w(bd+1:-1:2); w; w(end-1:-1:end-bd)]);
    s = s(bd+bd+1:end);
end
```

*Figura 3: Codice filtraggi originale.*

Allo scopo indicato nei paragrafi precedenti, la progettazione dell'algoritmo

```
for i = 1:n
    % Hilbert transform:
    temp = hilbert([x(xhalf+1:-1:2); x; x(end-1:-1:end-xhalf)]);
    z = temp(xhalf+1:end-xhalf);

    % component selection:
    frq = imag(diff(z) ./ (z(1:end-1) + z(2:end)) * 2);
    bf = bf1; % use first corner frequency for phase filter
    media_frq = trend(frq);

    % oscillator:
    p(:,i) = [0; cumsum(media_frq)];
    % phase comparator:
    e(:,i) = unwrap(angle(z)) - p(:,i);
    % filter:
    media_delta = trend(e(:,i));
    % phase corrector:
    p(:,i) = p(:,i) + media_delta;

    % demodulator:
    media_amp = trend(z .* exp(-j * p(:,i)));
    a(:,i) = abs(media_amp);

    % Compute component:
    r(:,i) = a(:,i) .* cos(p(:,i));
    x = x - r(:,i);

end
```

*Figura 4: Codice Matlab originale.*

è stata svolta su Matlab attraverso l'uso di un segnale campionato dalla scheda e per mezzo dell'analisi dell'elaborazione di esso. Il nuovo codice presenterà una serie di modifiche a quello precedentemente studiato. Si è utilizzato un segnale di questo tipo poiché permette di essere il più veritiero possibile e, nelle fasi successive, di avere un confronto diretto tra i dati elaborati da Matlab e quelli della scheda. Il segnale in questione è visibile in Figura 5. Si è generato effettuando con la scheda un movimento simile ad un pendolo in modo da ottenere un segnale quasi sinusoidale la cui ampiezza decade nel tempo, in tal modo permette di simulare un'azione motoria ripetuta nel tempo. Inoltre, essendo quasi sinusoidale, permette di stimare il periodo per ogni finestra anche senza l'utilizzo dell'algoritmo così da semplificare la verifica dei dati ottenuti.

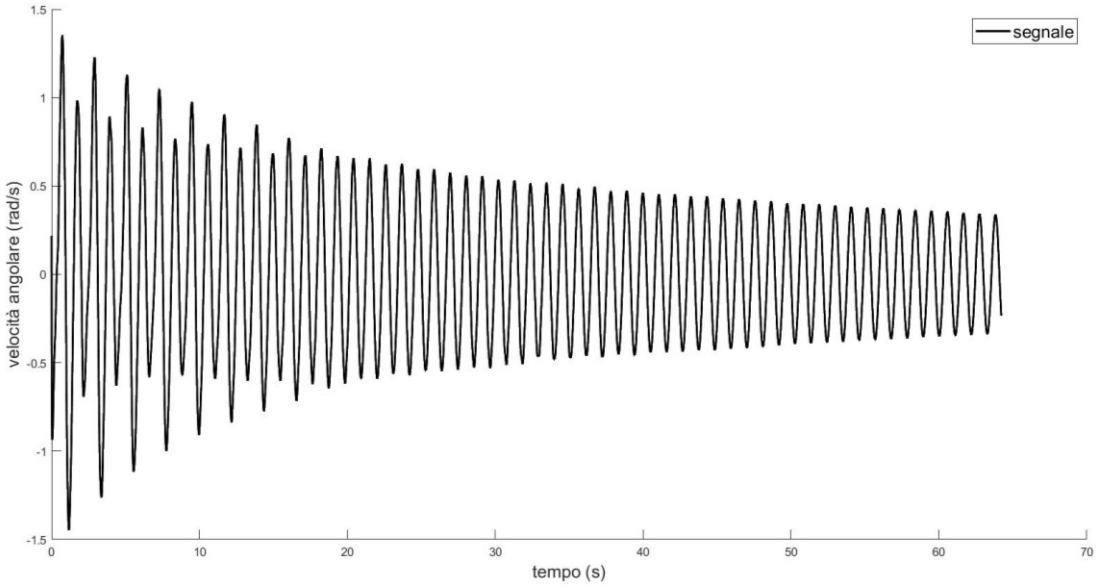


*Figura 5: Esempio di segnale campionato dalla scheda a scopo sperimentale.*

Al fine di simulare la sequenza di finestre in tempo reale, dove la finestra rappresenta l'intervallo di tempo di cui si vogliono ricavare i dati, si è scomposto il segnale in vari intervalli di tempo. L'obbiettivo è quello di migliorare i risultati ottenuti dall'algoritmo originale, osservabili in Figura 6. Osservando la figura, è possibile notare come ai bordi di ogni finestra siano presenti degli spike, consistenti in valori lontani dal resto della

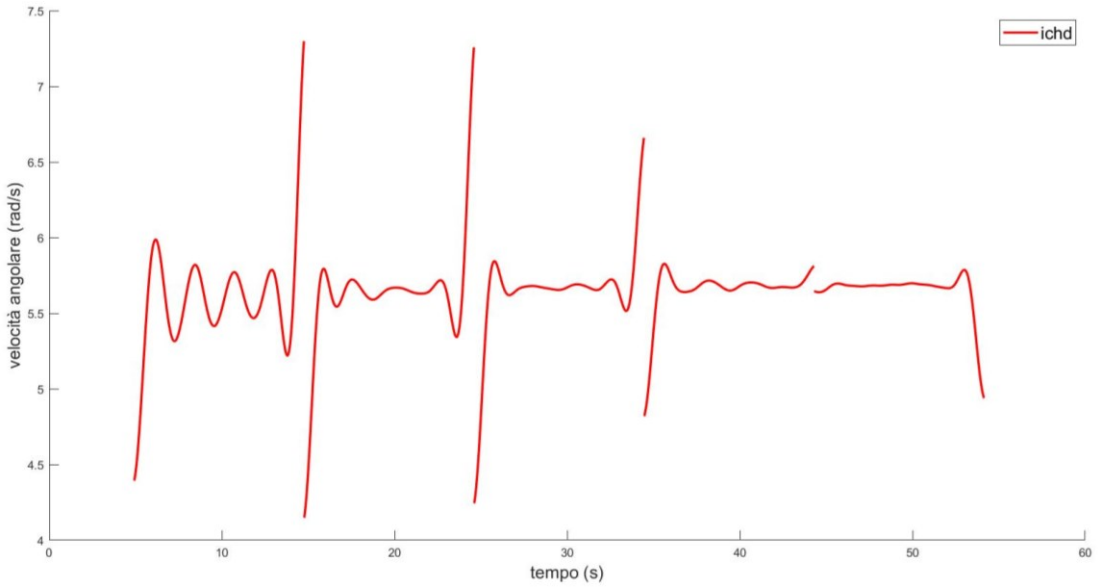


*Figura 6: Prima componente della frequenza con algoritmo originale.*

finestra. Nella versione che si è sviluppata si è ridotta la presenza di essi. L'intero segnale campionato rappresenta un'attività motoria, quindi risulta possibile considerare il passato e il futuro per ogni finestra di tempo. Per fare ciò il calcolo delle finestre viene ritardato di 512 campioni che corrispondono circa a 5 secondi rispetto all'origine del segnale; questi 512 campioni corrispondono al ritardo del filtraggio. Ogni campione rappresenta il valore all'istante temporale in cui vengono campionati i dati dal giroscopio. Impiegando il segnale campionato si è ottimizzato il codice per sfruttare la presenza di questi intervalli di tempo; pertanto, utilizzando il passato e il futuro in ogni finestra si limitano gli errori ai bordi di esse causati dal filtraggio.

L'algoritmo originale prevedeva una stima attraverso una specchiatura del presente per il passato e futuro, osservabile in Figura 3 e 4 rispettivamente nei filtraggi e nella trasformata di Hilbert. La prima modifica consiste nel rimuovere la specchiatura e sfruttare il ritardo. Questa modifica permette di migliorare i risultati nel caso in cui il segnale vari spesso tra una finestra e la successiva.

Allo scopo di analizzare e testare il nuovo algoritmo, il segnale è stato scomposto in blocchi corrispondenti a: passato, presente e futuro; la scomposizione è visibile in Figura 7. Il procedimento ha reso possibile la verifica di eventuali errori nella

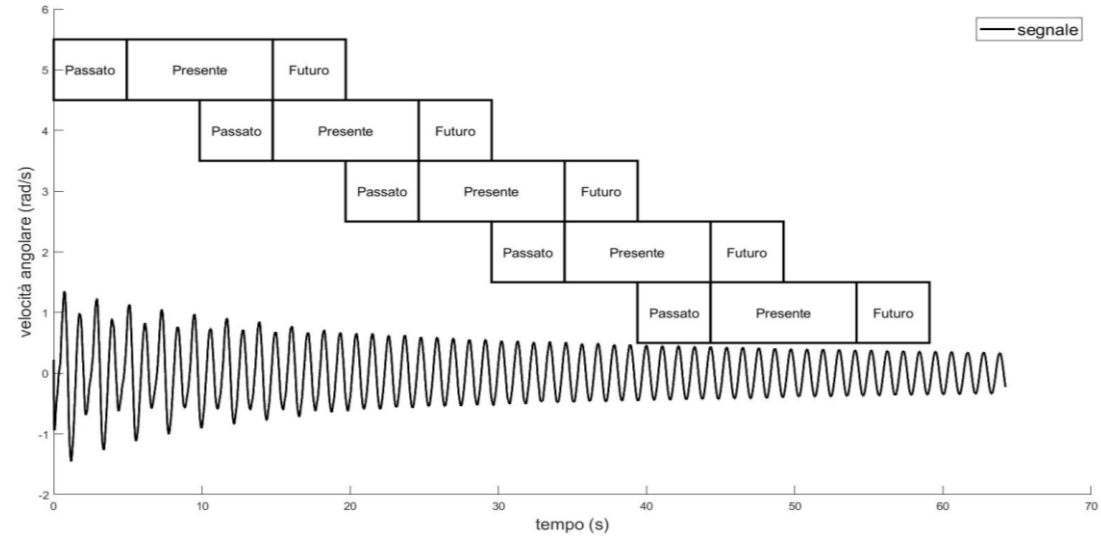


*Figura 7: Scomposizione del segnale in passato, presente e futuro.*

composizione della finestra da processare. La messa in coda dei dati nel campionamento in tempo reale eseguita sul micro verrà trattata nei paragrafi successivi. Il segnale composto da passato, presente e futuro viene fornito come input alla funzione. Le variabili presenti all'interno della funzione, in ordine temporale di utilizzo, sono:

- *z*, controparte analitica del segnale originale;
- *frq*, fase istantanea;
- *media_frq*, media delle fasi istantanee;
- *p*, fase del segnale;
- *e*, errore della fase del segnale;
- *media_delta*, correzione della fase;
- *media_amp*, componente demodulata;
- *a*, ampiezza demodulata;
- *r*, componente selezionata da sottrare al segnale originale.

Essendo il segnale composto da 2048 campioni, con una frequenza di campionamento dei dati pari a 104Hz, la prima finestra verrà generata solamente dopo aver campionato 20 secondi di dati; essa corrisponderà dunque all'intervallo di tempo compreso tra 5 e 15 secondi. Questo segnale viene utilizzato nel primo filtraggio: esso ha il compito di effettuare la trasformata di Hilbert. Il solo utilizzo del segnale costruito non comporta grossi miglioramenti, perciò si sono apportati degli sviluppi anche ai filtraggi. A tal fine è necessario dichiarare delle variabili in cui sarà salvata la seconda metà del presente; in tal modo per la finestra successiva sarà possibile utilizzare il passato. In Figura 8 è possibile notare queste differenze; in figura, la voce *hpast*, è una delle variabili precedentemente spiegate. Quest'ultimo passaggio comporta un incremento di memoria utilizzata, ma al contempo permette di ottenere dati più precisi nel caso di segnali che variano spesso.

