UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Elettronica

Sviluppo di firmware per lo streaming via USB di dati acquisiti da sensori inerziali e bioelettrici

Firmware development for USB streaming of data acquired by inertial and bioelectric sensors

Relatore:
Prof. **Giorgio Biagetti**

Tesi di Laurea di:
Nicholas Lilla

A.A. **2019 / 2020**

SOMMARIO

INTRODUZIONE. 2

1 - IL PROTOCOLLO USB. 3

1.1 - GENERALITA'...3

1.2 - LA STORIA...3

1.3 - I COMPONENTI DEL BUS...5

1.4 - SPECIFICHE ELETTRICHE..6

1.5 - IL FORMATO DATI...9

1.6 - LA PERIFERICA USB...19

1.7 - SPECIFICHE POWER..23

1.8 - SPECIFICHE MECCANICHE...25

2 - IL PROGETTO: FIRMWARE DEVICE-SIDE. 27

2.1 - IL MICROCONTROLLORE..27

2.2 - IL CODICE...30

3 - IL PROGETTO: FIRMWARE HOST-SIDE. 51

3.1 - LA LIBRERIA LIBUSB...51

3.2 - LA STRATEGIA..51

3.3 - IL CODICE...52

4 - IL FUNZIONAMENTO. 59

5 - CONCLUSIONI E SVILUPPI FUTURI. .63

BIBLIOGRAFIA. 64

SITOGRAFIA. 64

Introduzione

L'obbiettivo principale dell'esperienza è quello di sviluppare un firmware per lo streaming via USB di dati acquisiti da sensori inerziali e bioelettrici.
Il punto di partenza è un dispositivo già realizzato per un progetto strategico di ateneo. Questo è munito di un sensore per l'acquisizione di segnali bioelettrici come elettromiografia (EMG) ed elettrocardiogramma (ECG), accoppiato ad un sensore inerziale (accelerometro e giroscopio) per la rilevazione dell'attività motoria.
Attualmente, i sensori indossabili stanno diventando comuni in molti campi di applicazione come la sanità, lo sport, il fitness, l'intrattenimento o i sistemi di guida autonoma. In particolare, i segnali corporei menzionati in precedenza si sono dimostrati in grado di fornire informazioni accurate e affidabili sulle attività delle persone e sui loro comportamenti. Pertanto questi possono essere utilizzati in modo efficiente per il monitoraggio giornaliero di pazienti affetti da demenza o da disturbi neuro degenerativi caratterizzati cioè dal progressivo decadimento delle facoltà cognitive e motorie.
Il protocollo attualmente in essere per la comunicazione dati è il BLE 4.0, questo è estremamente discreto in quanto wireless e consente un facile interfacciamento con i dispositivi di elettronica di consumo oggigiorno frequentemente utilizzati come smartphone e tablet. Il limite principale di tale protocollo è il bit rate che si aggira intorno ad 1Mbit/s.
Proprio per quest'ultima considerazione si è pensato di aggiornare il dispositivo implementando un protocollo di comunicazione USB (Universal Serial Bus) in grado di garantire il trasferimento dei dati a relativamente alta velocità (12Mbit/s).
Nel seguito si farà un' introduzione generale al protocollo USB e poi si descriverà in maniera dettagliata quanto realizzato nell'ottica di ottenere quanto sopra descritto.

1 - Il protocollo USB

1.1 - Generalità

L'Universal Serial Bus (USB) è un'interfaccia seriale progettata per standardizzare il collegamento delle periferiche ai personal computer. La comunicazione tramite USB è attualmente molto diffusa ed è diventata un'interfaccia “obbligatoria” per tutti i sistemi embedded che devono essere interfacciati ad un calcolatore elettronico.

1.2 - La storia

I primi PC introdotti sul mercato erano caratterizzati da molte interfacce e connettori dedicati a periferiche esterne. Il numero di connettori e le difficoltà per l'utilizzatore erano molteplici. Proprio per sopperire a queste problematiche, nel 1996, l'USB Implementers Forum (USB-IF) presentò la prima versione del protocollo USB. Lo slogan fu “one plug to rule them all” che in italiano significa “un connettore per dominarli tutti”. L'intento era quello di realizzare un protocollo che potesse supportare molteplici applicazioni utilizzando un solo bus. In altri termini l'obbiettivo era quello di introdurre un protocollo universale e semplice per l'utilizzatore.
Il nome stesso del protocollo USB ovvero Universal Serial Bus, mette in evidenza uno degli scopi principali del nuovo standard, cioè poter essere universale e indipendente dalla periferica: mouse, tastiera, memoria di massa, webcam o altro.

Specifiche USB 1.0

La storia dell'USB inizia con la versione 1.0 rilasciata dall'USB IF nel 1996. Le specifiche vennero pensate per poter supportare periferiche lente come tastiera e mouse, ma anche periferiche con esigenze di trasferimento dati maggiori come gli allora diffusi floppy disk o le stampanti.
Il protocollo definisce due modalità:

- Low Speed, con bit rate fino a 1.5Mbits/s;
- Full Speed, con bit rate fino a 12Mbits/s

Nonostante l'introduzione all'inizio del 1996, l'effettiva diffusione dell'USB richiese circa due anni con la versione 1.1 del 1998.

Specifiche USB 2.0

Il numero di periferiche che facevano uso della porta USB crebbe, ed in particolare l'esigenza della velocità di trasferimento dati andava oltre le specifiche USB 1.x. Nell'aprile del 2000, l'USB IF introdusse le nuove specifiche USB 2.0. Venne introdotta in questa occasione la nuova modalità:

- High Speed, con bit rate fino a 480Mbits/s

Secondo le specifiche USB2.0 e precedenti, l'arbitraggio del bus spetta esclusivamente all'Host che svolge il ruolo del master. Non è dunque consentita la connessione punto-punto di due device in assenza di un Host. Proprio per sopperire a questa limitazione, come supplemento della specifica

2.0, è stato introdotto il protocollo USB OTG (On The Go). Quest'ultimo permette ad un device di comportarsi come un semplice Host al quale possono essere collegate ulteriori periferiche.
In altri termini il protocollo USB OTG aggiunge al device funzionalità normalmente svolte dal master per consentire la connessione punto-punto di due periferiche USB in assenza di un Host.

Specifiche USB 3.0

Nel novembre del 2008 l'USB IF ha introdotto le nuove specifiche USB 3.0 con lo scopo di aumentare il bit rate supportato fino 5Gbits/s. In questa occasione viene definita la modalità:

- Super Speed, con bit rate fino a 5Gbits/s

Le nuove specifiche hanno introdotto anche ulteriori cambiamenti al fine di supportare un'esigenza collaterale nata con il tempo, ovvero la ricarica dello smartphone. In particolare fino alle specifiche USB 2.0 un dispositivo poteva richiedere fino ad un massimo di 500mA, mentre le periferiche USB 3.0 possono richiedere fino a 900mA.

Specifiche USB 3.1

Sebbene il protocollo USB 3.0 abbia esteso il bit rate a 5Gbits/s, nel 2013 l'USB IF ha introdotto la nuova modalità:

- Super Speed+, con bit rate fino a 10Gbits/s

Le nuove specifiche hanno mantenuto la retro compatibilità con USB 2.0 e USB 3.0.

Specifiche USB 4.0

L'ultima versione nota dello standard USB è la 4.0. Questa è stata annunciata ufficialmente a marzo del 2019 e debutterà nel 2021. La velocità di trasferimento dei dati si attesta intorno ai 40 Gbits/s.

1.3 - I componenti del bus

La comunicazione sul bus USB è di tipo master/slave e l'architettura è a stella a più livelli. Un unico master è responsabile della configurazione e della comunicazione attraverso il bus (escludendo la modalità OTG).
Le specifiche USB definiscono i seguenti elementi:

- Host;
- Device (periferica);
- Hub.

L'Host è il responsabile della comunicazione sul bus ed è tipicamente un calcolatore elettronico. Esso rappresenta il master, ovvero colui che inizia ogni comunicazione (escludendo il protocollo OTG) e che controlla tutti i dispositivi collegati al bus USB. In totale possono essere connessi all'Host 127 dispositivi tra Hub e device. Normalmente l'Host ha un numero limitato di porte USB dove è possibile fisicamente collegare dei device esterni. Per estendere il numero di porte si utilizzano gli Hub, i quali costituiscono l'elemento centrale della connessione a stella. Ogni Hub permette di moltiplicare le connessioni USB e di adeguare, se necessario, la velocità di comunicazione con il device. Lo stesso Host integra al suo interno un Hub che prende il nome di root Hub . I device sono i punti terminali della rete a stella, questi devono rispondere a tutti i comandi fondamentali del protocollo per consentire all'Host di riconoscere la periferica durante la fase di configurazione.

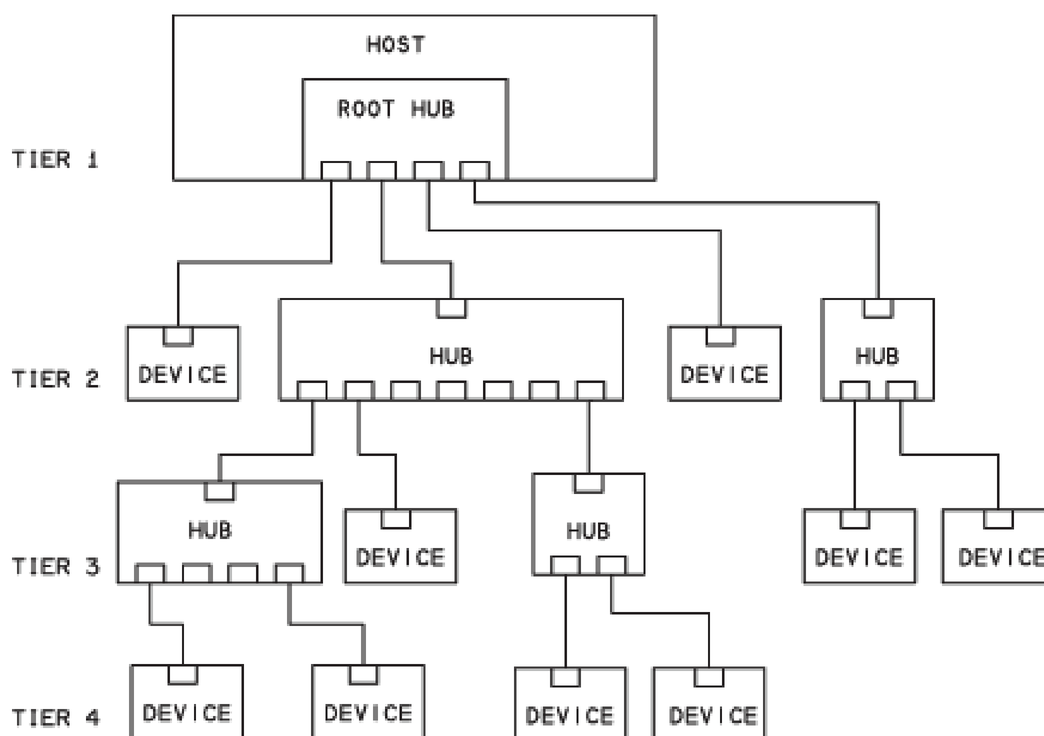

Illustrazione 1.1: topologia del bus USB

1.4 - Specifiche elettriche

Il protocollo USB definisce per la trasmissione dati un bus seriale con linee differenziali.
In base alla specifica del protocollo si ha una trasmissione seriale Half Duplex (USB 1.x e 2.0) o Dual Simplex (USB 3.x) asincrona, ovvero senza trasmissione diretta del segnale di clock.
Oltre alla tipologia di trasmissione, anche il bit rate, varia in base alla modalità utilizzata.
La trasmissione Dual Simplex non è da confondere con quella Full Duplex. In una trasmissione Full Duplex, infatti, vi sono due linee simultanee unidirezionali vincolate per progettazione hardware ad assumere direzioni opposte. Tale vincolo è rimosso in una trasmissione Dual Simplex in cui si hanno due linee simultanee unidirezionali che possono però assumere anche la medesima direzione.

Specifiche elettriche Low Speed e Full Speed

Gran parte dei microcontrollori, avendo risorse limitate supportano prevalentemente lo standard Low Speed o Full Speed. Il bit rate e la tolleranza richiesta alla frequenza di clock variano in base allo standard supportato. I dettagli sono riportati nella tabella sottostante:

Standard	Bit rate	Clock Accuracy
Low Speed	1.5Mbit/s	1,50%
Full Speed	12Mbit/s	0.25%

Tabella 1.1: specifiche del clock e del bit rate per le modalità *Low Speed* e *Full Speed*

Utilizzando la modalità Low Speed, essendo richiesta un'accuratezza di clock non troppo elevata, è possibile utilizzare anche un oscillatore RC, spesso integrato all'interno dei microcontrollori.
I bit trasmessi sul bus USB sono di tipo differenziale, in particolare il bus è composto da due linee elettriche nominate D+ e D-, oltre a massa e V_{BUS}.
Se (D-) - (D+) > 200 mV e (D-) > 2V si dice che il bus USB è nello stato di “0 differenziale” mentre se (D+) - (D-) > 200 mV e (D+) > 2V nello stato di “1 differenziale”.
I bit ora descritti non rappresentano in realtà il bit reale che si vuole trasmettere. Per giungere a questo bisogna introdurre un livello di astrazione aggiuntivo, ovvero il simbolo J e il simbolo K.
La corrispondenza tra il bit sulla linea del bus ed i simboli J e K dipendono dallo standard, come riportato in tabella:

Stato	Low Speed	Full Speed
0 differenziale	J	K
1 differenziale	K	J

Tabella 1.2: equivalenza tra bit sul bus e simboli J e K

Avendo definito i simboli J e K e averli invertiti dal punto di vista del significato dei bit sul bus, permette di non dover pensare al valore elettrico del bus stesso quando si ha a che fare con le rispettive modalità Low Speed o Full Speed.

La codifica dei dati

Introdotti i simboli J e K si può giungere all'effettivo valore dei bit trasmessi sul bus tenendo conto della codifica NRZI (Non Return to Zero Inverted). In particolare, nella codifica NRZI usata dallo standard USB2.0, il livello logico basso corrisponde ad una variazione di tensione sul bus, il livello logico alto a tensione costante sul bus. Per cui, ragionando in simboli, quando si ha una variazione di simbolo si ha uno “0” mentre se il simbolo rimane lo stesso si ha un “1”. In tal caso “0” e “1” sono i bit trasferiti sul bus, convenzionalmente dall'LSB (Less Significant Bit) all'MSB (Most Significant Bit).
Oltre alla codifica NRZI il protocollo USB fa uso del Bit Stuffing, ovvero dell'inversione automatica del bit qualora ci sia un numero eccessivo di “1”. Nel caso delle specifiche USB viene introdotto un bit di Stuffing (“0”) ogni 6 bit pari ad “1”. Questo permette di inserire un numero di variazioni di bit sufficiente a garantire che ogni sistema possa rimanere sincronizzato in maniera opportuna. Infatti, il bus USB, non trasmettendo direttamente il clock, richiede che ogni dispositivo rimanga sincronizzato sul bit rate richiesto per la comunicazione.
Per facilitare la sincronizzazione dei dispositivi, ogni pacchetto dati inizia anche con la trasmissione del campo SYNC, che nel caso delle modalità Low Speed e Full Speed è rappresentato dai simboli KJKJKJKK.
La struttura dei pacchetti verrà descritta più nel dettaglio in seguito.
A scopo riassuntivo sono riportati nella tabella sottostante i vari segnali che si possono trovare sul bus USB.