```
% Hilbert transform:
temp = hilbert([x(xhalf+1:-1:2); x; x(end-1:-1:end-xhalf)]);
z = temp(xhalf+1:end-xhalf);
```

```
% Hilbert transform:
temp = hilbert(x);
temp3 = hpast(1:512,i);
hpast(:,i) = temp(1025:1536);
z = [temp3;temp(xhalf+1:end)];
```

*Figura 8: Modifiche al filtraggio Hilbert e costruzione di "z". A sinistra versione originale, a destra versione sviluppata.*

Il primo risultato ottenuto, corrispondente a 2048 campioni, è necessario a costruire la variabile *z*; di questi, i primi 512 campioni acquisiti vengono salvati in una delle variabili precedentemente nominate. La variabile *z* è composta dal passato calcolato nella precedente finestra e i 1024 campioni della seconda metà del segnale ottenuto dal primo filtraggio; nel caso della prima finestra si considera il passato nullo e ciò varrà per tutti i filtraggi. A questo punto la funzione prosegue selezionando la parte immaginaria della derivata ottenuta dal segnale *z* con il metodo delle differenze successive; questo passaggio corrisponde al calcolo delle fasi istantanee. Realizzato questo nuovo segnale si procede al filtraggio di esso in modo da ottenere la frequenza istantanea media: avverrà attraverso la funzione che si occupa di ciò.
Il filtraggio, come per quelli successivi, avviene tramite un filtro FIR (Finite Impulse Response) di lunghezza 1025 avendo una finestra di tipo Gaussiana. La funzione adibita a ciò richiede come dati il segnale presente da filtrare e il passato: attraverso

questi ultimi si costruisce il segnale da filtrare composto da passato, presente e futuro. Il futuro si otterrà specchiando gli ultimi valori del presente.
Prima di filtrare si effettua la copia degli ultimi 512 campioni del presente nella variabile corrispondente al passato in modo tale da avere sempre a disposizione quest'ultimo per le finestre successive.
Successivamente avviene il filtraggio vero e proprio che sfrutta il metodo dell'overlap and add il quale permette di velocizzare il tempo di esecuzione, usufruendo anche della FFT (Fast Fourier Transform). La descrizione completa di come avviene il filtraggio è riportata di seguito:

- Dividere il segnale in due blocchi di lunghezza pari alla metà della lunghezza del segnale;
- Applicare la FFT a 2048 punti al primo blocco e conseguentemente impiegherà zero padding per gli ultimi 1024 campioni;
- Effettuare il prodotto del risultato della FFT con il filtro FIR anch'esso trasformato con Fourier;
- Applicare la IFFT (Inverse Fast Fourier Transform) del prodotto precedentemente eseguito e salvare gli ultimi 1024 punti;
- Si ripete il procedimento per il secondo blocco con l'unica differenza che i primi 1024 punti vengono sommati ai 1024 salvati dal blocco 1.

Il nuovo codice corrispondente al filtraggio è osservabile in Figura 9. Le variabili *L* e *nfft* sono necessarie per effettuare l'overlap and add.

```
function [sfft,wpast] = trend_fft(w,B,wpast)
L    = single(1024);
nfft = single(2048);
assert(length(B) == nfft);
assert(length(wpast) == L/2);
assert(length(w) >= L);

if length(w) >= L+L/2
    tofilt = [wpast;w(1:L+L/2,:)];
else
    tofilt = [wpast;w;w(L-1:-1:L-512,:)];
end
assert(length(tofilt) == nfft);

X = fft(tofilt(1:L,:),nfft,1);
Y = ifft(X.*B,[],1);
sfft = Y(L+1:end);

X = fft(tofilt(L+1:end,:),nfft,1);
Y = ifft(X.*B,[],1);
sfft = sfft + Y(1:L);

wpast = w(513:1024);

end
```

*Figura 9: Nuova funzione adibita al filtraggio*

In uscita dalla funzione di filtraggio si ottiene la somma ottenuta dai due blocchi e il passato per il filtraggio della finestra successiva.
Il segnale in questione sarà chiamato *media_frq* e verrà utilizzato per costruire la frequenza del segnale originale. Per ottenerla si effettuerà la somma cumulativa di *media_frq*, ma il primo valore della fase in questo momento della funzione corrisponde a zero. Esso verrà corretto in seguito insieme agli altri valori.
Per poter calcolare l'errore della fase, salvato come variabile *e*,

è necessario utilizzare la funzione fornita da Matlab *unwrap* [4], non essendo continua, altrimenti non potrebbe essere filtrata. Questa operazione consiste nello shiftare gli angoli finché il salto tra angoli successivi è minore di π aggiungendo multipli di ±2π. Il filtraggio di questa correzione si differenzia dal resto dei filtraggi in quanto nel calcolo dell'errore non è necessario utilizzare il passato della correzione. Pertanto, il segnale costruito al fine di essere filtrato consiste in un passato e futuro ottenuti tramite la specchiatura dei valori relativi al presente. Questa è l'unica modifica con il filtraggio precedente, il resto viene effettuato nella medesima maniera. In effetti, è possibile notare in Figura 10 la specchiatura e il metodo dell'overlap and add.

```
function sfft = trend_delta(w,B)
L    = single(1024);
nfft = single(2048);
tofilt = [w(513:-1:2);w;w(L-1:-1:L-512,:)];
X = fft(tofilt(1:L,:),nfft,1);
Y = ifft(X.*B,[],1);
sfft = Y(L+1:end);
X = fft(tofilt(L+1:end,:),nfft,1);
Y = ifft(X.*B,[],1);
sfft = sfft + Y(1:L);
end
```

*Figura 10: Funzione filtraggio per errore di fase.*

Una volta ottenuta la correzione di fase, la si può utilizzare per correggere la fase iniziale; facendo la somma delle due fasi si ottiene la fase finale salvata su *p*.

Di seguito si procede al calcolo dell'ampiezza mediante la coherent demodulation, consistente nell'utilizzazione di un segnale immaginario ottenuto dal segnale *z* sfasato per la fase calcolata in precedenza. Questo segnale dispone delle ampiezze a bassa frequenza, perciò, filtrandolo con il passa basso usato precedentemente, si ottiene l'ampiezza. Per il filtraggio si è implementato lo stesso metodo utilizzato per la fase, infatti anche esso sfrutta la memorizzazione del passato e l'overlap and add.

Infine, le ultime modifiche apportate all'algoritmo vero e proprio, riguardano il calcolo della componente *r*, utilizzata per poter ricavare il segnale facente riferimento alla seconda componente. La prima versione del codice era applicata a un segnale di lunghezza 1024 campioni; essendo ora raddoppiata e dovendo sottrare l'ampiezza di lunghezza 1024 a quest'ultimo segnale, risulta necessario calcolare prima i 1024 campioni centrali per la componente e specchiarli nei primi e ultimi 512. Questa operazione viene svolta solamente per la prima componente, poiché, dovendo calcolare solo due componenti, non è necessario ottenerne una terza; questo comporterà dei vantaggi sui tempi di esecuzione. Le differenze tra le due versioni sono visibili in Figura 11.

```
% Compute component:
r(:,i) = a(:,i) .* cos(p(:,i));
x = x - r(:,i);
```

```
% Compute component:
if(i<2)
r(513:1536,i) = a(:,i) .* cos(p(:,i));
r(:,i) = [r(1025:-1:2+512,i);r(513:1536,i);r(1536-1:-1:1536-512,i)];
x = x - r(:,i);
```

*Figura 11: Differenze nella costruzione di "r". A sinistra versione originale, a destra nuova versione.*

Inoltre, il codice sviluppato è stato ottimizzato per poter essere convertito tramite il Coder [5] presente in Matlab. Le principali modifiche sono necessarie al fine di poter generare il codice C adatto alla scheda utilizzata. Siccome è necessario evitare l'allocazione dinamica di variabili in C, anche in Matlab saranno prefissate le

dimensioni delle variabili per poter evitare la frammentazione del heap della memoria. Altre ottimizzazioni effettuate riguardano il tipo di dato delle variabili: essendo la scheda a 32 bit si utilizzeranno dati con 4 byte di risoluzione al posto del tipo default pari a 8 byte. Il miglioramento ottenuto ai bordi delle finestre è osservabile in Figura 12. Le prime finestre soffrono ancora di errori ai bordi causati dalla decisione di

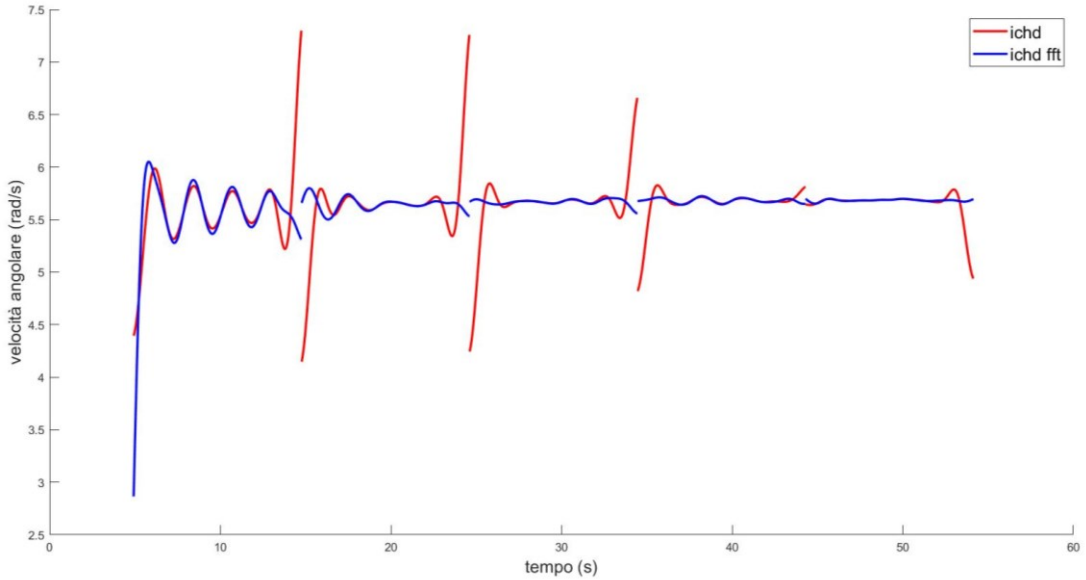


*Figura 12: Confronto della prima componente della frequenza delle due versioni.*

considerare un passato nullo nei filtraggi, ma con il proseguire dell'algoritmo si nota una riduzione degli spike. Inoltre, è possibile notare in Figura 13 come sia variato il codice rispetto alla versione iniziale.