Segnale	D+	D-	Low Speed	Full Speed
0 differenziale	Alto	Basso		
1 differenziale	Basso	Alto		
Stato di Idle			D- connesso a +3.3V con resistore da 1.5KΩ ±5%	D+ connesso a +3.3V con resistore da 1.5KΩ ±5%
Simbolo J			0 differenziale	1 differenziale
Simbolo K			1 differenziale	0 differenziale
SE0 (Single Ended 0)	Basso	Basso		
SE1 (Single Ended 1)	Alto	Alto		
SOP (Start Of Packet)			Da Idle a K	Da Idle a K
EOP (End Of Packet)			SE0 per un bit seguito da J	SE0 per un bit seguito da J
SYNC			KJKJKJKK	KJKJKJKK
Reset			SE0 per almeno 10ms	SE0 per almeno 10ms

Tabella 1.3: riassunto dei diversi possibili segnali sul bus USB

Interfaccia sul bus USB per la modalità Low Speed e Full Speed

Le specifiche USB descrivono in maniera precisa come devono essere realizzati i dispositivi d'interfaccia sul bus USB, sia dal lato Host (Host-side) che dal lato del device (device-side).

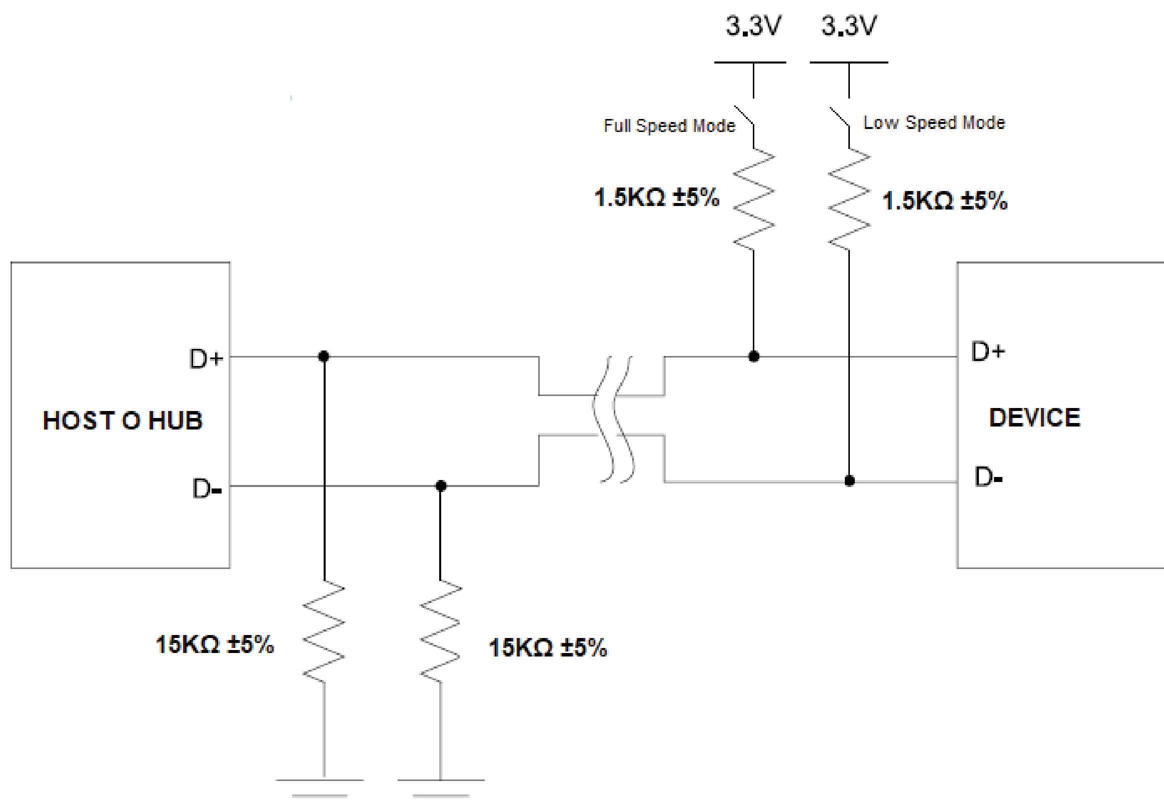

Illustrazione 1.2: interfaccia sul bus USB per la modalità Low Speed e Full Speed

I resistori di pull-up lato device devono essere abilitati in base allo standard che si decide di adottare. In particolare, un device che opererà sempre in una modalità potrebbe avere un solo resistore collegato in maniera fissa a 3.3V.

Specifiche elettriche per la modalità High Speed

Lo standard High Speed supporta un bit rate di 480Mbit/s. Un dispositivo USB High Speed, deve presentarsi alla connessione come dispositivo Full Speed, ovvero con un resistore di pull-up sulla linea D+. Dopo la sequenza di Handshake che permette di impostare la modalità High Speed, il resistore di pull-up deve essere eliminato dalla linea e la trasmissione può avvenire al massimo bit rate. Il bus High speed prevede due ulteriori stati definiti chirpJ e chirpK. Questi sono utilizzati durante la fase di configurazione per distinguere un device High Speed da uno Full Speed.
Dal momento che i microcontrollori, per natura delle risorse disponibili, non supportano normalmente le specifiche High Speed, sebbene supportino lo standard USB 2.0, non si entrerà nel seguito nel dettaglio della modalità High Speed.

1.5 - Il formato dati

Il protocollo USB, al fine di supportare un'ampia tipologia di applicazioni, fornisce diverse modalità di trasmissione dati. La comunicazione tra Host e device si definisce transfer. Ognuna di queste si suddivide a sua volta in una o più transaction, le quali a loro volta si suddividono in packets.
La transaction è formata da un packet d'intestazione detto token (gettone) che definisce il tipo e il device destinatario della transaction stessa. Il token è seguito dal data packet che include eventuali dati inviati/ricevuti. In chiusura viene inviato l'handshake packet che contiene informazioni sul controllo d'errore nella transaction.
Ogni packet ha a sua volta una ben precisa struttura, questa verrà discussa nel dettaglio in seguito.

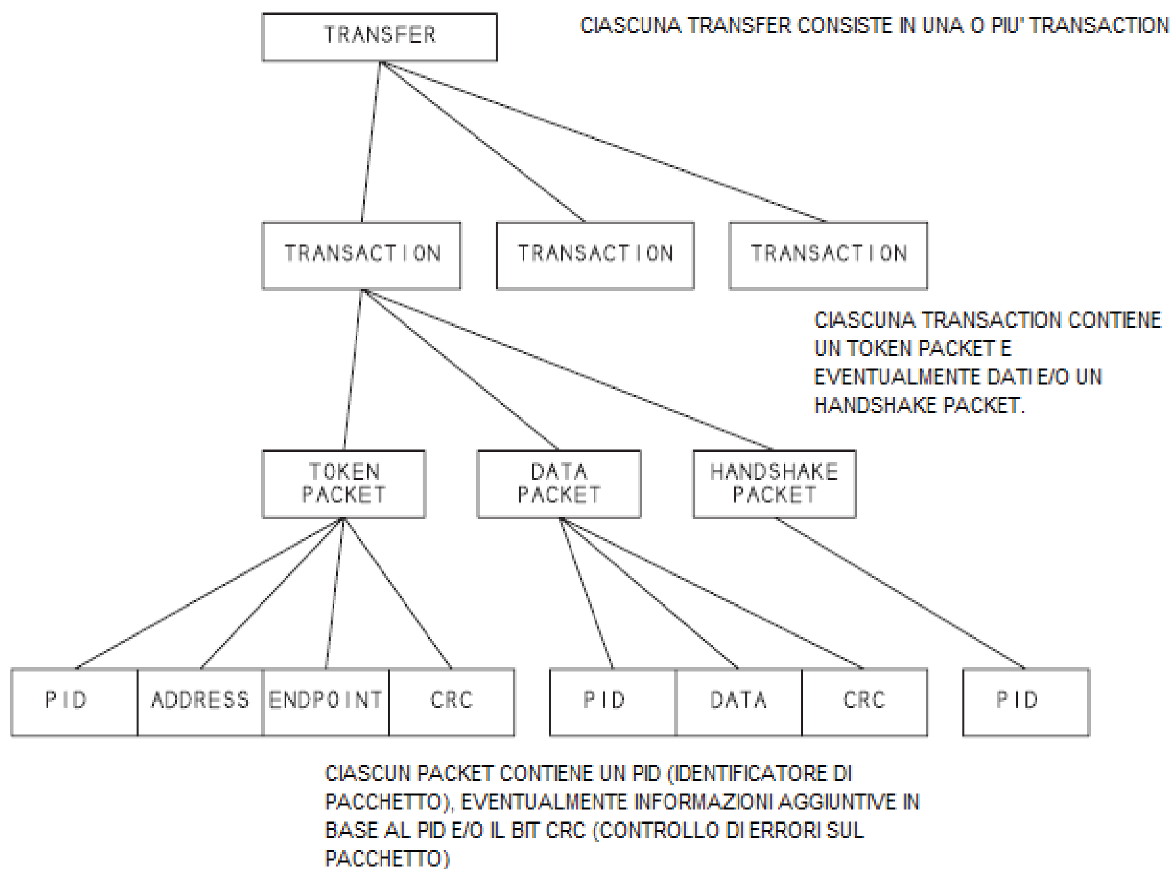

Illustrazione1.3: struttura di una transfer

Le Transfer

Nel protocollo USB esistono quattro tipi di transfer:

- Control;
- Interrupt;
- Bulk;
- Isochronous

Tutte sono caratterizzate dal non avere una reale formattazione dati, eccetto per la transfer control quando è utilizzata per inizializzare il device. Ciascun tipo di transfer ha proprie caratteristiche che la rende o meno adatta per applicazioni specifiche.

Control

La modalità control è utilizzata durante la fase di inizializzazione del device. In altri termini, una control transfer è un messaggio standard diretto da Host a device nella fase di configurazione del device stesso. Proprio per questo motivo, ogni periferica deve riconoscere tali control transfer.
Le control transfer si possono suddividere in tre fasi (stage):

1. La fase di setup in cui l'host invia la setup transaction che identifica la control transfer;
2. La fase di data transfer che comprende zero o più data transactions;
3. La fase di status costituita da una transaction di handshake e verifica degli errori.

Come descritto in precedenza ogni transaction avrà la sua struttura a pacchetti e sarà formata da un token packet, un data packet (opzionale) e un handshake packet.
La dimensione del data packet per una control transfer dipende dallo standard. Nella tabella sottostante sono riportate le massime dimensioni del data packet al variare dello standard.

Standard	Dimensione pacchetto dati (byte)
Low speed	8
Full speed	8, 16, 32, 64
High speed	64

Tabella 1.4: dimensione del data packet per una control transfer al variare dello standard

Sebbene la modalità control è utilizzata principalmente nella fase di inizializzazione del dispositivo (processo di enumerazione), nulla impedisce di utilizzare tale modalità anche per la comunicazione dati vera e propria.
La velocità massima raggiungibile con una control transfer, considerando anche tutti i byte aggiuntivi di controllo, è di 24Kbyte/s in modalità low speed, 832Kbyte/s in full speed e 15.8Mbyte/s in high speed.
La periferica deve rispondere ad una control transfer entro 500ms oppure inviare un messaggio opportuno per informare l'Host di ripetere la richiesta in un secondo momento.
La struttura di una control read transfer e di una control write transfer è illustata in figura qui di seguito. Il significato dei singoli pacchetti verrà analizzato nel prosieguo.

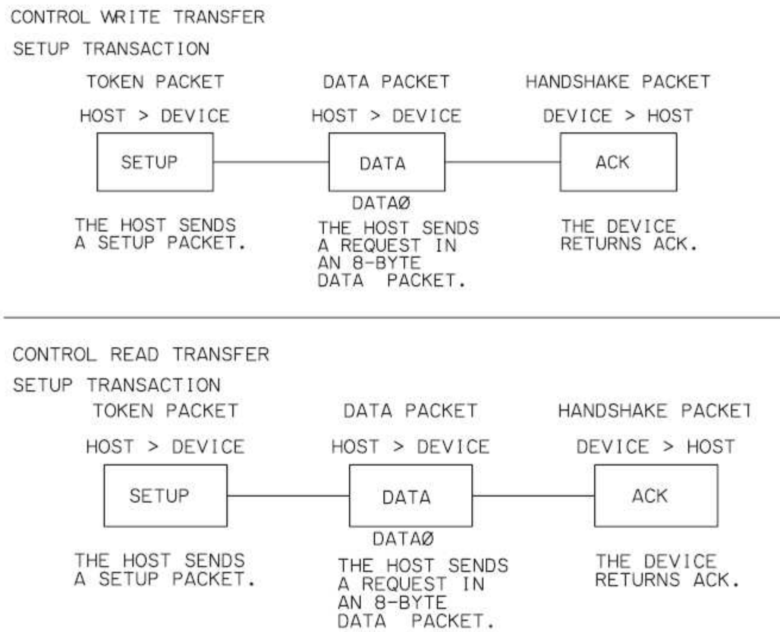

Illustrazione 1.4: struttura di una control transfer

Interrupt

L'interrupt transfer permette di trasferire dati con periodicità stabilita. Questa tipologia di trasferimento dati è utilizzata principalmente da dispositivi che devono fornire dati in maniera rapida, come per esempio mouse e tastiere. L'indicazione interrupt potrebbe essere fuorviante, in quanto, in realtà il device non genera nessun interrupt che permette di richiamare l'attenzione dell'Host. La comunicazione è sempre guidata dall'Host che interroga periodicamente il device (polling) al fine di determinare se sono presenti o meno dei dati. La periodicità è stabilità dal device e non dall'Host che si limita esclusivamente all'apertura della transfer. Quanto appena detto non è vero nel caso di periferiche USB 3.x le quali possono inviare una richiesta all'Host al fine di poter richiedere una lettura dei dati eventualmente disponibili. La struttura di un interrupt transfer è costituita da una o più data transaction con dimensione del data packet indicata nella tabella sottostante.

Standard	Dimensione pacchetto dati (byte)
Low speed	1 … 8
Full speed	1 … 64
High speed	1 … 2024

Tabella 1.5: dimensione del data packet per un'interrupt transfer al variare dello standard

Bulk

La modalità bulk, sebbene non supporti dei tempi prestabiliti per lo scambio delle informazioni, permette di inviare grosse quantità di dati. I dati in una bulk transfer vengono trasferiti come in un'interrupt transfer eccetto per il fatto che non si ha una trasmissione dati con periodicità prestabilita. I trasferimenti di tipo bulk occupano tutta la banda disponibile al termine degli altri trasferimenti e viene rallentata in caso di congestione del bus. La bulk transfer, che non è supportata dalla modalità Low Speed, si compone di una o più data transaction che hanno tutte la stessa direzione da host a device o viceversa. In particolare fanno uso della modalità di trasmissione bulk i dispositivi di memoria di massa e anche quelli appartenenti alla classe CDC (Communication

Device Class). Un'analisi più dettagliata delle diverse tipologie di classi di dispositivi verrà condotta in seguito.
La dimensione massima di un data packet per il trasferimento bulk è riportata nella tabella sottostante:

Standard	Dimensione pacchetto dati (byte)
Low speed	Non ammessa
Full speed	8, 16, 32, 64
High speed	512

Tabella 1.6: dimensione del data packet per una bulk transfer al variare dello standard

La velocità massima di trasferimento per una bulk transfer è 1.2Mbyte/s in modalità full speed e 53Mbyte/s in modalità high speed.
Una bulk transfer ha una struttura perfettamente identica ad un'interrupt transfer in accordo con le specifiche USB 2.0, ciò che le differenzia è il diverso scheduling host-side.
Nella figura qui di seguito è mostrata la struttura di una IN e di una OUT bulk/interrupt transaction.
Il significato dei singoli pacchetti (ACK,NAK, STALL ecc.) verrà chiarito a breve.