```
for i = 1:2
% Hilbert transform:
temp = hilbert(x);
temp3 = hpast(1:512,i);
hpast(:,i) = temp(1025:1536);
z = [temp3;temp(xhalf+1:end)];
% component selection:
frq = imag(diff(z) ./ (z(1:end-1) + z(2:end)) * 2);
[media_frq,frqpast(:,i)] = trend_fft(frq(513:1536),B,frqpast(:,i));
media_frq =real(media_frq);
% oscillator:
p(:,i) = [0; cumsum(media_frq(1:1023))];
% phase comparator:
e(:,i) = (unwrap(angle(z(513:1536)))) - p(:,i);
% filter:
media_delta = trend_delta(e(:,i),B);
media_delta = real(media_delta);
% phase corrector:
p(:,i) = p(:,i) + media_delta;
% demodulator:
[media_amp,mediapast(:,i)] = trend_fft(z(513:1536) .* exp(-j * p(:,i)),B,mediapast(:,i));
a(:,i) = abs(media_amp);
% Compute component:
if(i<2)
r(513:1536,i) = a(:,i) .* cos(p(:,i));
r(:,i) = [r(1025:-1:2+512,i);r(513:1536,i);r(1536-1:-1:1536-512,i)];
x = x - r(:,i);
end
```

*Figura 13: Nuovo codice Matlab.*

### 3.2. Matlab Coder

Il Matlab Coder è l'Add-On fornito da Matlab; quest'ultimo permetterà di convertire il codice scritto su Matlab in C. Esso necessita a sua volta di ulteriori Add-On, forniti anch'essi da Matlab. Quelli necessari sono: Embedded Coder [6], Embedded Coder Support Package For ARM Cortex-M Processor [7] ed Embedded Coder Support Package for STMicroelectronics Discovery Boards [8]. L'uso di questi Add-On è necessario per ottimizzare il Coder alla generazione di codici adatti a sistemi embedded.

Avviando il Coder, il primo passaggio consiste nel fornire la funzione che si intende convertire. Successivamente si potrà fornire uno script che implementa la funzione; questo passaggio permetterà di generare automaticamente i tipi di dati in ingresso e in uscita della funzione oppure potrà essere effettuato anche manualmente per ogni variabile.

Dopo di che, prima di poter generare il codice attraverso il Coder, si effettuerà un controllo per verificare la possibilità di realizzare il codice finale in linguaggio C.

Al fine di superare quest'ultimo, è necessario applicare alcune modifiche al codice Matlab: esse consistono nel dichiarare specificatamente i parametri per la generazione del filtro utilizzato. Una volta eseguito il controllo ed avuto esito positivo da esso, sarà possibile dichiarare le specifiche per il codice da generare. Quest'ultime faranno riferimento alla scheda utilizzata e all'ottimizzazione che si vuole ottenere della funzione implementata.

Le specifiche fornite per la generazione del codice sono:

- Disabilitazione della dichiarazione di dati di dimensione variabile;
- Uso di matrici invece di vettori più lunghi;
- La quantità massima di memoria dello stack;
- Uso della libreria CMSIS [9];
- Ottimizzazione per determinare i dati costanti e i dati variabili;
- Toolchain: GNU Tools for ARM embedded Processors [10].

L'allocazione di variabili statiche e non dinamiche è necessaria affinché non avvenga una frammentazione del heap della memoria del microcontrollore. L'utilizzo di matrici rispetto a vettori più lunghi permette di ottenere variabili più facilmente gestibili anche per operazioni successive, specialmente per il calcolo delle componenti, in modo che ogni riga corrisponda ad una componente.

La scheda, avendo una quantità pari a 8KiB di memoria per lo stack, ha implicato la limitazione dell'uso di quest'ultimo. Il Coder ha quindi generato le principali variabili in modo che esse vengano salvate nella memoria flash della scheda. L'utilizzo della libreria CMSIS permette di sostituire le funzioni generiche realizzate dal Coder con funzioni ottimizzate. Al fine di tale utilizzo, quest'ultime sono necessarie nell'uso di operazioni relative all'ambito del DSP come la Fast Fourier Transform, utilizzata frequentemente nel codice. Essendo presente nel processore ARM Cortex M4 un'unità di calcolo adibita ad operazione in ambito DSP, consente un risparmio notevole nei tempi di esecuzione.

Le funzioni ottimizzate sono fornite da ARM e sono compatibili con il tipo di processore presente sulla scheda. Per poter sfruttare questa funzionalità del Coder è necessario fornire il tipo di Hardware sul quale il codice deve essere elaborato. Non essendo presente il microcontrollore utilizzato dalla scheda nell'elenco fornito da

Matlab, si è proceduto scegliendo una scheda che utilizzasse lo stesso processore ARM al fine di poter ottenere le funzioni corrette nell'uso di operazioni relative al DSP. La scheda in questione è prodotta dalla STMicroelectronics ed è la: STM32F4 Discovery. Il tipo di Toolchain è stato fornito per poter permettere al Coder di utilizzare tipi di dati specifici a processori ARM.
Per poter determinare il tipo di variabile, se costante o variabile, il Coder analizza il codice Matlab per un tempo più lungo rispetto al tempo base; in questo modo, nel codice ICHD, il Coder permetterà di ottenere i filtri come un vettore di valori costanti senza dover generarli ad ogni esecuzione.
Il codice contenente la funzione ICHD e i filtraggi verrà generato in un unico file ed inoltre verranno generati altri file che contengono le dichiarazioni di funzioni e variabili. Infine, fornite tutte le specifiche necessarie alla generazione, verrà creato il codice della funzione. Esso verrà implementato sulla scheda e ulteriormente ottimizzato manualmente come descritto nel paragrafo successivo.

# 4. Implementazione

## 4.1. Implementazione ICHD e ottimizzazione

Il codice generato dal Matlab Coder può essere ulteriormente ottimizzato e, a tal fine, prima di essere aggiunto all'intero firmware, è stato analizzato e sono state apportate le necessarie modifiche dove possibile. I principali problemi riscontrati sono il numero di operazioni superflue che il codice riportava e la presenza di variabili non necessarie che causavano un aumento di memoria occupata. Di seguito verranno riportate le operazioni che sono state rimosse per cercare di ridurre il tempo di esecuzione dell'algoritmo. I nomi delle variabili corrispondo alle stesse presenti nel codice Matlab; perciò, è possibile fare riferimento alla descrizione precedentemente svolta per il loro significato.
La prima modifica avviene nella costruzione della variabile *z*. In questo caso, il codice generato dal Coder effettuava la copia del passato del segnale ottenuto da Hilbert in una variabile per poi copiare quest'ultima in *z*: si è ottimizzato effettuando la copia diretta del passato su *z*. Quest'azione dovrebbe essere svolta dal compilatore, ma per esserne certi si è preferito specificarlo. Questa sottile differenza è illustrata in Figura 14.

```
for (ixLead = 0; ixLead < 512; ixLead++) {
  work_im = hpast[i][ixLead].re;
  tmp2_re = hpast[i][ixLead].im;
  hpast[i][ixLead] = temp[ixLead + 1024];
  z[ixLead].re = work_im;
  z[ixLead].im = tmp2_re;
}
```

```
for (ixLead = 0; ixLead < 512; ixLead++) {
  z[ixLead].re = hpast[i][ixLead].re;
  z[ixLead].im = hpast[i][ixLead].im;
  hpast[i][ixLead] = temp[ixLead + 1024];
}
```

*Figura 14: Costruzione di "z". A sinistra versione generata dal Matlab Coder, a destra versione ottimizzata.*

Il passaggio successivamente ottimizzato consiste nel comporre il segnale *frq* da filtrare. Quest'ultimo veniva costruito tramite l'uso della funzione *memcpy*, due cicli *for* e un ciclo *for* necessario a memorizzare il passato per la finestra successiva, come indicato in Figura 15. Per ridurre il numero di operazioni in codice assembly, i tre cicli *for* sono stati unificati in uno solo. I due cicli servivano a copiare il passato della finestra precedente e per specchiare il futuro. Utilizzando un unico ciclo *for* si riducono le

```
for (ixLead = 0; ixLead < 512; ixLead++) {
  tofilt[ixLead] = frqpast[i][ixLead];
}

memcpy(&tofilt[512], &frq[512], 1024U * sizeof(float));
for (ixLead = 0; ixLead < 512; ixLead++) {
  tofilt[ixLead + 1536] = frq[1534 - ixLead];
}

for (ixLead = 0; ixLead < 512; ixLead++) {
  frqpast[i][ixLead] = frq[ixLead + 1024];
}
```

*Figura 15:Versione generata da Matlab Coder per la costruzione di "frq".*

operazioni in codice assembly perché questo tipo di ciclo necessita di molti passaggi. Quindi il codice assume la forma della Figura 16.
Successivamente alle trasformate di Fourier si è notato che i dati necessari ad ottenere i segnali trasformati venivano prima copiati in una variabile e di seguito eseguito il calcolo; in questo caso, come nel caso della costruzione di *z*, si è evitato ciò.

```
memcpy(&tofilt[512], &frq[512], 1024U * sizeof(float));
for (ixLead = 0; ixLead < 512; ixLead++) {
  tofilt[ixLead + 1536] = frq[1534 - ixLead];
  tofilt[ixLead] = frqpast[i][ixLead];
  frqpast[i][ixLead] = frq[1024+ixLead];
}
```

*Figura 16: Versione ottimizzata della costruzione di "frq".*

Nel calcolo di *media_delta* la versione originale del codice prevedeva di strutturare la funzione in parte reale e in parte immaginaria, ma, poiché la parte immaginaria è nulla, si è modificata la dichiarazione. La modifica consiste nel dichiarare *media_delta* come *float*, il che permette di risparmiare una quantità di memoria pari a 4 KiB. In Figura 17 è possibile notare la precedente dichiarazione.

```
typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;
```

*Figura 17: Tipo di dato utilizzato dal Matlab Coder per media_delta.*

Inoltre, nell'operazione di copia del segnale anti trasformato viene copiata solamente la parte reale, così da ridursi le operazioni che il firmware deve svolgere; nello specifico, non deve effettuare 1024 copie e somme che risultano non necessarie.
Queste operazioni di copie ripetute avvengono spesso nel firmware, quindi, analizzando quest'ultimo, si è ridotto il codice ad operazioni dirette. Anche nel calcolo delle somme cumulative avviene ciò: il codice fornito dal Coder copia tutti i valori di *media_delta* su una variabile ed esegue la somma cumulativa; successivamente, copia la variabile su *p*.
Nell'ottimizzare il codice si evitano i passaggi precedenti relativi alla prima copia di *media_delta*. È possibile osservare le differenze delle due versioni in Figura 18 e di seguito verranno spiegate.

```
/*  oscillator: */
for (ixLead = 0; ixLead < 1023; ixLead++) {
  b_x[ixLead] = media_delta[ixLead].re;
}

for (ixLead = 0; ixLead < 1022; ixLead++) {
  b_x[ixLead + 1] += b_x[ixLead];
}

p[i][0] = 0.0F;
for (ixLead = 0; ixLead < 1023; ixLead++) {
  p[i][ixLead + 1] = b_x[ixLead];
}
```

```
/*  oscillator: */
p[i][0] = 0.0F;
for(ixLead = 0 ;ixLead < 1023;ixLead++){
    media_delta[ixLead + 1] += media_delta[ixLead];
    p[i][ixLead+1]=media_delta[ixLead];
}
```

*Figura 18: Differenza tra versione originale e ottimizzata. A sinistra, versione originale generata da Matlab Coder. A destra, versione ottimizzata.*

Per ottenere la variabile *p* si esegue la somma cumulativa attraverso un ciclo *for* direttamente su *media_delta*, il cui risultato verrà poi copiato su *p*. Nel codice iniziale

veniva eseguito attraverso tre cicli *for* e anche in questo caso si è ridotto il numero di cicli ad uno solo. Il primo valore di *p* è posto uguale a zero prima di effettuare la copia. Così procedendo si è potuto eliminare la dichiarazione della variabile utilizzata temporaneamente: tale modifica ha permesso di risparmiare 4092 byte. A causa del cambiamento del tipo di dato di *media_delta* si è dovuta riscrivere una parte di codice. Quest'ultima corrisponde al calcolo della demodulazione; essa è rappresentata in Figura 19. *Media_delta* veniva usata come variabile temporanea, ma all'interno del codice è presente una variabile a questo scopo e, pertanto, si è sostituita *media_delta* con la variabile temporanea.