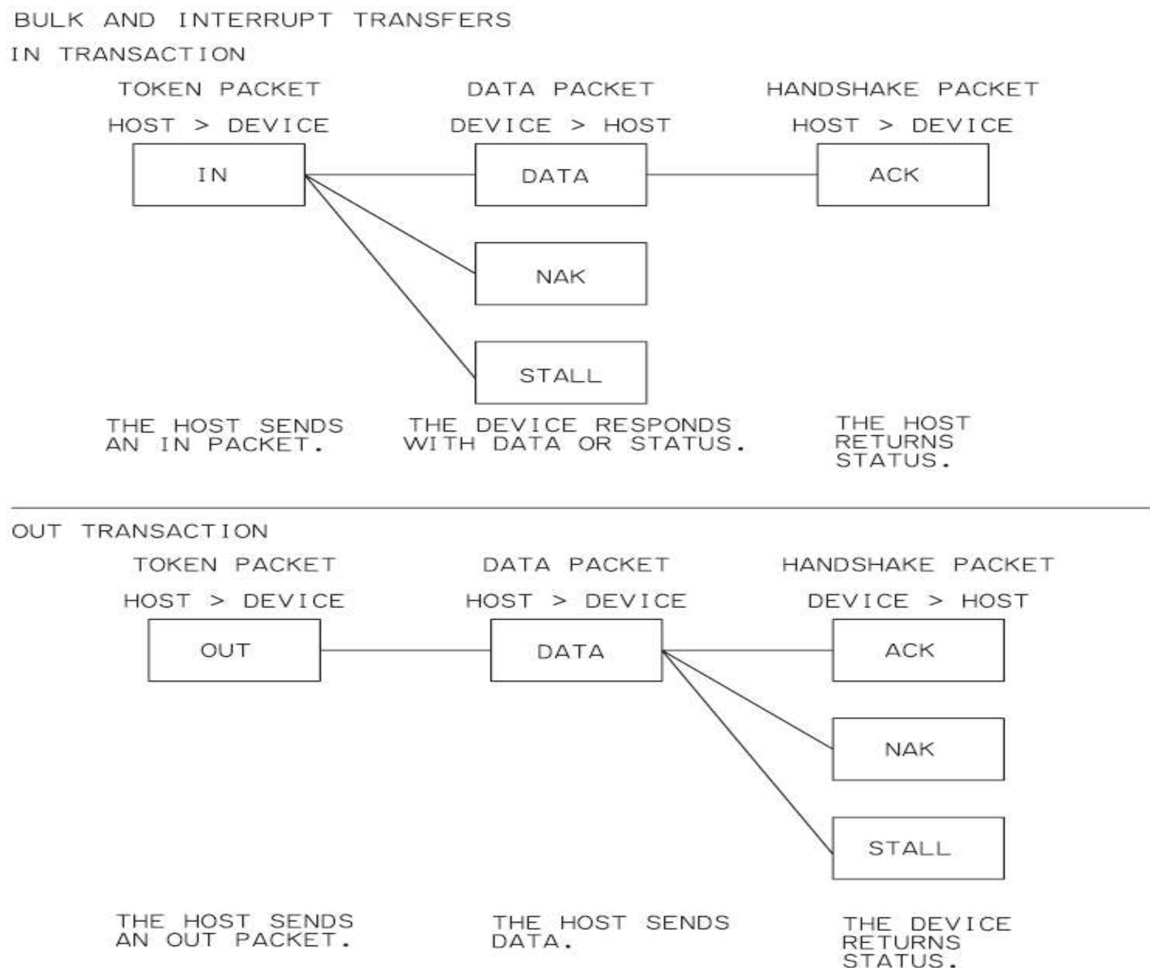

Illustrazione 1.5: struttura di una bulk/interrupt (IN e OUT) transaction

Isochronous

La modalità Isochronous ha un tempo di trasmissione garantito e può essere utilizzata per inviare ad esempio stream audio/video. Questa è pensata per raggiungere un'elevata trasmissione dati, tuttavia, non sempre risulta essere una modalità conveniente. Questo perché non sono presenti segnali di Acknowledge ovvero di segnalazione di correttezza dei dati. Proprio per questo motivo, è sconsigliato utilizzare tale modalità ad esempio per la trasmissione di file, a meno di non implementare sistemi di correzione al livello del device. Tuttavia se si hanno tali esigenze, normalmente, è bene usare un'altra modalità.
Nella tabella sottostante sono riportati i dettagli delle dimensioni dei pacchetti dati che possono essere inviati in questa modalità.

Standard	Dimensione pacchetto dati (byte)
Low speed	Non ammessa
Full speed	0 … 1024
High speed	0 … 1023

Tabella 1.7: dimensione del data packet per una isochronous transfer al variare dello standard

Le transaction in una isochronous transfer includono solo token e data packet (manca l'handshake packet per il controllo d'errore). La struttura nel dettaglio è mostrata qui di seguito:

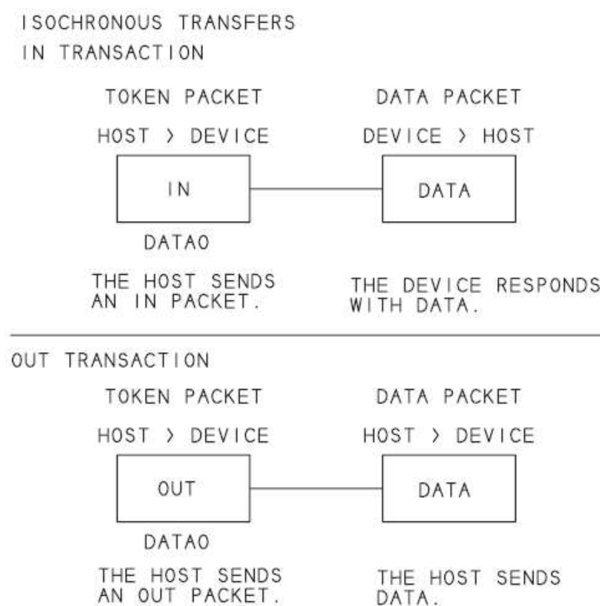

Illustrazione 1.6: struttura isochronous transfer

Le Transaction

Come si è indicato in precedenza le transaction sono suddivise in packet, ognuna inizia con un token packet inviato dall'Host alla periferica. Il token packet è seguito da zero o più data packet. La transaction è chiusa dall'handshake packet di controllo dell'errore sulla transaction. Prima di iniziare una nuova transaction con un device, l'Host deve precedentemente concludere quella in corso.

I Packet

Preliminarmente analizziamo la struttura di un pacchetto generico, dopodiché descriveremo nel dettaglio i diversi packet con le possibili varianti.
La struttura generale di un pacchetto è la seguente:

- **SYNC**
  Ogni pacchetto inizia con 8 bit di sincronismo per comunicazioni low/full speed e 32 bit per high speed. Questi bit sono utilizzati per sincronizzare il clock del device con quello dell'Host.

- **PID**
  Il frame di sincronismo è seguito dal Packet Identifier (PID) codificato a 4 + 4 bit.
  Il campo PID è di un byte ma solo 4 bit vengono utilizzati e i restanti 4 bit rappresentano il complemento ad 1 (bit reverse) del valore reale. Questo permette di avere un controllo sul valore del PID stesso.
  Il byte di PID ha quindi la seguente struttura:

PID_0	PID_1	PID_2	PID_3	$\overline{PID_0}$	$\overline{PID_1}$	$\overline{PID_2}$	$\overline{PID_3}$

  Con PID_i con i = 0, … , 3 si intende l'i-esimo bit del byte di PID.
  La sopralineatura sta ad indicare l'operazione logica di inversione (NOT).
  Come si intende dal nome, il PID, è usato per identificare il tipo di pacchetto che si sta inviando. E' quindi fondamentale la corretta ricezione del PID. Se infatti il device interpreta in maniera errata il byte di PID proveniente dall'Host, la sua risposta non può altro che essere non idonea alla richiesta dell'Host stesso. Proprio per evitare che ciò accada, il protocollo USB prevede la ripetizione in bit reverse dei 4 bit di PID in modo da comporre nel complesso un byte.
  A seconda della tipologia di pacchetto (Token packet, Data packet e Handshake packet), esistono diversi possibili valori del PID, ciascuno dei quali identifica una particolare tipologia di pacchetto.
  La tabella sottostante mostra i diversi possibili valori:

Tipologia di pacchetto	Valore del PID	Packet ID
Token	0001	OUT token
	1001	IN token
	0101	SOF token
	1101	SETUP token
Data	0011	DATA0
	1011	DATA1
	0111	DATA 2
	1111	MDATA
Handshake	0010	ACK Handshake
	1010	NAK Handshake
	1110	STALL Handshake
	0110	NYET (No Response Yet)
Special	1100	PREamble (da Host)
	1100	ERR (da Hub)
	1000	Split
	0100	Ping

Tabella 1.8: PacketID per diversi tipi di pacchetto

- **ADD**
Il PID è seguito dal campo indirizzo (7 bit) utilizzato per specificare quale device è il destinatario del pacchetto. Dato che l'indirizzo è a 7 bit, è possibile connettere un totale di 127 device. L'indirizzo 0 non è valido, in quanto un qualsiasi device al quale non è ancora stato assegnato un indirizzo deve rispondere ai pacchetti inviati all'indirizzo 0.

- **ENDP**
Il campo indirizzo è seguito dal campo endpoint (4 bit). Questo è utilizzato per specificare l'endpoint destinatario del pacchetto che si sta inviando. Essendo il campo endpoint a 4 bit, è possibile avere un totale di 16 diversi endpoint.

- **CRC**

  Dopo i 4 bit utilizzati per identificare l'endpoint destinatario del pacchetto, il protocollo USB prevede la presenza del campo CRC (Cyclic Redundancy Checks). Questo è costituito da 5 bit per i token packet e 16 bit per i data packet. In particolare occorre distinguere i bit di CRC dall'handshake packet. Infatti, mentre quest'ultimo è un controllo sull'intera transaction, il CRC rappresenta un controllo sul pacchetto.

- **EOP**
Il pacchetto termina con una speciale sequenza di bit. Questa va sotto al nome di sequenza EOP ed è composta dal livello SE0 (Single Ended 0), ovvero D+ e D- al livello basso, seguito da un simbolo J, ovvero D+ alto.

Occorre ora analizzare più nel dettaglio i diversi tipi di pacchetto. Esistono, come già abbiamo visto, quattro diversi tipi di pacchetto:

- Token Packet;
- Data Packet;
- Handshake Packet;
- Special Packet

Token packet

Un token packet viene sempre inviato dall'Host al device all'inizio di una transaction per identificare il dispositivo e l'endpoint a cui fare riferimento.
Esistono quattro diversi tipi di token packet:

- **SOF**: viene inviato all'inizio di ogni frame. Il campo dati del SOF (Start Of Frame) packet contiene il numero di frame;

SYNC	PID (0101)	FRAME NUMBER	CRC

- **SETUP**: è il token che identifica la Setup transaction in una control message transfer;

SYNC	PID (1101)	DEVICE ADDRESS	CRC

- **IN**: definisce la direzione della transaction dati (IN è riferito all'Host);

SYNC	PID (1001)	DEVICE ADDRESS	CRC

- **OUT**: definisce la direzione della transaction dati (OUT è riferito all'Host);

SYNC	PID (0001)	DEVICE ADDRESS	CRC

Data packet

Un Data packet contiene i dati trasferiti tra Host e device. Il token packet inviato all'inizio della transaction stabilisce se si tratta di dati trasferiti verso l'Host o verso il device. Un pacchetto dati contiene i seguenti campi:

- PID: packetID;
- Data: byte dati in numero dipendente dal tipo di trasferimento;
- CRC: Cyclic Redundancy Check per l'identificazione di eventuali errori;

Esistono quattro diversi tipi di data packet:

- **DATA0**

SYNC	PID (0011)	DATA	CRC

- **DATA1**

SYNC	PID (1011)	DATA	CRC

- **DATA2**

SYNC	PID (0111)	DATA	CRC

- **MDATA**

SYNC	PID (1111)	DATA	CRC

Quando la transfer richiede più transaction dati, chi invia i dati alterna i PID DATA0 e DATA1 nell'intestazione del data packet. In questo modo il ricevitore è in grado di riconoscere la sequenza dei packet. I PID DATA2 e MDATA sono utilizzati con lo stesso scopo nei trasferimenti high speed.

Handshake packet

Questo packet chiude una transazione e viene inviato da chi riceve i dati per segnalare la ricezione corretta o eventuali condizioni d'errore. Esistono quattro diversi tipi di handshake packet:

- **ACK**: questo pacchetto è restituito in caso di corretta transazione;

SYNC	PID (0010)

- **NAK**: questo pacchetto non ha il significato di errore ma semplicemente di dati non disponibili. Se l'Host richiede dei dati ad un device che non ha dati disponibili oppure non è in grado di rispondere alla richiesta allora il device invia un NAK segnalando all'Host di riprovare in un secondo momento (il timeout è fissato a 500ms). Piuttosto che una condizione d'errore è più corretto dire che il NAK corrisponde al “riprova più tardi”.

SYNC	PID (1010)

- **STALL**: ha il significato di richiesta non riconosciuta. Questa risposta viene inviata nel caso in cui la richiesta faccia riferimento a funzioni non implementate sul device interrogato.

SYNC	PID (1110)

- **NYTE**: questo pacchetto è utilizzato solo in transazioni high speed per segnalare la condizione di device non disponibile per la comunicazione. A differenza del NAK che posticipa lo scambio dati in corso, quando un device chiude la transazione con NYET significa che accetta lo scambio dati richiesto dall'Host ma non è più disponibile per altre transactions.

SYNC	PID (0110)

Special packet

Gli special packet sono pacchetti di norma non inviati nella normale trasmissione dati tra Host e device. Esistono essenzialmente quattro tipi di special packet. Il più importante tra questi è il comando **PING** (PID 0100) . Come abbiamo già visto, l'handshake packet non serve solo per segnalare eventuali errori e richiedere di spedire nuovamente i dati, ma anche per permettere di avvertire l'Host se il Device non è pronto, inviando in particolare un NAK (Not Acknowledged). Dal momento che un eventuale segnalazione da parte del device relativa al non essere pronto avviene solo alla fine della trasmissione dei dati, qualora il dispositivo non dovesse essere pronto frequentemente, si potrebbe perdere molto tempo, ovvero banda sul bus.
Per questa ragione l'USB 2.0 High Speed ha introdotto anche la funzione PING che permette all'Host, qualora abbia ricevuto una richiesta di attesa da parte del dispositivo, di riprovare con il comando PING per verificare se il dispositivo è nuovamente pronto, piuttosto che inviare nuovamente tutti i dati e ricevere nuovamente una richiesta di attesa.

1.6 - La periferica USB

Endpoints

Ogni device possiede degli endpoint ovvero dei buffer di memoria per mezzo dei quali è possibile scambiare informazioni con l'Host. In altri termini, si definisce endpoint il punto finale della comunicazione tra Host e device. Gli endpoint possono essere di tre tipi: IN, OUT e Control.
IN e OUT si riferiscono alla direzione della comunicazione. In particolare tale verso viene definito con riferimento all'Host, per cui un endpoint IN rappresenta un buffer di memoria presente sul device che fornisce informazioni all'Host mentre un endpoint OUT rappresenta un buffer nel device che riceve dati in uscita dall'Host. Ogni device USB deve includere l'endpoint 0 che deve essere di tipo control. L'endpoint 0 è quello base di configurazione, è bidirezionale e viene utilizzato dall'Host al fine di configurare il device durante la fase di enumerazione (il processo di enumerazione di un device verrà descritto in seguito).
Gli endpoint hanno oltre che un verso di trasmissione anche una dimensione che identifica il buffer stesso.

La PIPE

Le connessioni logiche tra host e device si definiscono PIPE. Ogni PIPE è associata ad un endpoint. Ogni device USB all'avvio deve attivare la control PIPE associata all'endpoint 0. La PIPE associata all'endpoint 0 è bidirezionale di tipo message (control message) mentre quelle associate agli altri endpoint sono monodirezionali di tipo stream.