```
/*  phase corrector: */
/*  demodulator: */
for (ixLead = 0; ixLead < 1024; ixLead++) {
  br = p[i][ixLead] + (media_delta[ixLead].re + X[ixLead].re);
  p[i][ixLead] = br;
  if (-br == 0.0F) {
    work_im = 1.0F;
    tmp2_re = 0.0F;
  } else {
    work_im = arm_cos_f32(-br);
    tmp2_re = arm_sin_f32(-br);
  }

  br = z[ixLead + 512].re;
  tmp2_im = z[ixLead + 512].im;
  work_re = br * work_im - tmp2_im * tmp2_re;
  tmp2_re = br * tmp2_re + tmp2_im * work_im;
  media_delta[ixLead].re = work_re;
  media_delta[ixLead].im = tmp2_re;
}


for (ixLead = 0; ixLead < 512; ixLead++) {
  temp[ixLead] = mediapast[i][ixLead];
}

memcpy(&temp[512], &media_delta[0], 1024U * sizeof(creal32_T));
for (ixLead = 0; ixLead < 512; ixLead++) {
  temp[ixLead + 1536] = media_delta[1022 - ixLead];
}
```

*Figura 19: Uso di "media_delta" come variabile temporanea nel codice originale.*

Inoltre, nel preparare quest'ultima al filtraggio si è svolta la medesima ottimizzazione attuata alla costruzione di *z* riguardo al numero di cicli. L'ultima ottimizzazione che si è svolta consiste nell'eliminare la dichiarazione di *r* in modo da risparmiare 8 KiB. È possibile eliminare *r* andando a utilizzare la variabile *tofilt* al suo posto, variabile creata allo scopo di essere utilizzata per i filtraggi. Inoltre, anche la costruzione di *tofilt* viene eseguite allo stesso modo di *z*.

Infine, per implementare l'algoritmo nel firmware è stato necessario includere le librerie generate dal Coder e le librerie del pacchetto CMSIS nel Makefile del firmware. Le ottimizzazioni, riducendo i tempi di esecuzione, permettono di consumare meno energia perché il processore ha un consumo maggiore mentre svolge operazioni; inoltre, riducendo la memoria occupata, si permette di avere ulteriore memoria libera per lavori successivi.

## 4.2. Modifiche firmware e gestione dati

L'implementazione sul micro ha richiesto diverse modifiche al firmware originale. Le principali sono necessarie per poter campionare e inviare continuamente i dati inerziali durante l'esecuzione dell'algoritmo in quanto risulta essere molto maggiore del tempo di campionamento; la perdita di dati sarebbe significativa se il campionamento venisse fermato. Queste modifiche riguardano principalmente la gestione degli interrupt e degli observer [11] per il BLE (Bluetooth Low Energy).

È stato necessario modificare le varie priorità degli interrupt poiché sia il campionamento sia la trasmissione avvengono attraverso l'uso di interrupt e

dispongono di funzioni con priorità diverse. La Nordic dichiara che le funzioni vengono chiamate in ordine di priorità dell'interrupt e, quando viene chiamato un interrupt all'interno di un'altra funzione, deve disporre di una priorità maggiore o uguale per poter essere eseguito. Le priorità della Nordic si possono osservare in Figura 20.
Il campionamento avviene attraverso una trasmissione I2C. Essa disponeva precedentemente del livello 2, come si può notare tra i più prioritari, perciò, il livello di priorità è stato impostato a 7, altrimenti avrebbe avuto la precedenza rispetto al SoftDevice [12]. Esso consiste in un insieme di protocolli wireless che integrano i SoC (system on a chip) della serie nrf5 della Nordic. Questi vengono forniti dalla Nordic come file binari già compilati. Considerando che quest'ultimo viene utilizzato per la trasmissione dei dati via bluetooth, non avrebbe consentito la trasmissione.
Un'altra modifica apportata per poter effettuare lo streaming senza interrompere il campionamento è stata la gestione degli observer BLE. Quest'ultimi risultano essere dei gestori per gli eventi del bluetooth, che si occupano della gestione delle funzioni e i loro parametri.

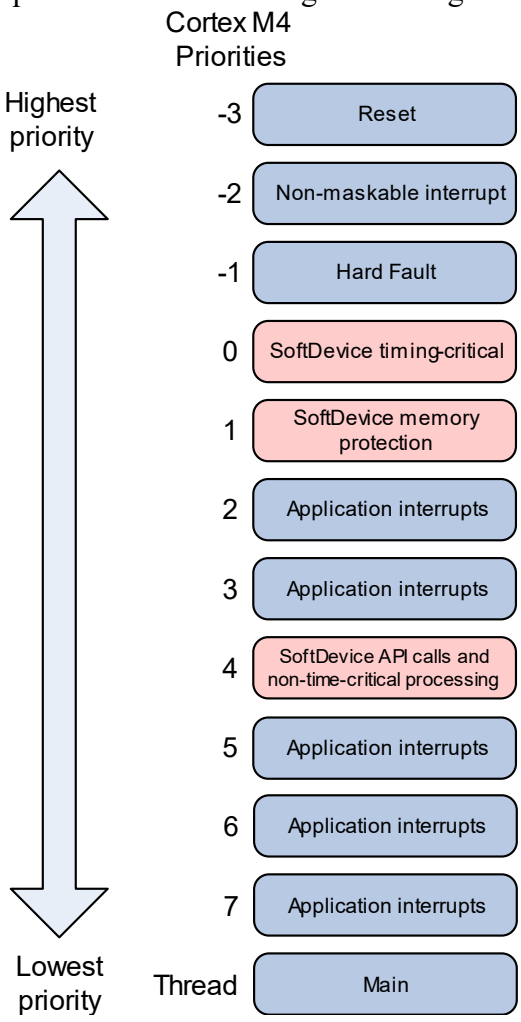

*Figura 20: Priorità Cortex M4. [14]*

All'interno del codice possono esistere diversi observer. Essi erano precedentemente configurati per poter essere utilizzati dallo scheduler degli eventi il quale non permetteva di inviare i dati, questo perché lo scheduler ha la priorità inferiore all'interno del firmware. Inoltre, lo scheduler consiste nel mettere in coda le attività da svolgere ed eseguirle nell'ordine in cui sono state aggiunte. È necessario usufruire degli observer tramite interrupt in modo tale da configurarlo per lo scopo in questione: così facendo gli observer hanno la stessa priorità della funzione da cui vengono chiamati. Questa modifica comporta un cambiamento nello svolgimento delle operazioni bluetooth che l'intero firmware deve intraprendere. Ciò comporta una maggior difficoltà nella stesura del firmware, ma permetterà di ottimizzarlo maggiormente.
Conseguentemente, è stato modificato il firmware in tutte le parti in cui è utilizzato un observer: dove precedentemente le funzioni venivano chiamate attraverso lo scheduler, a seguito delle modifiche apportate, sono chiamate in maniera diretta come funzioni. Al contrario, laddove le funzioni non sono adatte ad essere utilizzate durante interrupt, se chiamate dagli observer, devono essere configurate per essere impiegate con lo scheduler. Si provvederà ad aggiungerle in coda allo scheduler durante l'esecuzione dell'interrupt, ma queste verranno eseguite successivamente.

Allo scopo di gestire i vari eventi relativi al campionamento dei dati, è stata implementata una macchina a stati finiti che permette di selezionare il caso adatto. Il tipo di caso viene decretato attraverso le caratteristiche bluetooth, cioè a seconda di quale trasmissione bluetooth verrà attivato lo stato inerente.
Nel firmware sono presenti diverse caratteristiche; a tale scopo viene utilizzata quella per lo streaming dei dati, già esistente sul codice. Inoltre, si è dovuto creare una nuova caratteristica necessaria ad inviare i dati ottenuti dall'algoritmo ICHD: essa funziona sia in modalità lettura che in modalità *notify*. Quest'ultimo metodo consiste nel far abilitare la trasmissione dal client; in questo modo, ogni volta in cui saranno disponibili nuovi dati il server li invierà al client. Questo metodo è più veloce della lettura perché il client non deve richiedere sempre se la caratteristica è stata aggiornata.
Nel firmware sono presenti dei servizi bluetooth che contengono delle caratteristiche; le due caratteristiche utilizzate sono all'interno del servizio relativo ai dati provenienti dal sensore *lsm6dso*. I servizi [13] sono una raccolta di dati che possono contenere delle caratteristiche; il Bluetooth Special Interest Group (Bluetooth SIG) fornisce dei servizi predefiniti. Le caratteristiche [13] risultano essere dove i dati e le informazioni vengono effettivamente rappresentate. Le caratteristiche relative ai dati provenienti dal sensore vengono presentate all'utente tramite l'uso della applicazione "nRF Connect for Mobile" fornita da Nordic, come in Figura 21.

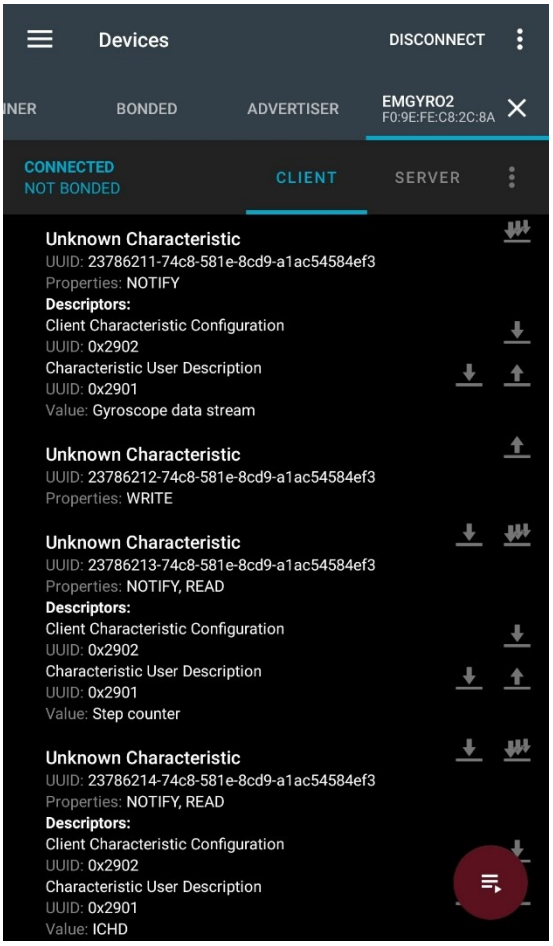


*Figura 21: Interfaccia applicazione Nordic delle caratteristiche bluetooth.*

I vari casi che possono essere attivati dalle caratteristiche sono:

- Streaming e ICHD spenti;
- Streaming acceso e ICHD spento;
- Streaming spento e ICHD acceso;
- Streaming e ICHD accesi.

I vari stati corrispondono ognuno ad un numero e la loro dichiarazione è stata eseguita come mostrato in Figura 22.
Per poter abilitare i vari stati è necessario richiedere i dati via bluetooth.
Nei casi in cui ICHD risulta acceso è previsto un ulteriore stato che prevede il salvataggio dei dati campionati in un buffer che, successivamente, sarà copiato nei dati da processare nella finestra successiva. Inoltre, in questi casi, per il calcolo dell'algoritmo vengono utilizzati i dati provenienti dall'asse x del giroscopio.
A prescindere che venga attivato lo streaming o l'algoritmo ICHD, la scheda inizierà il campionamento dei dati: ad ogni lettura avvenuta tramite la trasmissione I2C il sistema entrerà nella macchina a stati.