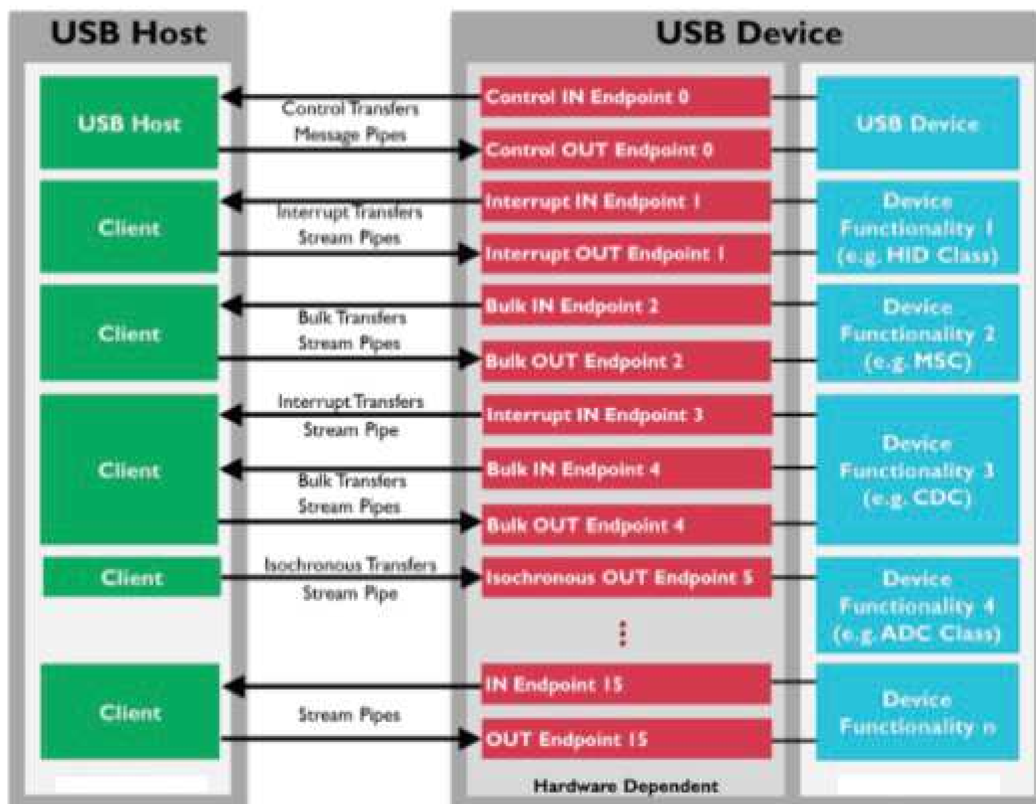

Illustrazione 1.4: connessioni host-device

Classe dei dispositivi

Il protocollo USB sebbene non definisca un formato dati predefinito, definisce delle classi di dispositivi, associando ad esse particolari caratteristiche da un punto di vista della modalità di trasmissione che deve essere utilizzata. Ogni classe definisce il numero di endpoint che deve attivare il device, i protocolli utilizzati dal device ed eventuali control message specifiche della classe.
Tra le classi di dispositivi più note si hanno:

- HID : Human Interface Device;
- MSC: Mass Storage Device;
- CDC: Communication Device Class;
- ADC: Audio Device Class,

Tutte le classi di dispositivi supportate dal protocollo USB sono riportate nella tabella sottostante:

ID	Descriptor	Device Class
0x00	Device	L'informazione della classe è fornita nell'Interface Descriptor
0x01	Interface	Classe Audio
0x02	Entrambi	CDC
0x03	Interface	HID
0x05	Interface	Physical
0x06	Interface	Image
0x07	Interface	Stampanti
0x08	Interface	Memoria di massa (MSC)
0x09	Device	HUB
0x0A	Interface	CDC
0x0B	Interface	Smart Card
0x0D	Interface	Content security
0x0E	Interface	Video
0x0F	Interface	Personal healthcare
0x10	Interface	Audio/Video
0x11	Device	Billboard Device Class
0x12	Interface	USB Type C Bridge Class
0xDC	Entrambi	Diagnostic Device
0xE0	Interface	Wireless Controller
0xEF	Entrambi	Miscellaneus
0xFE	Interface	Application specific
0xFF	Entrambi	Vendor specific

Tabella 1.9: classe dei dispositivi USB

In particolare la tabella riporta il codice identificativo (ID) della classe e anche il tipo di Descriptor a cui fa riferimento, ovvero Device, Interface o entrambi. Maggiori dettagli sui descriptor sono forniti nel paragrafo seguente.

Descrittori USB

Quando un device viene collegato ad un Host viene inizializzato al fine di assegnargli un indirizzo univoco sul bus. Oltre a questo, l'Host richiede informazioni al device per poterlo identificare e determinare quale driver debba essere caricato al fine di permettere il corretto funzionamento della periferica. Tutte queste informazioni sono contenute all'interno di vari Descriptor (descrittori) contenuti nel dispositivo. I descrittori sono delle tabelle (in linguaggio di programmazione delle strutture) che definiscono in dettaglio le caratteristiche del device e permettono all'Host di acquisire tutte le informazioni necessarie per stabilire la comunicazione corretta con il device stesso. Le informazioni sono distribuite in diversi descrittori ciascuno dei quali caratterizzato da un codice identificativo. L'Host in base al Descriptor che vuole leggere invia al device il relativo comando (Get Descriptor) e questo deve rispondere con il contenuto della struttura dati richiesta.
I descrittori includono due codice noti come *VendorID* e *ProductID.*
Il primo è assegnato dall'USB IF, il secondo può essere assegnato dal produttore del device.
Il *VendorID* è il codice identificativo del venditore e a ciascuno di questi corrisponde 65536 *ProductID.* Quest'ultimo è il codice identificativo del prodotto. La combinazione *VendorID* e *ProductID* è univoca. I Descriptor definiti dalle specifiche USB sono i seguenti:

- Device Descriptor;
- Configuration Descriptor;
- Interface Descriptor;
- Endpoint Descriptor;
- String Descriptor

Il Device Descriptor è utile per i seguenti motivi:

- permette di definire la classe del dispositivo (è possibile farlo anche nell'interface descriptor);
- contiene i parametri *VendorID* e *ProductID;*
- permette di impostare del testo informativo per il dispositivo, ovvero puntatori a stringhe che contengono una breve descrizione del prodotto.

Il Configuration Descriptor è utile per i seguenti motivi:

- permette di stabilire il numero di interfacce implementate nel device;
- informa l'Host sul tipo di alimentazione del device (esterna o fornita dal bus) e sul meccanismo di sospensione dell'attività del bus;
- permette di impostare la massima corrente che il dispositivo assorbirà.

L' Interface Descriptor è utile per i seguenti motivi:

- permette di stabilire il numero di endpoint dell'interfaccia;
- permette di stabilire la classe del dispositivo.

L' Endpoint Descriptor è utile per i seguenti motivi:

- permette di stabilire l'indirizzo dell'endpoint;
- permette di stabilire il tipo di endpoint (Control, Isochronous, Bulk, Interrupt);
- permette di stabilire la massima dimensione del pacchetto dati;

Lo String Descriptor è utile per i seguenti motivi:

- permette di impostare del testo informativo utile per una sintetica panoramica del device.

Oltre a quelli sopra citati, esistono ulteriori tipi di descrittori che vanno sotto al nome di Class Descriptor. Questi sono specifici per la classe del device e vanno aggiunti a quelli standard precedentemente descritti.

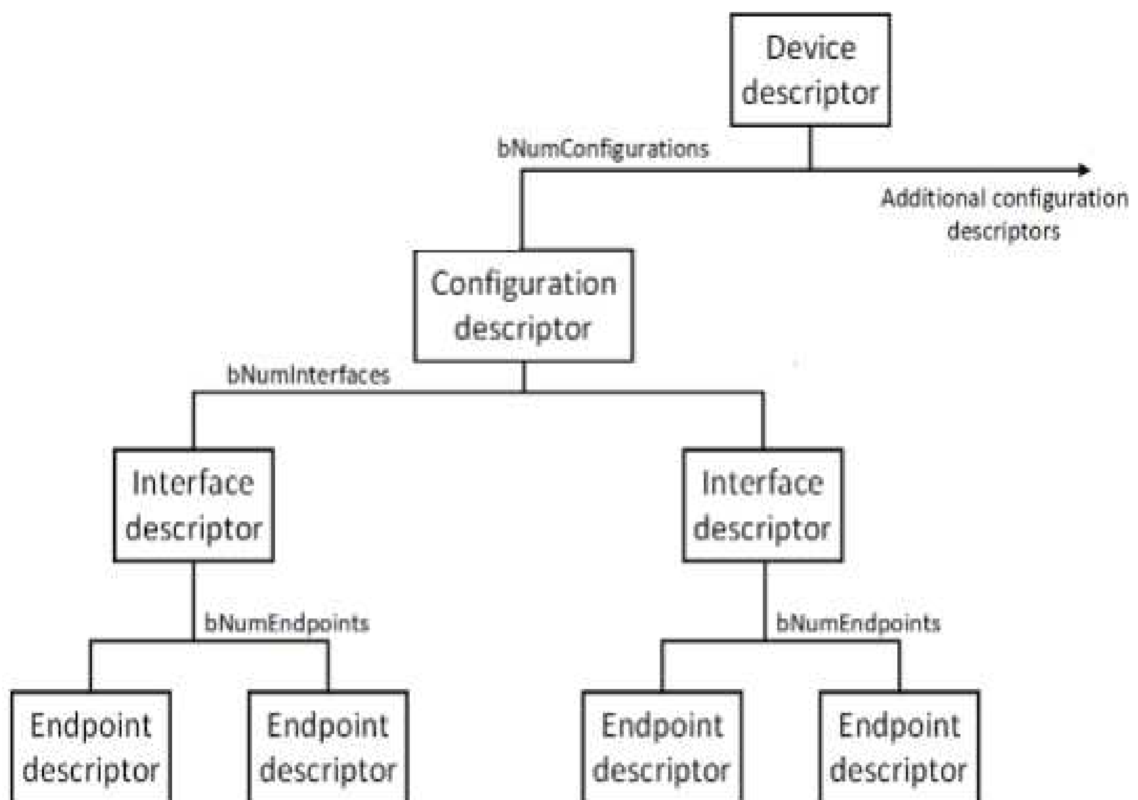

Illustrazione 1.5: descrittori periferica USB

Enumerazione di un device

Ogni volta che un device viene collegato ad un Host, vengono eseguite varie operazioni che vanno sotto al nome di enumerazione del device. Come prima cosa l'Host deve riconoscere che il dispositivo è stato collegato (Attached). Questo avviene controllando le linee del bus USB e il relativo resistore di pull-up che viene inserito dal dispositivo. Questo determina una variazione di tensione sul bus che appunto identifica sia il collegamento del dispositivo che un eventuale scollegamento (Detached) dello stesso. Rilevato il collegamento di un dispositivo alla porta USB, l'Host lo inizializza associandogli un indirizzo unico che verrà poi utilizzato per identificare in maniera univoca la periferica sul bus. Tale valore può variare da connessione a connessione e l'utilizzatore finale non deve compiere alcuna operazione affinché ciò avvenga. Con l'avvenuta inizializzazione viene formato un canale di comunicazione tra l'Host e il dispositivo che prende il nome di PIPE. Tramite tale canale avviene la comunicazione effettiva tra l'Host e il device. Ogni device potrebbe avere più canali di comunicazione a seconda del tipo ma comunque un solo indirizzo. In particolare per ogni verso di comunicazione viene a crearsi una PIPE. La sola formazione logica del canale di comunicazione non è ancora sufficiente per utilizzare il dispositivo, il quale su richiesta deve fornire altre informazioni al fine di classificare il dispositivo stesso e permettere al sistema operativo di caricare il relativo driver.

1.7 - Specifiche Power

Gli aggiornamenti delle specifiche USB hanno coinvolto diversi aspetti. Per quanto riguarda gli aspetti relativi all'alimentazione è bene trattare le varie versioni separatamente visto che diverse versioni hanno specifiche differenti.

Specifiche Power USB 1.x e 2.0

In accordo con le specifiche USB 1.x e 2.0 si possono distinguere due famiglie di device:

- high power device: possono assorbire fino a 500mA;
- low power device: possono assorbire al massimo 100mA.

In entrambi i casi è richiesto che durante l'enumerazione del device la corrente assorbita sia inferiore a 100mA. Affinché un device possa assorbire maggiore corrente è necessario che avvenga l'enumerazione dello stesso e che sia letto il Configuration Descriptor. In particolare, il parametro bMaxPower di tale descrittore contiene il valore massimo della corrente in step di 2mA.
I livelli di corrente supportati pongono dei vincoli ai dispositivi USB, in particolare impone agli stessi un eventuale alimentazione esterna nel caso in cui si voglia supportare una corrente maggiore di 500mA. Un dispositivo potrebbe anche prelevare parte della corrente dalla porta USB e una parte da un alimentatore esterno. In questo contesto si definiscono due tipi di dispositivi:

- Bus Powered: prelevano la corrente direttamente dal bus e non richiedono alimentazione esterna;
- Self Powered: sono dotati di alimentazione propria e possono prelevare anche una parte della corrente dal bus, in accordo con le specifiche USB.

In accordo con le specifiche USB 1.x e 2.0, i device collegati ad una porta USB devono supportare la modalità Suspend State durante la quale è richiesto che vengano ridotti i consumi di corrente a 2.5mA. Si richiede inoltre che l'entrata in Suspend State avvenga mediamente dopo un secondo dal pervenimento della richiesta. La richiesta di Suspend State deve essere inoltrata al device nel caso in cui non ci sia attività sul bus per oltre 3ms.

Specifiche Power USB 3.x

Le specifiche USB 3.x arricchiscono le possibilità offerte dalla porta USB per quanto riguarda la massima corrente erogabile. Come per le specifiche USB 1.x e 2.0, si mantiene la divisione di High Power Device e Low Power Device. In questo caso però, la prima categoria supporta fino a 900mA mentre la seconda fino a 150mA. Un dispositivo conforme alle specifiche USB 2.0 collegato ad una porta USB 3.x deve continuare a rispettare le specifiche USB 2.0 ovvero al massimo la corrente erogata deve essere di 100mA e 500mA rispettivamente per Low Power Device e High Power Device. Per poter utilizzare correnti superiori a 150mA è necessario che il dispositivo venga propriamente inizializzato durante la fase di enumerazione e che il valore di corrente richiesto dal dispositivo sia scritto all'interno del Configuration Descriptor. Il valore massimo della corrente è contenuto nel parametro bMaxPower di tale descrittore ed è espresso in step di 8mA (non più di 2mA come per USB 1.x e 2.0).
Le specifiche USB 3.x allo stesso modo delle specifiche USB 2.0 richiedono che un dispositivo entri in Suspend State qualora non ci siano attività sul bus. Oltre a tale modalità le specifiche USB 3.x introducono altre possibilità per permettere il risparmio di corrente. Questo aspetto è da tenere particolarmente in considerazione per i dispositivi alimentati a batteria. Le diverse modalità di funzionamento sono:

- U0: modalità operativa normale in cui avviene la comunicazione tra Host e i device.
- U1: modalità a basso consumo con rapida transizione alla modalità operativa U0.
- U2: modalità con ulteriore riduzione dei consumi in cui il dispositivo elettronico può anche disattivare il clock di sistema a scapito di un più lungo tempo di transizione per tornare in modalità operativa U0.
- U3: modalità equivalente alla Suspend State.

1.8 - Specifiche meccaniche

I connettori USB

Le specifiche USB sono state realizzate sin dall'inizio con l'idea di permettere dal lato dell'utilizzatore un'esperienza d'uso semplice. Nel corso degli anni, le specifiche USB sono state aggiornate e di conseguenza anche le specifiche relative ai connettori si sono evolute nel tempo. Con l'avvento dell'USB1.x sono stati progettati due diversi tipi di connettore noti come Type A e Type B. In particolare dal lato Host è presente il Type A mentre dal lato del device il Type B. Entrambi si dividono a loro volta in connettore maschio e femmina, o come li definisce l'USB IF plugs e receptacles. A causa dell'incessante miniaturizzazione elettronica, sono stati ideati negli anni ulteriori tipi di connettore con lo scopo di renderli di pratico utilizzo anche per device di dimensioni particolarmente ridotte. Questo ha portato alla definizione dei connettori di tipo Mini A, Micro A, Mini B e Micro B. Con l'introduzione delle specifiche USB 3.x, l'USB IF ha introdotto nuovi connettori per il supporto delle specifiche SuperSpeed (SS) e SuperSpeed+ (SS+).
Questi supportano le nuove modalità introdotte e mantengono retro compatibilità con le precedenti versioni del protocollo.