```
typedef enum {
        OFF,                    //0:stream off
        ONLY_DATA,              //1:streaming only data gyro.
        DATA_PLUS_ICHD, // 2:streaming data and ichd
        ONLY_ICHD,              //3:streaming only ichd
        BUFFER_ONLY,    //4:buffer only ichd
        BUFFER_PLUS,            //5:both data
        WAIT_NOTIFY             //6:for test data

}stream_type;
```

*Figura 22: Stati possibili di dati richiesti.*

Il codice utilizzato come macchina a stati viene di seguito riportato in maniera scritta e non grafica a causa della sua lunghezza:

```
switch(stream_status) {
        case OFF:
                break;
        case ONLY_DATA:                 //caso solo dati
                p=gyro_buf + gyro_buf_tail++;
                gyro_buf_tail &= GYRO_BUF_IDX_MASK;
                p->timestamp = NRF_TIMER1->CC[3];
                p->status = 0;
                memcpy(p->data, gyro_data, 14);
                if (((gyro_buf_tail - gyro_buf_head) & GYRO_BUF_IDX_MASK) >= 1) {
                                app_sched_event_put(NULL, 0, received_gyro_data_block);
                }
                break;
        case DATA_PLUS_ICHD:        //caso dati e ichd
                data_ichd[ndata]=gyro_data[1];
                p=gyro_buf + gyro_buf_tail++;
                gyro_buf_tail &= GYRO_BUF_IDX_MASK;
                p->timestamp = NRF_TIMER1->CC[3];
                p->status = 0;
                memcpy(p->data, gyro_data, 14);
                ndata++;
                if (((gyro_buf_tail - gyro_buf_head) & GYRO_BUF_IDX_MASK) >= 1) {
                                app_sched_event_put(NULL, 0, received_gyro_data_block);
                }
                break;
        case ONLY_ICHD:                 //caso ichd.
                data_ichd[ndata]=gyro_data[1];
                ndata++;
                app_sched_event_put(NULL, 0, received_gyro_data_block);
                break;

        case BUFFER_ONLY: //stato buffer: riempimento vettore buffer.

                buffer_ichd[ndata-1024]=gyro_data[1];
                ndata++;
                break;
        case BUFFER_PLUS:  //stato buffer e dati: riempimento vettore e invio dati.
                p=gyro_buf + gyro_buf_tail++;
                gyro_buf_tail &= GYRO_BUF_IDX_MASK;
                p->timestamp = NRF_TIMER1->CC[3];
                p->status = 0;
                buffer_ichd[ndata-1024] = gyro_data[1];
```

```
                memcpy(p->data, gyro_data, 14);
                ndata++;
                if (((gyro_buf_tail - gyro_buf_head) & GYRO_BUF_IDX_MASK) >= 1) {
                                invio_buffer();
                }
                                                break;
        case WAIT_NOTIFY:
                break;
        }
```

La descrizione degli stati riportati nel codice precedente avviene di seguito.
Nello stato che prevede lo streaming, i dati campionati verranno preparati alla trasmissione bluetooth di essi, si può notare che nella preparazione di essi viene inserito anche un riferimento temporale; infine, verranno inviati.
Nello stato in cui è attivo solo ICHD, i dati verranno copiati in un vettore, chiamato *ichd_data*, necessario come input della funzione: il vettore rappresenta la finestra temporale dei dati.
Nello stato in cui sono entrambi attivi, verranno svolte ambedue le attività spiegate precedentemente.
Nello stato che comprende il buffer avviene il medesimo procedimento svolto dallo stato ICHD, l'unica differenza consiste nel copiare i dati in un buffer piuttosto che nel vettore principale.
Al fine di mantenere il conteggio del numero di campioni ottenuti all'interno della macchina a stati, viene incrementata una variabile: essa è utilizzata nei casi in cui ICHD sia attivo. In ogni stadio viene chiamata la funzione *received_gyro_data_block*.
Il codice relativo a questa funzione è riportato di seguito:

```
void received_gyro_data_block (void *p_event_data, uint16_t event_size)
{
  if(ndata==1023){
    ichdfft_initialize();
  }
  if((ndata==2048) || (ndata>2048)){
    static stream_type temp_stream;
    temp_stream = stream_status;
    ndata=1024;
    if(temp_stream == DATA_PLUS_ICHD){
      stream_status = BUFFER_PLUS;
    }else{
      stream_status = BUFFER_ONLY;
    }
    ichd_call();
    nrf_gpiote_event_disable(2);
    memmove(&data_ichd[0],&data_ichd[1024],1024*sizeof(float));
    memcpy(&data_ichd[1024],buffer_ichd,256*sizeof(float));
    stream_status = temp_stream;
    nrf_gpiote_event_enable(2);
```