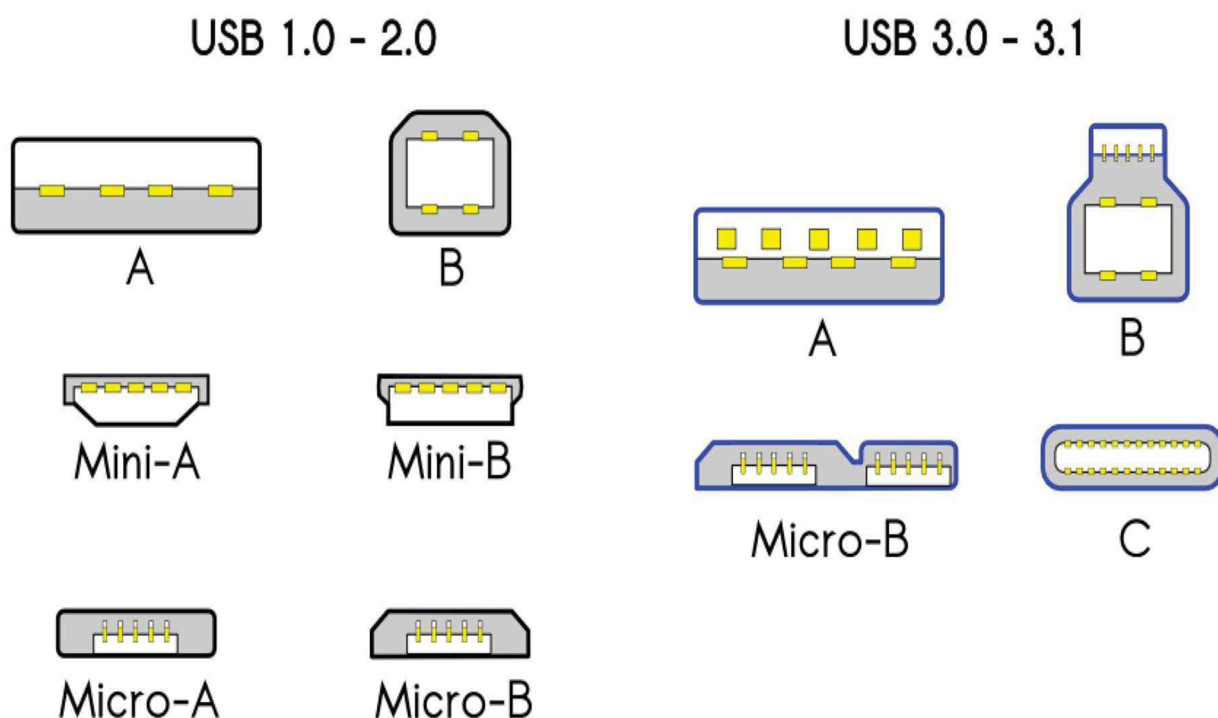

Illustrazione 1.6: connettori USB

Cavi USB

Il cavo utilizzato per le connessioni su bus USB è un cavo schermato a 4 fili. Due di questi sono utilizzati per la comunicazione, i restanti sono usati per distribuire l'alimentazione alle periferiche. Nelle specifiche USB1.0 le caratteristiche dei cavi ne specificavano la lunghezza. In particolare per la modalità Low Speed la massima lunghezza del cavo è di 3m mentre di 5m per la modalità Full Speed. Nonostante l'attenuazione offerta dal cavo sia proporzionale alla frequenza del segnale che lo percorre, in modalità Full Speed il cavo può essere più lungo. Questo perché il cavo USB1.x per la modalità Low Speed è specificato in maniera da essere più economico, in particolare non è previsto lo schermaggio né della coppia dei cavi per i dati né del cavo nel suo complesso. Nel caso di modalità Full Speed, invece, il cavo che include la coppia dati e l'alimentazione deve essere schermato.
La coppia di cavi utilizzati per la linea dati è di tipo *Twisted Pair,* ovvero un cavo intrecciato.
I cavi intrecciati in maniera simmetrica offrono particolare immunità al rumore esterno, in particolare alle radiazioni elettromagnetiche a bassa frequenza, mentre lo schermo conduttivo presente sul cavo USB offre maggiore schermaggio ad alte frequenze.
A partire dalle specifiche USB2.0, l'USB IF non ha più specificato la lunghezza massima dei cavi ma ha posto dei limiti sull'attenuazione offerta dagli stessi in funzione delle diverse frequenze operative dei vari standard. In questo modo in base alla qualità del cavo e alla modalità operativa è possibile utilizzare un cavo più o meno lungo. Nonostante ciò le specifiche USB suggeriscono di non eccedere i 5m anche per la modalità High Speed indipendentemente dalla qualità del cavo utilizzato.

Illustrazione 1.7: cavo USB

2 - Il progetto: firmware device-side

2.1 - Il microcontrollore

Il microcontrollore utilizzato è il Nordic Semiconductor nRF52840.
Questo può funzionare come USB full-speed device (12Mbit/s) ed è completamente compatibile con le specifiche USB2.0.

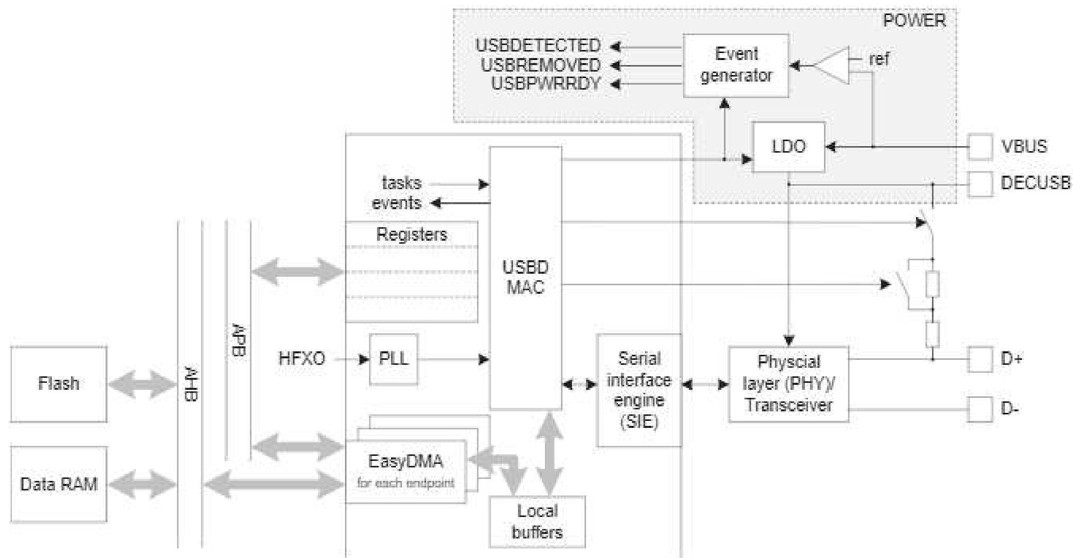

Illustrazione 2.1: diagramma a blocchi del device USB (USBD)

Qui di seguito vengono indicate le principali caratteristiche del device USB:

- Device full-speed 2.0 (12Mbit/s) perfettamente compatibile con le specifiche USB2.0;
- USBD stack disponibile nel Nordic SDK;
- Transceiver (PHY) USB integrato;
- Endpoints:
  - 2 control (1 IN, 1 OUT);
  - 14 bulk/interrupt (7 IN, 7 OUT);

Bulk endpoint #	USB IN endpoint	USB OUT endpoint
[1]	0x81	0x01
[2]	0x82	0x02
[3]	0x83	0x03
[4]	0x84	0x04
[5]	0x85	0x05
[6]	0x86	0x06
[7]	0x87	0x07

- 2 isocroni (1 IN, 1 OUT);

ISO endpoint #	USB IN endpoint	USB OUT endpoint
[0]	0x88	0x08

- Meccanismo di doppio buffering per endpoint isocroni (IN e OUT);
- Supporta la modalità USB suspend, resume e remote wake-up;
- Buffer di 64 bytes per ciascun endpoint bulk e interrupt;
- Buffer fino a 1023bytes per ciascun endpoint isocrono.

Il comportamento generale di un device USB può essere modellato attraverso un diagramma a stati. Le specifiche USB 2.0 definiscono dei particolari stati per il device USB come illustrato qui di seguito:

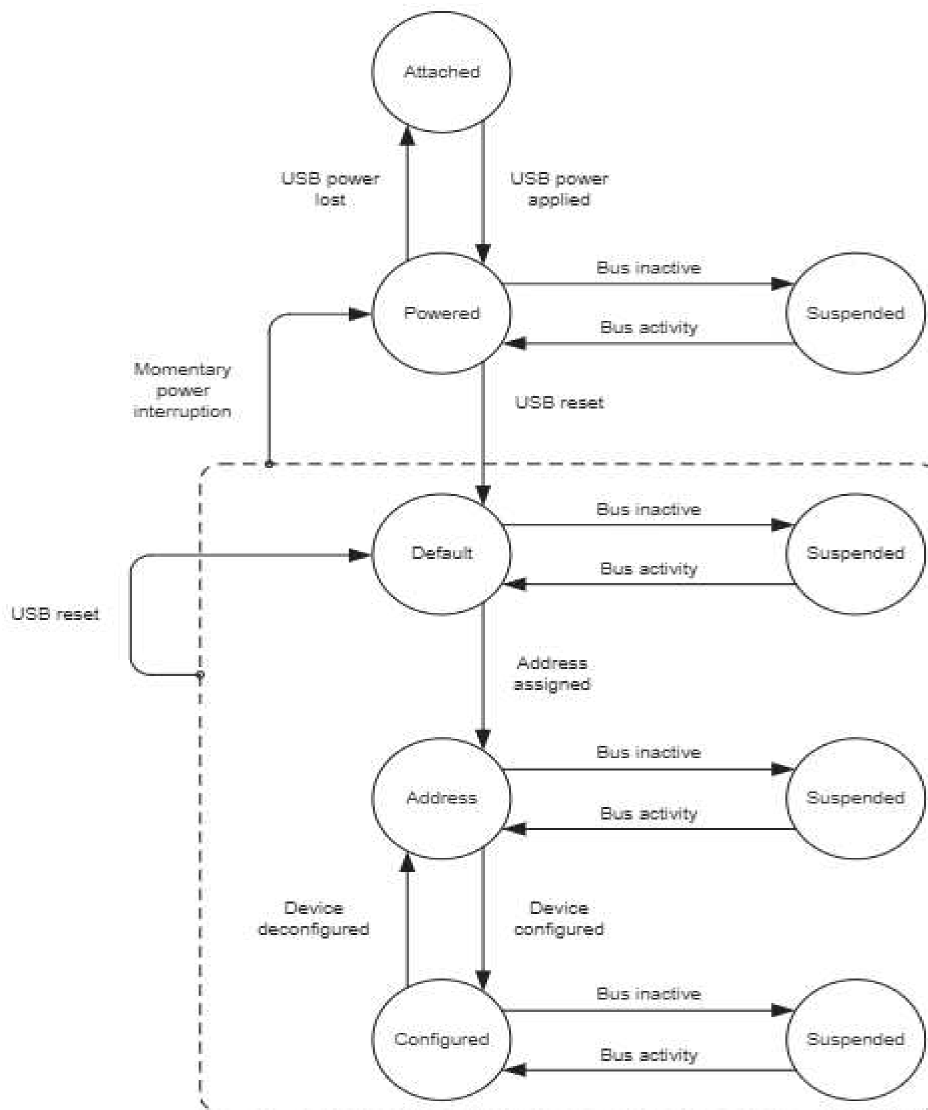

Illustrazione 2.2: diagramma a stati device USB (2.0 spec.)

Il device deve necessariamente cambiare il proprio stato in accordo con il traffico del bus USB.
Gli stati ammessi in accordo con le specifiche USB2.0 sono:

- Attached: device collegato alla porta USB;
- Powered: device collegato alla porta USB (attached) e tensione V_{BUS} applicata al device;
- Default: dopo che il device è stato alimentato (powered), questo non deve rispondere ad alcuna transazione sul bus fino a quando non riceve un reset. Dopo aver ricevuto il reset, il device è indirizzabile all'indirizzo di default. E' bene sottolineare che il reset a cui si fa menzione è una normale condizione del bus USB parte integrante della fase di enumerazione. Questo non deve essere confuso con il chip reset.
- Address: l'Host assegna a ciascun device USB un indirizzo unico dopo il reset;
- Configured: prima di utilizzare le funzionalità del device USB è preliminarmente necessario configurarlo. La configurazione avviene con il processamento della richiesta standard SetConfiguration();
- Suspended: un device entra automaticamente in questo stato quando non si registra traffico sul bus per un tempo superiore a 3ms.

2.2 - Il codice

L'idea di base è quella di implementare un dispositivo USB composito.
Un dispositivo USB composito è una periferica che supporta più di una classe di dispositivi.
Il protocollo USB, infatti, sebbene non definisca un formato dati predefinito, definisce delle classi di dispositivi, associando ad esse particolari caratteristiche da un punto di vista della modalità di trasmissione che deve essere utilizzata. Ogni classe definisce il numero di endpoint che deve attivare il device, i protocolli utilizzati dal device ed eventuali control message specifiche della classe.
Nel caso in esame occorre innanzitutto distinguere due diversi tipi di flusso dati, quello di controllo (SYS) e quello proveniente dai sensori (ECG, EMG e ACC&GYRO).
Il primo è utilizzato per la diagnostica dello stato del sistema ed eventualmente per il debug, il secondo è uno streaming di dati ad alta velocità.
Nel caso in esame, si è pensato di gestire il flusso dati SYS tramite protocollo CDC ACM e il flusso dei dati proveniente dai sensori tramite BULK transfer.
Il protocollo CDC ACM non è adatto per trasferire dati binari pacchettizzati in quanto orientato ai caratteri e non ai pacchetti. Per questo motivo non è adatto per il trasferimento dei dati provenienti dai sensori ma può essere utilizzato per quanto riguarda eventuali segnali di controllo/debug come il SYS.
Dato che, l'unica differenza tra il segnale ECG ed EMG sta nella frequenza di campionamento e dato che il sensore (ADS1293) è provvisto di un filtro che la aggiusta automaticamente, non c'è più motivo di considerare due casi separati ma ci si riferirà nel prosieguo esclusivamente all'EMG per entrambi i casi. Si sceglie per quanto riguarda il trasferimento dei dati provenienti dai sensori di utilizzare bulk transfer in quanto adatte allo scopo e tra le altre cose anche simili al protocollo BLE 4.0 già in uso.
Nel protocollo BLE 4.0 infatti, l'host trasmette periodicamente il "connection event" e se sono disponibili dati da trasmettere questi vengono trasmessi immediatamente. In una bulk transfer, l'analogo del "connection event" è il token IN packet.

Classe CDC ACM

La classe CDC (Communication Device Class) è utilizzata principalmente per dispositivi di telecomunicazione come telefoni digitali e per dispositivi di rete come modem ADSL o adattatori Ethernet. La classe specifica più sottoclassi in modo da supportare diversi tipi di dispositivi di comunicazione. Una di queste è la sottoclasse ACM (Abstract Control Model), questa è essenzialmente utilizzata in quanto permette di emulare una porta seriale (COM).
Tipicamente si parla di classe CDC ACM con riferimento alla specifica sottoclasse ACM della classe CDC. Questa ha due interfacce:

- interfaccia COMM: contiene un interrupt IN endpoint ed è utilizzata per notificare all'host lo stato della porta seriale.
- interfaccia DATA: contiene un bulk IN endpoint ed un bulk OUT endpoint. Questa è utilizzata per la ricezione e la trasmissione dei dati.

La classe CDC ACM è supportata dal microcontrollore utilizzato pertanto sono disponibili API specifiche per la definizione della classe, per l'invio e la ricezione dei dati e per la gestione in generale del device CDC ACM. In questo caso per accedere al device, occorre host-side distinguere diversi casi:

- sistema operativo Linux: non è necessario installare alcun driver ed il device sarà visto come /dev/ttyACM0 o similari;
- sistema operativo Windows 8.1 o più obsoleti: il driver “usbser” non è automaticamente assegnato al device CDC ACM e per questo un file .inf deve essere necessariamente installato. Fatto questo, si può accedere al device utilizzando emulatori di terminale come il PuTTy o il Teraterm.
- Sistema operativo Windows 10: non è necessario installare alcun driver e si può accedere al dispositivo utilizzando direttamente PuTTY o Teraterm.

Classe personalizzata “MY CLASS”

Come già detto in precedenza, l'idea è quella di implementare un device USB composito in grado cioè di supportare più di una classe di dispositivi. L'USB-IF mette a disposizione una vasta gamma di classi, tuttavia, adattarne una preesistente comporterebbe degli svantaggi sia in termini di difficoltà di realizzazione sia per la gestione host-side. Per questo motivo, piuttosto che adattare una classe preesistente si è scelto di crearne una personalizzata che fosse perfettamente adatta allo scopo in quanto definita appositamente.
La classe in questione ha un'interfaccia che contiene una coppia di bulk endpoint (1 IN, 1 OUT).
Il bulk IN endpoint è utilizzato per il trasferimento dei dati provenienti dai sensori dal device all'host. Il bulk OUT endpoint è stato invece introdotto per garantire la possibilità di instaurare una comunicazione nel verso opposto quindi dall'host al device.
Infatti, è necessario comunicare al device ad esempio la frequenza di campionamento dell'ADC, il gain degli amplificatori e altre informazioni accessorie che permettono al device di impostare correttamente il formato dei dati. Abbiamo già detto infatti che l'unica differenza tra segnale ECG ed EMG è la frequenza di campionamento, è quindi indispensabile per il device conoscere tra le altre cose anche tale frequenza di campionamento. Questa deve essere comunicata dall'host al device in accordo con lo specifico segnale che si vuole richiedere al device stesso. Da qui, l'esigenza di un bulk OUT endpoint.
Essendo tale classe completamente customizzata, non esistono API predefinite per la specifica classe ed inoltre si dovrà necessariamente realizzare un programma host-side in grado di accedere al dispositivo. Dato che, come menzionato in precedenza, per la classe CDC ACM si hanno API predefinite di alto livello, al fine di evitare di utilizzare API di livello differente si è scelto di utilizzare per la gestione della classe personalizzata istruzioni dello stesso livello di quelle utilizzate per la classe CDC ACM. Questa è buona norma in quanto si è constatato che l'utilizzo di API di differente livello nello stesso firmware può causare il malfunzionamento dello stesso a causa di conflitti che si instaurano tra le diverse istruzioni.
A tal scopo si è pensato di realizzare una libreria specifica per la classe personalizzata in modo da poter accedere alla stessa in maniera analoga a quanto si fa per la classe CDC ACM.
La libreria in questione verrà descritta qui di seguito nel dettaglio.

Libreria “app_usbd_myclass.c”

La libreria “app_usbd_myclass.c” verrà scandagliata sezione per sezione qui di seguito.

Inclusione header file

```
#include "sdk_common.h" // Contiene tutti gli header file necessari al funzionamento degli esempi dell'SDK
#include "app_usbd_myclass.h" // Contiene i prototipi delle funzioni non statiche definite nel file .c
#include <inttypes.h> // Header file che permette di utilizzare tipi di dato interi.
```

Definizione di “myclass_get”

```
/*
La parola chiave inline è una speciale direttiva al compilatore che, se eseguita, consiste nel sostituire
la chiamata a funzione con il corpo della funzione stessa.
La funzione è inoltre dichiarata come statica il che significa che è accessibile soltanto da questo file .c
A partire da un'istanza USB generica la funzione restituisce un'istanza USB di tipo myclass.
*/
static inline app_usbd_myclass_t const * myclass_get(app_usbd_class_inst_t const * p_inst)
{
    ASSERT(p_inst != NULL);
    return (app_usbd_myclass_t const *)p_inst;
}
```

Definizione del gestore eventi per la classe “my class”

```
// La funzione qui di seguito è il gestore di eventi relativo all'istanza di tipo myclass.
static ret_code_t myclass_event_handler(app_usbd_class_inst_t const *  p_inst,
                                        app_usbd_complex_evt_t const * p_event)
{
    ASSERT(p_inst != NULL);
    ASSERT(p_event != NULL);

    ret_code_t ret = NRF_SUCCESS;
    switch (p_event->app_evt.type)
    {
        case APP_USBD_EVT_DRV_SOF:
            break;
        case APP_USBD_EVT_DRV_RESET:
            // Reset ADC ... cdc_acm_reset_port(p_inst);
            break;
        case APP_USBD_EVT_DRV_SETUP:
            //ret  = setup_event_handler(p_inst, (app_usbd_setup_evt_t const *)p_event);
            break;
        case APP_USBD_EVT_DRV_EPTRANSFER:
            //ret = cdc_acm_endpoint_ev(p_inst, p_event);
            break;
        case APP_USBD_EVT_DRV_SUSPEND:
            break;
        case APP_USBD_EVT_DRV_RESUME:
            break;
        case APP_USBD_EVT_INST_APPEND:

            break;
        case APP_USBD_EVT_INST_REMOVE:
            break;
        case APP_USBD_EVT_STARTED:
            break;
        case APP_USBD_EVT_STOPPED:
            break;
        case APP_USBD_EVT_POWER_REMOVED:
           // user_event_handler(p_inst, APP_USBD_CDC_ACM_USER_EVT_PORT_CLOSE);
           //cdc_acm_reset_port(p_inst);
            break;
        default:
            ret = NRF_ERROR_NOT_SUPPORTED;
            break;
    }

    return ret;
}
```

Definizione di “myclass_feed_descriptors”

Questa è la parte centrale della libreria “app_usbd_myclass.c” in quanto contiene tutti i descrittori relativi alla classe personalizzata. Un device, una volta che lo si collega ad un Host, viene enumerato. Affinché ciò avvenga è necessario che l'Host recuperi dal device informazioni utili al fine di poterlo identificare e caricare i driver corretti.

Tutte le informazioni che permettono all'Host di comunicare correttamente con il dispositivo, inizializzarlo e determinare il driver da caricare, sono contenute all'interno di vari descrittori che risiedono nel device. Le informazioni sono distribuite in diversi descrittori ciascuno dei quali caratterizzato da un codice identificativo. L'Host in base al Descriptor che vuole leggere invia al device il relativo comando (Get Descriptor) che contiene il codice del Descriptor che vuole leggere e il device deve rispondere con il contenuto della struttura dati richiesta.

```
// La funzione in questione è la parte centrale del file app_usbd_myclass.c e permette di definire i descrittori
static bool myclass_feed_descriptors(app_usbd_class_descriptor_ctx_t * p_ctx,
                                     app_usbd_class_inst_t const     * p_inst,
                                     uint8_t                         * p_buff,
                                     size_t                            max_size)
{
    static uint8_t ifaces    = 0;
    ifaces = app_usbd_class_iface_count_get(p_inst); // restituisce il numero di interfacce
    app_usbd_myclass_t const * p_myclass = myclass_get(p_inst);
    // p_myclass è un istanza di tipo myclass in accordo con la definizione di "myclass_get"

    APP_USBD_CLASS_DESCRIPTOR_BEGIN(p_ctx, p_buff, max_size);

    static uint8_t i = 0;

    for (i = 0; i < ifaces; i++)
    {
        /* INTERFACE DESCRIPTOR */
        APP_USBD_CLASS_DESCRIPTOR_WRITE(0x09); // bLength
        APP_USBD_CLASS_DESCRIPTOR_WRITE(APP_USBD_DESCRIPTOR_INTERFACE); // bDescriptorType = Interface

        static app_usbd_class_iface_conf_t const * p_cur_iface = NULL;
        p_cur_iface = app_usbd_class_iface_get(p_inst, i);
        // p_cur_iface contiene l'informazione dell'interfaccia

        APP_USBD_CLASS_DESCRIPTOR_WRITE(app_usbd_class_iface_number_get(p_cur_iface)); // bInterfaceNumber
        APP_USBD_CLASS_DESCRIPTOR_WRITE(0x00); // bAlternateSetting
        APP_USBD_CLASS_DESCRIPTOR_WRITE(0x02); //app_usbd_class_iface_ep_count_get(p_cur_iface)); // bNumEndpoints
        APP_USBD_CLASS_DESCRIPTOR_WRITE(0xff); // bInterfaceClass = vendor specific
        APP_USBD_CLASS_DESCRIPTOR_WRITE(0x00); // bInterfaceSubclass
        APP_USBD_CLASS_DESCRIPTOR_WRITE(0x00); // bInterfaceProtocol
        APP_USBD_CLASS_DESCRIPTOR_WRITE(0x00); // iInterface
```

Finora si è descritta l'interfaccia, in particolare occorre notare che si è specificato il numero di endpoint e la classe (vendor specific). Si passa ora alla descrizione degli endpoint.

```
        /* ENDPOINT DESCRIPTORS */
        static uint8_t endpoints = 0;
        endpoints = app_usbd_class_iface_ep_count_get(p_cur_iface); // Restituisce il numero di endpoint

        static uint8_t j = 0;

        for (j = 0; j < endpoints; j++)
        {
            APP_USBD_CLASS_DESCRIPTOR_WRITE(0x07);                          // bLength
            APP_USBD_CLASS_DESCRIPTOR_WRITE(APP_USBD_DESCRIPTOR_ENDPOINT); // bDescriptorType = Endpoint

            static app_usbd_class_ep_conf_t const * p_cur_ep = NULL;
            p_cur_ep = app_usbd_class_iface_ep_get(p_cur_iface, j);
            APP_USBD_CLASS_DESCRIPTOR_WRITE(app_usbd_class_ep_address_get(p_cur_ep)); // bEndpointAddress
            APP_USBD_CLASS_DESCRIPTOR_WRITE(APP_USBD_DESCRIPTOR_EP_ATTR_TYPE_BULK); // bmAttributes
            APP_USBD_CLASS_DESCRIPTOR_WRITE(LSB_16(NRF_DRV_USBD_EPSIZE)); // wMaxPacketSize LSB
            APP_USBD_CLASS_DESCRIPTOR_WRITE(MSB_16(NRF_DRV_USBD_EPSIZE)); // wMaxPacketSize MSB
            APP_USBD_CLASS_DESCRIPTOR_WRITE(0x00); // bInterval

         }

     }

    APP_USBD_CLASS_DESCRIPTOR_END();
}
```

Per quanto riguarda gli endpoint si specifica l'indirizzo, il tipo, la dimensione del buffer relativo all'endpoint ed inoltre un parametro che nel codice è indicato con bInterval. Quest'ultimo parametro è utile nel caso di endpoint di tipo interrupt o isocrono mentre è ignorato per endpoint di tipo bulk.

Definizione della struttura "app_usbd_myclass_class_methods"

```
// Struttura in cui vado a specificare il nome del gestore di eventi ed il nome della funzione relativa ai descrittori
// Questo è un paradigma di programmazione molto comune, si definiscono dapprima delle funzioni statiche e poi si definisce
// una struttura in cui si mettono dei puntatori alle funzioni precedentemente definite.
const app_usbd_class_methods_t app_usbd_myclass_class_methods = {
        .event_handler = myclass_event_handler,
        .feed_descriptors = myclass_feed_descriptors,
};
```

File header "app_usbd_myclass.h"

Il file header "app_usbd_myclass.h" incluso all'inizio del corrispondente file .c deve contenere i prototipi delle eventuali funzioni non statiche presenti in quest'ultimo.
Questo serve per informare il compiler dell'esistenza di una funzione che ha un determinato nome e vuole certi argomenti. Quando nel programma principale questa funzione viene incontrata il compiler può chiamarla, sarà poi compito del linker nell'ultima fase di compilazione reperirla.
Entriamo ora nel dettaglio del file header "app_usbd_myclass.h".

```
#include <stdint.h>
#include <stdbool.h>
#include "nrf_drv_usbd.h"
#include "app_usbd_class_base.h"
#include "app_usbd.h"
#include "app_usbd_core.h"
#include "app_usbd_descriptor.h"
#include "app_util.h"
#include "app_usbd_cdc_desc.h"
#include "app_usbd_cdc_types.h"
#include "app_usbd_cdc_acm_internal.h"

APP_USBD_CLASS_FORWARD(app_usbd_myclass);

typedef struct {
    uint8_t dummy;
} app_usbd_myclass_inst_t;

#define APP_USBD_MYCLASS_CONFIG(iface, epin1, epin2)            \
  ((iface, epin1, epin2))
// La macro di cui sopra serve per specificare che la configurazione della classe "myclass" prevede
// un interfaccia e due endpoint

extern const app_usbd_class_methods_t app_usbd_myclass_class_methods;
// La struttura di cui sopra è definita nel file .c e contiene il nome del gestore di eventi
// ed il nome della funzione relativa ai descrittori.
```

```
#define APP_USBD_MYCLASS_GLOBAL_DEF(instance_name,                        \
                                    data_ifc,                             \
                                    data_ep1,                             \
                                    data_ep2)                             \
        APP_USBD_CLASS_INST_GLOBAL_DEF(                                                  \
                instance_name,                                                           \
                app_usbd_myclass,                                                        \
                &app_usbd_myclass_class_methods,                                         \
                APP_USBD_MYCLASS_CONFIG(data_ifc, data_ep1, data_ep2),        \
                ()                      \
        );

#ifdef DOXYGEN
typedef struct {} app_usbd_myclass_t;
#else
APP_USBD_CLASS_TYPEDEF(app_usbd_myclass,                            \
           APP_USBD_MYCLASS_CONFIG(0,0,0),                  \
           APP_USBD_CLASS_INSTANCE_SPECIFIC_DEC_NONE,               \
           APP_USBD_CLASS_INSTANCE_SPECIFIC_DEC_NONE);
#endif

static inline app_usbd_class_inst_t const*
app_usbd_myclass_inst_get(app_usbd_myclass_t const * myclass)
{
return &myclass->base; // return base class
}
```

Nel file .h si definisce in maniera globale l'istanza di classe customizzata per mezzo di una macro. Quanto fatto è molto simile a quello che si fa per la classe CDC ACM. La classe CDC ACM è tuttavia supportata nel Nordic SDK e quindi non è stata necessaria la scrittura del file .c e neppure del corrispondente file header.
Oltre a ciò nel file header è definita la funzione “app_usbd_myclass_inst_get”.
Questa ritorna a partire dalla classe, l'istanza che poi verrà aggiunta nel main tramite l'istruzione “app_usbd_class_append”.
Dopo aver descritto nel dettaglio la libreria “app_usbd_myclass.c” e il corrispondente file header .h, si passa alla descrizione del programma principale.

Programma principale

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

#include "nrf.h"
#include "nrf_drv_usbd.h"
#include "nrf_drv_clock.h"
#include "nrf_gpio.h"
#include "nrf_delay.h"
#include "nrf_drv_power.h"
#include "app_error.h"
#include "app_util.h"
#include "boards.h"

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

#include "bsp.h"
#include "bsp_cli.h"

#include "nrf_cli.h"
#include "nrf_cli_uart.h"

#include "app_timer.h"

#include "app_usbd_serial_num.h"
#include "app_usbd_core.h"
#include "app_usbd_string_desc.h"
#include "app_usbd.h"

#include "app_usbd_cdc_acm.h"
#include "app_usbd_myclass.h" // <--- inclusion of the header file described above.
```

Inizialmente si includono i file header delle librerie che verranno poi utilizzate nel prosieguo del codice.
Fatto questo si passa alla definizione di pulsanti e LEDs. In particolare, si definisce:

- BTN_SYSTEM_OFF: pulsante che consente di spegnere completamente il sistema;
- LED_USB_STATUS: led che indica lo stato del sistema;
- LED_USB_POWER: led che indica lo stato dell'alimentazione del sistema;
- LED_RECEIVE;
- LED_SEND;

LED_RECEIVE e LED_SEND sono utilizzati per indicare il verso della trasmissione dati.
E' importante sottolineare che in tal caso RECEIVE/SEND sono riferiti al device e non all'host.