```
    }
    if (pkt_q_lens[2] <= 1) ble_gyro_packet_retry(&m_gyro);
    if(stream_status==ONLY_DATA|| stream_status == DATA_PLUS_ICHD)
ble_gyro_packet_prepare(&m_gyro);
}
```

In essa sono presenti due verifiche: la prima, se il numero di campioni è pari a 1023, necessaria al primo avvio per impostare il passato nullo; la seconda, se il numero di campioni corrisponde a 2048. Se verificata la seconda, il codice procede salvando lo stato in cui si trova il sistema per poter così successivamente modificare lo stato in questione con lo stato buffer e conseguentemente la variabile di conteggio verrà posta pari a 1024. Essa verrà così impostata per poter effettuare un conteggio di altri 1024 campioni corrispondenti all'intervallo di una finestra.
A questo punto, il firmware procederà chiamando la funzione *ichd_call* che consiste nella chiamata all'algoritmo e al calcolo di frequenza e ampiezza. Il codice di *ichd_call*;

```
void ichd_call (){
        ichdfft(data_ichd,result_a,result_p);
        qsort(result_a[0],1024, sizeof(float),compare);
        as[0]=result_a[0][511]+ result_a[0][512];
        qsort(result_a[1],1024, sizeof(float),compare);
        as[1]=result_a[1][511]+ result_a[1][512];
        as[0]=as[0]/2;
        as[1]=as[1]/2;
        for(int j=0;j<2;j++){
                for(int i=0;i<1023;i++){
                        if(i==0){
                                derivate[j][i]=(result_p[j][i+1]-result_p[j][i]);
                        }else if(i<1022){
                                derivate[j][i]=(result_p[j][i+2]-result_p[j][i])*0.5;
                        }else if (i==1022){
                                derivate[j][i]=(result_p[j][i+1]-result_p[j][i]);
                        }
                }
        }
        qsort(derivate[0],1024,sizeof(float),compare);
        fs[0]=derivate[0][511]+derivate[0][512];
        qsort(derivate[1],1024,sizeof(float),compare);
        fs[1]=derivate[1][511]+derivate[1][512];
        fs[0]=fs[0]/2;
        fs[1]=fs[1]/2;
        update_ichd_data(&m_gyro);
}
```

Il calcolo dell'ampiezza consiste nell'ordinare i dati ricevuti da ICHD e nel calcolarne la mediana; invece, per la frequenza, è necessario dapprima effettuare la derivata con il metodo delle differenze successive e in seguito calcolarne la mediana. Per ordinare i

dati si è utilizzata la funzione *qsort* presente nelle librerie standard del C. Infine, l'ultima operazione da svolgere in *ichd_call*, consiste nell'invio dei dati calcolati via bluetooth nella corrispondente caratteristica.
Durante l'esecuzione della funzione *ichd_call* continuano ad essere generati interrupt dalla trasmissione I2C e, per questo motivo, si è modificato lo stato per la macchina a stati finiti sfruttando in tal modo lo stato di buffer. In questo stato, per permettere l'invio dei dati campionati, è necessario chiamare la funzione per l'invio; è proprio quest'ultima che sfrutta la modifica spiegata precedentemente sugli interrupt e gli observer.
Conclusa l'esecuzione, è necessario copiare i dati presenti all'interno del buffer nel vettore principale: nel fare ciò, si è disabilitata la ricezione di interrupt relativi alla trasmissione I2C durante la copia e viene riabilitata subito dopo. Per eseguire la copia, i dati presenti nel vettore del segnale vengono shiftati di 1024 campioni e viene copiato l'intero buffer nei valori successivi. Eseguito quest'ulteriore passaggio, se abilitato, avviene lo streaming dei dati bluetooth. Quest'ultimo passaggio avviene anche se è solamente abilitato lo streaming. Inoltre, durante la fase di test e di analisi dei risultati del firmware è stato implementato un ulteriore caso, il quale serviva a evitare l'avvio dello stato *ONLY_ICHD*, in quanto, se abilitato, attiva il campionamento.
Attraverso questo nuovo stato, per abilitare il campionamento è necessario attivare successivamente anche lo streaming dei dati; a questo punto il sistema passa al caso in cui sono entrambi attivi. Questo nuovo stato permette di ottenere i dati utilizzati per l'algoritmo ICHD anche via bluetooth.
Essendo trasmessi i dati di tutti gli assi dell'accelerometro e giroscopio e i risultati di ICHD, si sono generati due file che li contengono rispettivamente. Infine, attraverso l'utilizzo del Matlab, il primo file viene adoperato per leggere i dati selezionando l'asse utilizzato dalla scheda e calcolare le finestre: i risultati ottenuti vengono confrontati con quelli del microcontrollore.

## 5. Risultati

I dati ottenuti dalla scheda sono stati confrontati con quelli ottenuti da Matlab con il segnale campionato. Nella sottostante Tabella 1 è possibile osservare l'errore presente tra il calcolo effettuato dalla scheda e il Matlab. Si nota che l'errore ottenuto è quasi trascurabile essendo minore dello 0.2% per tutti i dati.

| Numero Finestra | Prima componente ampiezza | Seconda componente ampiezza | Prima componente frequenza | Seconda componente frequenza |
|---|---|---|---|---|
| **1** | **0,0012%** | **0,0018%** | **0,0017%** | **0,0131%** |
| **2** | **0,0013%** | **0,0017%** | **0,0001%** | **0,0033%** |
| **3** | **0,0014%** | **0,0036%** | **0,0000%** | **0,1730%** |
| **4** | **0,0014%** | **0,0300%** | **0,0030%** | **0,0031%** |
| **5** | **0,0015%** | **0,0008%** | **0,0009%** | **0,0019%** |

*Tabella 1: Errore ottenuto dal confronto dati tra scheda e Matlab.*

Inoltre, è possibile osservare in Figura 23 l'andamento del segnale, delle ampiezze e delle frequenze calcolate tramite Matlab.

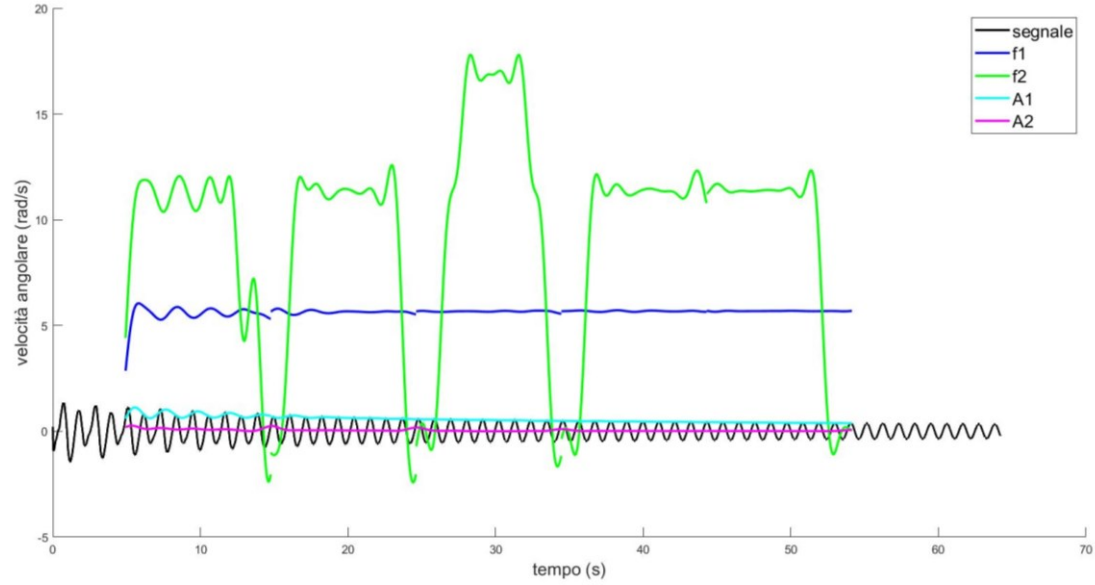


*Figura 23: Rappresentazione finale del segnale e dei dati elaborati.*

Sono stati anche calcolati i consumi di corrente relativi all'utilizzo dell'algoritmo. In figura 24 è possibile osservare l'andamento dei consumi di corrente; si può osservare dettagliatamente il consumo all'attivazione del campionamento e all'esecuzione dell'algoritmo. Il consumo medio durante l'esecuzione è: 1.78 mA; il consumo di

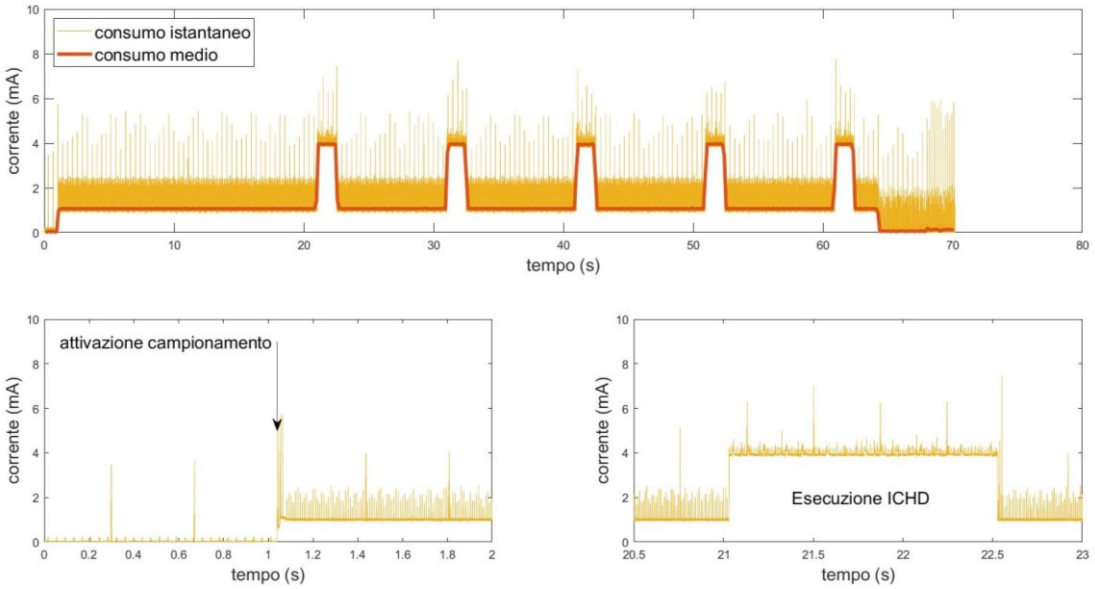


*Figura 24: Consumi di corrente. In alto, consumi durante campionamento ed esecuzione algoritmo. In basso a sinistra, dettaglio consumo del campionamento. In basso a destra, dettaglio dell'esecuzione dell'algoritmo ICHD.*