```
/*
Button for system OFF request.
This button would set the request for system OFF.
*/
#define BTN_SYSTEM_OFF     BSP_BOARD_BUTTON_1 // This is Button 2 on nRF52840 DK

/*
This LED is on when USB work properly.
It is turned OFF when a system OFF request is received.
*/
#define LED_USB_STATUS     BSP_BOARD_LED_0   // This is LED1 on nRF52840 DK

/*
Power detect LED.
The LED is ON when connection is detected on USB port.
It is turned off when connection is removed.
*/
#define LED_USB_POWER      BSP_BOARD_LED_1 // This is LED2 on nRF52840 DK

#define LED_RECEIVE        BSP_BOARD_LED_2
#define LED_SEND           BSP_BOARD_LED_3
/*
Enable power USB detection.
Configure if example supports USB port connection.
USBD_POWER_DETECTION is used to set if you want (or not) to have events when device is connected to USB port.
This is useful in case of external battery devices that are not always connected to the USB port.
If USBD_POWER_DETECTION=true, USB isn't automatically enabled but APP_USBD_EVT_POWER_DETECTED event is expected.
*/
#ifndef USBD_POWER_DETECTION
#define USBD_POWER_DETECTION true
#endif
#define STARTUP_DELAY 100 // Number of microseconds to start USBD after powering up.

// The macro defined below will be described later...
static void cdc_acm_user_ev_handler(app_usbd_class_inst_t const * p_inst, app_usbd_cdc_acm_user_event_t event);
```

Definizione dell'istanza di classe CDC ACM

```
/*
CDC (Comunication Device Class)_
- 1 interrupt IN endpoint for notifications from device to host;
- 1 bulk IN endpoint for data transfer from device to host;
- 1 bulk OUT endpoint for data transfer from host to device;
*/
#define CDC_ACM_COMM_INTERFACE      1
#define CDC_ACM_COMM_EPIN           NRF_DRV_USBD_EPIN4  // Interrupt IN endpoint
#define CDC_ACM_DATA_INTERFACE      2
#define CDC_ACM_DATA_EPIN           NRF_DRV_USBD_EPIN3  // Bulk IN endpoint
#define CDC_ACM_DATA_EPOUT          NRF_DRV_USBD_EPOUT3 // Bulk OUT endpoint

APP_USBD_CDC_ACM_GLOBAL_DEF (m_app_cdc_acm,
                             cdc_acm_user_ev_handler,
                             CDC_ACM_COMM_INTERFACE,
                             CDC_ACM_DATA_INTERFACE,
                             CDC_ACM_COMM_EPIN,
                             CDC_ACM_DATA_EPIN,
                             CDC_ACM_DATA_EPOUT,
                             APP_USBD_CDC_COMM_PROTOCOL_NONE);
```

La classe CDC ACM (Communication Device Class Abstract Control Model), come abbiamo già detto, dispone dell'interfaccia COMM e delll'interfaccia DATA.
In accordo con il codice di cui sopra, la prima è l'interfaccia numero 1 mentre la seconda è l'interfaccia numero 2.
L'interfaccia 1 (COMM) contiene un interrupt IN endpoint, nel nostro caso l'endpoint IN 4, quello cioè di indirizzo 0x84. L'interfaccia 2 (DATA) contiene una coppia di endpoint (1 IN e 1 OUT) di tipo bulk. In particolare la coppia scelta è l'endpoint IN 3 (0x83) e l'endpoint OUT 3 (0x03).
Fatte queste scelte, mediante la macro APP_USBD_CDC_ACM_GLOBAL_DEF si definisce globalmente l'istanza di classe CDC ACM. I parametri d'ingresso della macro sono in ordine:

- nome dell'istanza;
- gestore di eventi dell'istanza (precedentemente definito ma non ancora descritto);
- numero dell'interfaccia COMM;
- numero dell'interfaccia DATA,
- indirizzo IN endpoint contenuto nell'interfaccia COMM;
- indirizzo IN endpoint contenuto nell'interfaccia DATA;
- indirizzo OUT endpoint contenuto nell'interfaccia DATA;
- bInterfaceProtocol: nel nostro caso “APP_USBD_CDC_COMM_PROTOCOL_NONE” vale 0x00. Questa scelta corrisponde all'esigenza di implementare un'istanza di classe CDC e sottoclasse ACM. Se invece, si volesse ad esempio creare un istanza di classe CDC con l'intento non di emulare una porta seriale ma un protocollo Ethernet allora dovremmo specificarlo sostituendo ad “APP_USBD_CDC_COMM_PROTOCOL_NONE” la macro “APP_USBD_CDC_COMM_PROTOCOL_EEM”.
  In tal senso si realizzerebbe un istanza di classe CDC NCM (Network Control Model).

Definizione dell'istanza di classe “MYCLASS”

Definita l'istanza di classe CDC ACM, si definiscono due istanze di classe “myclass”. Come detto in precedenza, la classe “myclass” prevede un bulk IN endpoint ed un bulk OUT endpoint. Si è scelto di definire due diverse istanze, una per l'ECG/EMG e l'altra per l'ACC&GYRO. Si definiscono separatamente le due istanze in modo da garantire elevata flessibilità. In questo modo host-side si può decidere se accedere ad una, all'altra o ad entrambe contemporaneamente.
La macro APP_USBD_MYCLASS_GLOBAL_DEF è stata definita in “app_usbd_myclass.h”.
I parametri di ingresso sono in ordine:

- nome dell'istanza;
- numero dell'interfaccia;
- indirizzo endpoint IN;
- indirizzo endpoint OUT;

Ricordiamo che il tipo degli endpoint è specificato in “myclass_feed_descriptor” nella libreria “app_usbd_myclass.c”.

```
APP_USBD_MYCLASS_GLOBAL_DEF (m_app_myclass1, 0, NRF_DRV_USBD_EPIN1, NRF_DRV_USBD_EPOUT1);
APP_USBD_MYCLASS_GLOBAL_DEF (m_app_myclass2, 3, NRF_DRV_USBD_EPIN2, NRF_DRV_USBD_EPOUT2);
```

Per l'istanza m_app_myclass1 l'interfaccia è la numero 0, l'endpoint IN è quello all'indirizzo 0x81 e l'endpoint OUT è quello all'indirizzo 0x01.
Per l'istanza m_app_myclass2 l'interfaccia è la numero 3, l'endpoint IN è quello all'indirizzo 0x82 e l'endpoint OUT è quello all'indirizzo 0x02.
Il codice prosegue con la definizione del buffer di ricezione (m_rx_buffer) e di trasmissione (m_tx_buffer) utilizzati per la gestione dei dati relativi all'istanza di classe CDC ACM.

```
#define READ_SIZE 1
static char m_rx_buffer[READ_SIZE]; // RX buffer
static char m_tx_buffer[NRF_DRV_USBD_EPSIZE]; // TX buffer

int flag=0; // This flag is used to recognize a particular character in the RX buffer.
/*
flag=1 when the received character is "s"
flag=2 when the received character is "b"
flag=3 when the received character is "n"
*/
```

Occorre sottolineare inoltre la definizione della variabile intera di nome “flag”. Questa è utilizzata per riconoscere particolari caratteri nel buffer di ricezione.
In seguito viene definito il gestore degli eventi relativi all'istanza di classe CDC ACM.

```
static void cdc_acm_user_ev_handler(app_usbd_class_inst_t const * p_inst, app_usbd_cdc_acm_user_event_t event)
{
    app_usbd_cdc_acm_t const * p_cdc_acm = app_usbd_cdc_acm_class_get(p_inst);
    // The function returns a CDC ACM instance from a generic instance.
    // p_inst is a generic instance, p_cdc_acm is a CDC ACM istance.
    switch (event)
    {
        case APP_USBD_CDC_ACM_USER_EVT_PORT_OPEN:
        {

            /*Setup first transfer*/
            ret_code_t ret = app_usbd_cdc_acm_read(&m_app_cdc_acm,
                                                   m_rx_buffer,
                                                   READ_SIZE);
           /*
           "app_usbd_cdc_acm_read" is a function used to read data from CDC ACM port.
           [in]   p_cdc_acm    CDC ACM instance
           [out]  p_buf        Output buffer
           [in]   length       number of bytes to read
           If there's enough data in internal buffer to fill user buffer (m_rx_buffer),
           NRF_SUCCESS is returned and data is immediately available in m_rx_buffer.
          */
            UNUSED_VARIABLE(ret);
            break;
        }
        case APP_USBD_CDC_ACM_USER_EVT_PORT_CLOSE:
            break;
        case APP_USBD_CDC_ACM_USER_EVT_TX_DONE:
            bsp_board_led_invert(LED_SEND);
            break;
        case APP_USBD_CDC_ACM_USER_EVT_RX_DONE:
        {
            bsp_board_led_invert(LED_RECEIVE);
            ret_code_t ret;
            NRF_LOG_INFO("Bytes waiting: %d", app_usbd_cdc_acm_bytes_stored(p_cdc_acm));
            // " app_usbd_cdc_acm_bytes_stored(p_cdc_acm)" returns number of bytes stored in internal buffer
               do
               {
                   /*Get amount of data transfered*/
                   size_t size = app_usbd_cdc_acm_rx_size(p_cdc_acm);
                   // app_usbd_cdc_acm_rx_size(p_cdc_acm) returns number of bytes to read.
                   NRF_LOG_INFO("RX: size: %lu char: %c", size, m_rx_buffer[0]);

                   /* Fetch data until internal buffer is empty */
                   ret = app_usbd_cdc_acm_read(&m_app_cdc_acm,
                                               m_rx_buffer,
                                               READ_SIZE);
               } while (ret == NRF_SUCCESS);
               if (m_rx_buffer[0]=='s')
               {
               flag=1;
               }
               if (m_rx_buffer[0]=='b')
               {
               flag=2;
               }
               if (m_rx_buffer[0]=='n')
               {
               flag=3;
               }
               // The loop ends when internal buffer is empty.
               break;
          }
          default:
               break;
      }
  }
```

Valutiamo nel dettaglio i diversi eventi che sono gestiti:

- APP_USBD_CDC_ACM_USER_EVT_PORT_OPEN: apertura della porta CDC ACM;
- APP_USBD_CDC_ACM_USER_EVT_PORT_CLOSE: chiusura della porta CDC ACM;
- APP_USBD_CDC_ACM_USER_EVT_RX_DONE: trasferimento dati da host a device;
- APP_USBD_CDC_ACM_USER_EVT_TX_DONE: trasferimento dati da device ad host.

Il verso RX/TX della trasmissione è quindi riferito al device.
Quando si verifica l'evento APP_USBD_CDC_ACM_USER_EVT_RX_DONE significa che il device ha ricevuto dall'host un carattere. Questo viene memorizzato nel buffer di ricezione. A questo punto si valuta il carattere ricevuto indagando sul contenuto di m_rx_buffer, in particolare se è “s” si impone il flag uguale ad 1, uguale a 2 se è “b” ed infine uguale a 3 se è “n”.
Si vedrà poi nel seguito che l'evento APP_USBD_CDC_ACM_USER_EVT_TX_DONE si verifica ogni qual volta che il carattere ricevuto è “s” in quanto è proprio questo il caso in cui il device invia dati all'host (si utilizza la funzione app_usbd_cdc_acm_write).
Contestualmente al verificarsi degli eventi di cui sopra, vengono aggiornati gli stati dei LEDs.
Fatto questo si definiscono dei flag, utili al fine di identificare lo stato del sistema.
L’effetto del qualificatore “volatile” è quello di inibire la capacità di ottimizzazione del compilatore selettivamente per le variabili cui esso è applicato. Quando una variabile è “volatile” quindi, i blocchi di istruzioni in cui essa è impiegata saranno esclusi dall’applicazione di alcune ottimizzazioni. Se non si segnala una variabile come “volatile”, il compilatore suppone che quella variabile non cambia perché non vede assegnamenti ad essa e quindi può ottimizzare il codice e ignorare alcune condizioni richieste perché “pensa” che non sia possibile che sia cambiata.

```
/*
USB configured flag
The flag that is used to mark the fact that USB is configured and ready to transmit data.
*/
static volatile bool m_usbd_configured = true;

/*
System OFF request flag
This flag is used in button event processing and marks the fact that
system OFF should be activated from main loop.
*/
static volatile bool m_system_off_req = false;

/*
USB suspended.
The flag that is used to mark the fact that USB is suspended and requires wake up if new data is available.
NOTE: This variable is changed from the main loop.
*/
static bool m_usbd_suspended = false;

/*
The requested suspend state.
The currently requested suspend state based on the events received from USBD library.
If the value here is different than the @ref m_usbd_suspended
the state changing would be processed inside main loop.
*/
static volatile bool m_usbd_suspend_state_req = false;
```

Successivamente, si descrive il gestore di eventi relativo al device USB (USBD).
Questo non deve essere confuso con quello definito in precedenza (cdc_acm_user_ev_handler).
Quest'ultimo infatti si occupa della gestione degli eventi provenienti dalla sola istanza di classe CDC ACM.

```
static void usbd_user_ev_handler(app_usbd_event_type_t event)
{
    switch (event)
    {
        case APP_USBD_EVT_DRV_SUSPEND:
        /*
        A USB device will enter suspend when there is no activity on the bus (idle) for a given time.
        The USBD peripheral automatically detects lack of activity for more than a defined amount of time,
        and performs steps needed to enter suspend.
        When no activity is detected for longer than tUSB,SUSPEND (3ms),
        the USBD generates the USBEVENT with SUSPEND bit set in register EVENTCAUSE.
        A device which has been 'Suspended', as a result of no bus activity,
        must reduce its current consumption to 0.5 mA or less.
       */
         NRF_LOG_INFO("SUSPEND state detected");
         m_usbd_suspend_state_req = true; // Suspension request detected.
         break;
        case APP_USBD_EVT_DRV_RESUME:
        // A USB device will resume from suspend state when it receives any non idle signalling.
         NRF_LOG_INFO("RESUMING from suspend");
         m_usbd_suspend_state_req = false; // Suspension request not detected.
        break;
        case APP_USBD_EVT_STARTED:
            break;
        case APP_USBD_EVT_STOPPED:
            app_usbd_disable();
            bsp_board_leds_off();
            break;

        default:
            break;
    }
}
```

Gli eventi gestiti sono:

- APP_USBD_EVT_DRV_SUSPEND: generato quando il device deve entrare nella modalità di risparmio energetico (suspended mode).
- APP_USBD_EVT_DRV_RESUME: generato quando il device esce dalla modalità di risparmio energetico. In particolare ciò può accadere per due motivi:
    - l'host ricomincia a mandare frames al device;
    - se il device supporta la modalità remote wake-up e questa è abilitata dall'host, il device può richiedere all'host di uscire dalla modalità di risparmio energetico;
- APP_USBD_EVT_STARTED: generato quando il device USB è stato correttamente inizializzato;
- APP_USBD_EVT_STOPPED: generato quando il dispositivo USB viene stoppato nel suo normale funzionamento. In tal caso si provvede a disabilitare il dispositivo stesso.

Finora abbiamo visto come vengono gestiti gli eventi specifici dell'istanza di classe “myclass”, quelli relativi all'istanza di classe CDC ACM ed infine quelli relativi all'intero device USB.
Oltre a questi, si devono gestire una serie di altri eventi che sono relativi all'alimentazione del dispositivo USB. A tal fine si definisce il gestore di eventi relativo all'alimentazione del dispositivo USB (power_usb_event_handler). Questo è descritto nel dettaglio qui di seguito:

```
// USB power event handler
static void power_usb_event_handler(nrf_drv_power_usb_evt_t event)
{
    switch (event)
    {
    /*
    There are three different event:
    NRF_DRV_POWER_USB_EVT_DETECTED      USB power detected on the connector (plugged in).
    NRF_DRV_POWER_USB_EVT_REMOVED       USB power removed from the connector.
    NRF_DRV_POWER_USB_EVT_READY         USB power regulator ready.