potenza corrisponde a 6.59 mW. Analizzando il caso in cui il consumo di corrente sia continuo e senza interruzioni e tenendo in considerazione la batteria da 250 mAh presente sulla scheda, l'algoritmo può essere eseguito per 5 giorni. Attraverso i consumi è stato possibile determinare il tempo effettivo di esecuzione dell'algoritmo che corrisponde a 1.5 secondi. Inoltre, è possibile notare che precedentemente all'attivazione del campionamento dei dati sono presenti degli spike; essi consistono

nell'invio di dati per mantenere la connessione attiva. Essi avvengono ogni 371,25 ms per limitare il consumo quando i dati non vengono inviati. Questo arco temporale è dovuto ad un connection interval stabilito dal central e da un slave latency pari a 32.
La memoria occupata dal firmware in memoria Flash è pari alla somma delle voci RODATA, TEXT e ALTRO FLASH, ed è pari a 303 KiB, che corrisponde al 29.56% del totale della Flash. Inoltre, l'ottimizzazione del codice ha comportato un'occupazione della memoria RAM corrispondente alla somma delle voci BSS, DATA e ALTRO RAM, pari a 199 kiB, corrispondenti al 77.73% della RAM. Questi dati sono visibili in Tabella 2.

| | BSS | DATA | RODATA | TEXT | ALTRO FLASH | ALTRO RAM |
|---|---|---|---|---|---|---|
| **originale** | **226700** | **844** | **158416** | **151336** | **368** | **20** |
| **ottimizzato** | **202128** | **844** | **158416** | **151208** | **368** | **20** |
| **risparmio** | **24572** | **0** | **0** | **128** | **0** | **0** |

*Tabella 2: Confronto relativo alla memoria occupata dalla versione originale generata dal Matlab Coder e dalla versione ottimizzata.*

Si può notare un risparmio in BSS pari a circa 24 KiB di memoria; il BSS interessa la memoria RAM. Nella memoria Flash il risparmio è molto minore rispetto alla RAM, si tratta soltanto di 128 Byte. Il risparmio ottenuto in RAM corrisponde al 10.84% e in Flash il 0.08%. Confrontando la memoria occupata dal firmware prima che venisse implementato l'algoritmo, è possibile stimare la memoria necessaria a quest'ultimo. Essa è pari a 174 KiB per la RAM e 150 KiB per la memoria Flash.

## 6. Conclusioni

L'implementazione dell'algoritmo è avvenuta con successo; perciò, si è dimostrato di poter ottenere tali risultati anche su sistemi con minor potenza di calcolo. Le problematiche presenti precedentemente all'implementazione erano anzitutto la gestione dell'interrupt e l'invio continuo dei dati, che sono state superate; esse rappresentavano una importante complicazione da risolvere, altrimenti non sarebbe stato possibile aggiungere questa nuova funzione al firmware.
Inoltre, nella progettazione si è cercato di ridurre il più possibile il consumo di corrente della scheda in modo tale da permettere l'utilizzo dell'algoritmo per tempi lunghi senza compromettere troppo la carica della batteria.
Un'altra specifica che si è cercato di ottimizzare è l'uso della memoria; facendo ciò, è possibile ottenere risparmi di memoria eventualmente utili nello sviluppo di nuove soluzioni per la scheda. Inoltre, il firmware permette di essere eseguito in varie modalità: spetterà all'utente la scelta dei dati che intende ricevere.
Il principale obbiettivo del lavoro, cioè l'implementazione di un algoritmo che permetta di ottenere dati utili al riconoscimento di attività motorie, è stato conseguito con successo. In conclusione, il riconoscimento di quest'ultime dovrà essere eseguito con ulteriori algoritmi, ma ora si ha la disponibilità dei dati utili a quello scopo.

# Riferimenti


[1] G. Biagetti, P. Crippa, A. Curzi e S. Orcioni, «Analysis of the EMG Signal During Cyclic Movements Using Multicomponent AM-FM Decomposition,» *IEEE JOURNAL OF BIOMEDICAL AND HEALTH INFORMATICS,* vol. 19, n. 5, pp. 1672-1681, 2015.

[2] J. G. Proakis e M. Salehi, Communication System Engineering, Upper Saddle River, New Jersey: Prentice Hall, 2002.

[3] G. Biagetti, P. Crippa, D. Bocchini, M. Alessandrini, L. Falaschetti e C. Turchetti, «Embedded AM-FM Signal Decomposition Algorithm for Continuous Human Activity Monitoring,» in *26th International Conference on Knowledge-Based and Intelligent Information & Engineering System (KES 2022)*, Verona, Italy, 2022.

[4] The MathWorks, Inc., «MathWorks,» [Online]. Available: https://it.mathworks.com/help/matlab/ref/unwrap.html. [Consultato il giorno 20 09 2022].

[5] The MathWorks, Inc., «MathWorks,» [Online]. Available: https://it.mathworks.com/products/matlab-coder.html. [Consultato il giorno 20 09 2022].

[6] The MathWorks, Inc., «MathWorks,» [Online]. Available: https://it.mathworks.com/products/embedded-coder.html. [Consultato il giorno 20 09 2022].

[7] The MathWorks, inc., «MathWorks,» [Online]. Available: https://it.mathworks.com/matlabcentral/fileexchange/43095-embedded-coder-support-package-for-arm-cortex-m-processors. [Consultato il giorno 20 09 2022].

[8] The MathWorks, Inc., «MathWorks,» [Online]. Available: https://it.mathworks.com/matlabcentral/fileexchange/43093-embedded-coder-support-package-for-stmicroelectronics-stm32-processors. [Consultato il giorno 20 09 2022].

[9] Arm Limited, «arm Developer,» [Online]. Available: https://developer.arm.com/tools-and-software/embedded/cmsis. [Consultato il giorno 20 09 2022].

[10] Arm Limited, «arm Developer,» [Online]. Available: https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain. [Consultato il giorno 20 09 2022].

[11] Nordic Semiconductor, «infocenter nordicsemi,» [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Flib_softdevice_handler.html&cp=4_0_3_3_40_2_0&anchor=lib_sdh_register_observer. [Consultato il giorno 20 09 2022].

[12] Nordic Semiconductor, «infocenter nordicsemi,» [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_gsg_ses%2Fdita_common%2Fglossary%2Fglossary.html&anchor=softdevice. [Consultato il giorno 20 09 2022].

[13] MartinBL, «Devzone nordicsemi,» 26 08 2015. [Online]. Available: https://devzone.nordicsemi.com/guides/short-range-guides/b/bluetooth-low-energy/posts/ble-services-a-beginners-tutorial. [Consultato il giorno 26 09 2022].

[14] Nordic Semiconductor, «infocenter nordicsemi,» [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsds_s140%2FSDS%2Fs1xx%2Fprocessor_avail_interrupt_latency%2Fexception_mgmt_sd.html. [Consultato il giorno 20 09 2022].