    */
    case NRF_DRV_POWER_USB_EVT_DETECTED: // USB power detected.
        NRF_LOG_INFO("USB power detected");
        if (!nrf_drv_usbd_is_enabled()) // Check if driver is enabled.
        {
            nrf_drv_usbd_enable(); // Enable the driver if is not enabled yet.
        }
        break;

    case NRF_DRV_POWER_USB_EVT_REMOVED: // USB power removed from the connector.
        NRF_LOG_INFO("USB power removed");
        m_usbd_configured = false; // Flag used to indicate USB state.

        if (nrf_drv_usbd_is_started()) // Check if driver is started.
        {
            nrf_drv_usbd_stop(); // Stop USB functionality.
        }
        if (nrf_drv_usbd_is_enabled()) // Check if driver is enabled.
        {
            nrf_drv_usbd_disable(); // Disable e the driver if is not disabled yet.
        }
        /* Turn OFF LEDs */
        bsp_board_led_off(LED_USB_POWER);
        bsp_board_led_off(LED_RECEIVE);
        bsp_board_led_off(LED_SEND);
        break;

    case NRF_DRV_POWER_USB_EVT_READY: // USB power regulator ready.
    /*
     If we are here it means that NRF_DRV_POWER_USB_EVT_DETECTED event has already happened.
     So this means that USB driver has already been enabled. Now we have to start if it is
     not started yet.
    */
        NRF_LOG_INFO("USB ready");
        bsp_board_led_on(LED_USB_POWER);
        if (!nrf_drv_usbd_is_started()) // Check if driver is started.
        {
            nrf_drv_usbd_start(true); // Start driver if it is not started yet.
        }
        break;
    default:
        ASSERT(false);
    }
}
```

Gli eventi gestiti sono:

- NRF_DRV_POWER_USB_EVT_DETECTED: è stata rilevata l'alimentazione per il device USB;
- NRF_DRV_POWER_USB_EVT_REMOVED: l'alimentazione del device USB è stata rimossa;
- NRF_DRV_POWER_USB_EVT_READY: i regolatori di tensione utilizzati per fornire l'alimentazione alla periferica USB sono pronti. Tale evento segue sempre l'evento NRF_DRV_POWER_USB_EVT_DETECTED.

Gli eventi che finora abbiamo gestito non esaurisco la totalità di quelli che nel normale funzionamento del device possono capitare. Nella parte iniziale del codice abbiamo infatti definito un pulsante (BTN_SYSTEM_OFF) per lo spegnimento totale del dispositivo USB. Si evince dunque che ulteriori eventi che possono verificarsi sono la pressione ed il rilascio del pulsante. Questi devono essere opportunamente gestiti e ciò avviene mediante la definizione di un gestore di eventi specifico.

```
// BSP (Board Support Package) event handler
static void bsp_evt_handler(bsp_event_t evt)
{
    switch ((unsigned int)evt)
    {

    case BSP_EVENT_SYSOFF:
    // This event will be defined later. This is the event generated by pressing BTN_SYSTEM_OFF.
    {
        m_system_off_req = true; // A system off request is received.
        break;
    }

    default:
        return;
    }
}
```

In tal caso si gestisce il solo evento BSP_EVENT_SYOFF. Questo verrà definito in seguito e corrisponde all'evento di rilascio del pulsante BTN_SYSTEM_OFF.
Fatto questo, tutti gli eventi che possono occorrere nel normale funzionamento del programma sono stati gestiti. Gestire un evento significa specificare cosa deve accadere in risposta all'evento stesso.
Il codice prosegue con l'inizializzazione di alcuni moduli funzionali al funzionamento dell'USBD (USB Device).
Innanzitutto si inizializza il modulo "power" ed il modulo "clock".

```
static void init_power_clock(void)
{
    ret_code_t ret;
    /* Initializing power and clock */
    ret = nrf_drv_clock_init(); // Function for initializing the nrf_drv_clock module.
    APP_ERROR_CHECK(ret);
    ret = nrf_drv_power_init(NULL); // Initialize power module driver.
    APP_ERROR_CHECK(ret);
    nrf_drv_clock_hfclk_request(NULL); // Function for requesting high-accuracy source HFCLK (64 MHz).
    /*
     When the system requests one or more clocks from the HFCLK controller,
     the HFCLK controller will automatically provide them.
     If the system does not request any clocks provided by the HFCLK controller,
     the controller will enter a power saving mode.
    */
    while (!(nrf_drv_clock_hfclk_is_running()))
    {
        /* Just waiting */
    }
    ret = app_timer_init(); // Function for initializing the timer module.
    APP_ERROR_CHECK(ret);
    // Avoid warnings if assertion is disabled
    UNUSED_VARIABLE(ret);
}
```

La funzione "nrf_drv_clock_hfclk_request" è utilizzata per richiedere al controller HFCLK la sorgente ad alta precisione (HFXO). Questa è una sorgente di clock a 64MHz ottenuta mediante un oscillatore al cristallo esterno. La funzione nrf_drv_clock_hfclk_is_running() è chiamata per verificare in polling se la sorgente di clock richiesta è stata correttamente avviata.

Dopo aver inizializzato il modulo “power” ed il modulo “clock”, si passa all'inizializzazione del modulo BSP (Board Support Package). Quest'ultimo è un modulo che permette di fornire astrazioni e funzionalità utilizzando i kit di sviluppo Nordic Semiconductor.
Merita una menzione particolare l'istruzione “bsp_event_to_button_action_assign”.
Questa permette di associare ad un pulsante un determinato evento. Nel nostro caso è in questa occasione che si definisce l'evento “BSP_EVENT_SYSOFF” che abbiamo incontrato nel gestore di eventi del modulo BSP. I parametri della funzione “bsp_event_to_button_action_assign” sono in ordine:

- nome del pulsante;
- azione che genera l'evento (nel nostro caso il rilascio del pulsante);
- nome associato all'evento.

```
static void init_bsp(void)
{
    ret_code_t ret;
    ret = bsp_init(BSP_INIT_BUTTONS, bsp_evt_handler);
    APP_ERROR_CHECK(ret);
    ret = bsp_event_to_button_action_assign(BTN_SYSTEM_OFF,BSP_BUTTON_ACTION_RELEASE,BSP_EVENT_SYSOFF);
    // BSP_EVENT_SYSOFF is the event generated by releasing BTN_SYSTEM_OFF.
    APP_ERROR_CHECK(ret);
    // Avoid warnings if assertion is disabled
    UNUSED_VARIABLE(ret);
}
```

Fatto questo, i moduli propedeutici al funzionamento del firmware sono stati inizializzati. Successivamente, per comodità, si definisce la funzione “log_resetreason”. Può risultare comodo infatti, specie in fase di debug, conoscere il motivo che ha causato il reset dell'USBD.

```
static void log_resetreason(void)
{
    // Reset reason: reasons why a reset occurs
    uint32_t rr = nrf_power_resetreas_get(); // Function returns the reset reason mask
    NRF_LOG_INFO("Reset reason:");
    if (0 == rr)
    {
        NRF_LOG_INFO("- NONE");
    }
    if (0 != (rr & NRF_POWER_RESETREAS_RESETPIN_MASK)) // Reset from pin reset detected
    {
        NRF_LOG_INFO("- RESETPIN");
    }

    if (0 != (rr & NRF_POWER_RESETREAS_SREQ_MASK))
    // This is a software reset caused by software fault.
    {
        NRF_LOG_INFO("- SREQ");
    }
    if (0 != (rr & NRF_POWER_RESETREAS_LOCKUP_MASK)) // Reset from CPU lock-up detected.
    // Lockup is defined as the symptom of a function using the CPU and not releasing it for a period of time.
    {
        NRF_LOG_INFO("- LOCKUP");
    }
    if (0 != (rr & NRF_POWER_RESETREAS_OFF_MASK))
    // Reset due to wakeup from System OFF mode, when wakeup is triggered by DETECT signal from GPIO.
    {
        NRF_LOG_INFO("- OFF (Button system off request pressed)");
    }

}
```

Per comprendere la funzione è necessario preliminarmente introdurre il concetto di “maschera”. Supponiamo di volere per qualche motivo settare il MSB di un byte generico. E' utile in questo caso definire un byte ausiliario detto “maschera di bit” o semplicemente “maschera” che presenta tutti i bit pari a 0 eccetto il MSB pari a 1. Eseguendo l'OR bitwise tra il byte di partenza e la maschera si ottiene il byte di partenza con il MSB pari a 1.
Possiamo considerare anche un ulteriore esempio. Dato un byte generico vogliamo indagare se il MSB è pari ad 1. Per far questo basta mettere il byte di partenza in AND bitwise (&) con una maschera che presenta tutti i bit a 0 eccetto il MSB ad 1. In tal caso se il risultato dell'operazione è diverso da zero possiamo dedurre che il MSB del byte di partenza è ad 1, altrimenti non lo è.
Dopo aver descritto con alcuni esempi cosa si intende per “maschera” è possibile comprendere come nella funzione di cui sopra si indaga la natura del reset.
Innanzitutto si salva nella variabile rr (variabile a 32bit) il valore di ritorno della funzione nrf_power_resetreas_get(). Quest'ultima ritorna proprio la maschera relativa alla ragione del reset. In altri termini, la funzione nrf_power_resetreas_get() ritorna 32 bit ciascuno dei quali identifica un particolare motivo per il quale è avvenuto il reset. A questo punto si realizza l'operazione di AND bitwise tra il valore ritornato da nrf_power_resetreas_get() e 32bit tutti uguali a 0 eccetto uno che è uguale ad 1 e corrisponde al relativo motivo di reset che vogliamo indagare. Se il risultato dell'operazione è diverso da zero allora la ragione del reset è proprio quella corrispondente al bit impostato ad 1, altrimenti non lo è. E' questo il modus operandi utilizzato per l'identificazione del motivo del reset.
Fatto questo, entriamo nel main cioè il “cuore” del firmware.

```
int main(void)
{
    ret_code_t ret;
    static const app_usbd_config_t usbd_config = {.ev_state_proc = usbd_user_ev_handler};
    UNUSED_RETURN_VALUE(NRF_LOG_INIT(NULL)); // Macro for initializing the logs.
    NRF_LOG_DEFAULT_BACKENDS_INIT();
    NRF_LOG_INFO("LOG OK");
    init_power_clock(); // Function defined above to init power&clock.
    NRF_LOG_INFO("Power&Clock OK");
    init_bsp(); // Function for initializing BSP (Board Support Package).
    NRF_LOG_INFO("BSP OK");
```

Si definisce la struttura usbd_config di tipo app_usbd_config_t. Si riempie uno dei campi della struttura con il gestore di eventi relativo al device USB. Fare ciò è indispensabile perché tale configurazione deve essere passata alla funzione app_usbd_init() che serve per l'inizializzazione della libreria USB.
Fatto questo si inizializza:

- il modulo relativo ai LOG;
- il modulo “power” ed il modulo “clock” con la funzione init_power_clock() descritta in precedenza;
- il modulo BSP con la funzione init_bsp() precedentemente descritta.

Si utilizza poi la funzione app_usbd_serial_num_generate() per generare un numero seriale standard identificativo ed unico per ogni device. A questo punto, come detto in precedenza si passa alla funzione app_usbd_init() la struttura usbd_config di tipo app_usbd_config_t precedentemente definita e riempita con il gestore di eventi relativo al device USB.
Il passo successivo consiste nell'utilizzare la funzione precedentemente definita per la valutazione del motivo del reset. Una volta fatto questo si “ripulisce” il campo contenente la ragione del reset in modo tale da permettere la conoscenza della ragione dell'ultimo reset avvenuto.

```
app_usbd_serial_num_generate(); // Function used to generate default serial number.

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

NRF_LOG_INFO("USBD composite device started...");
log_resetreason(); // Function defined above that explain the reason of the reset.
nrf_power_resetreas_clear(nrf_power_resetreas_get()); // Function clears reset reason fields.
```

Veniamo ora ad una delle parti di codice più importante. Finora abbiamo definito la classe CDC ACM e la “myclass”. A questo punto tramite la funzione app_usbd_class_append() si aggiunge l'istanza della relativa classe. Questa funzione collega una determinata istanza a tutti gli endpoint richiesti.
Prima di fare questo è però necessario preliminarmente riempire la struttura di tipo app_usbd_class_inst_t. Questa conterrà le informazioni relative ai diversi endpoint e ai puntatori a funzione utilizzati da quella particolare classe USB.
Ciò consente di creare più facilmente device USB compositi che combinano cioè più classi.
Più concretamente, è necessario passare un puntatore all'istanza della classe ogni volta che si interagisce con la libreria USBD, per indicare alla libreria su quale particolare istanza si sta eseguendo un'azione. Questo spiega il motivo per cui alla funzione app_usbd_class_append() si passa un puntatore all'istanza della classe (class_cdc_acm per la classe CDC ACM e analogamente per myclass1 e myclass2).

```
// CDC ACM instance
app_usbd_class_inst_t const * class_cdc_acm = app_usbd_cdc_acm_class_inst_get(&m_app_cdc_acm);
ret=app_usbd_class_append(class_cdc_acm);
APP_ERROR_CHECK(ret);
NRF_LOG_INFO("CDC ACM OK");

// MY CLASS instance
app_usbd_class_inst_t const * myclass1 = app_usbd_myclass_inst_get(&m_app_myclass1);
ret=app_usbd_class_append(myclass1);
APP_ERROR_CHECK(ret);
NRF_LOG_INFO("MY CLASS 1 OK");

app_usbd_class_inst_t const * myclass2 = app_usbd_myclass_inst_get(&m_app_myclass2);
ret=app_usbd_class_append(myclass2);
APP_ERROR_CHECK(ret);
NRF_LOG_INFO("MY CLASS 2 OK");
```

Fatto questo si inizializzano i LEDs e si impone il loro stato iniziale.

```
// Configure LEDs
bsp_board_init(BSP_INIT_LEDS);
bsp_board_led_off(LED_RECEIVE);
bsp_board_led_off(LED_SEND);
bsp_board_led_on(LED_USB_STATUS);
```

Il codice prosegue con la distinzione di due casi:

- USBD_POWER_DETECTION = true
  In questo caso il rilevamento dell'alimentazione è abilitato. Questo significa che il modulo USB viene abilitato solo dopo che si è verificato l'evento NRF_DRV_POWER_USB_EVT_DETECTED. Per abilitare l'elaborazione degli eventi relativi all'alimentazione del modulo USB si utilizza la funzione nrf_drv_power_usb_evt_init(), la quale vuole come parametro d'ingresso l'indirizzo della struttura config di tipo nrf_drv_power_usbevt_config_t precedentemente riempita con il gestore di eventi power_usb_event_handler.

```
if (USBD_POWER_DETECTION) // defined as true at the beginning of the code.
{
    static const nrf_drv_power_usbevt_config_t config ={.handler = power_usb_event_handler};
    ret = nrf_drv_power_usbevt_init(&config);
    /*
      Initialize USB power event processing. Configures and setups the USB power event processing.
      Parameters:
      [in]  p_config              Configuration with values and event handler.
      Return values:

      NRF_ERROR_INVALID_STATE     This event cannot be initialized when SD is enabled and SD does not support
                                  USB power events.

      NRF_SUCCESS                 Successfully initialized and configured.

    */
    APP_ERROR_CHECK(ret);
}
```

- USBD_POWER_DETECTION = false
  In questo caso il rilevamento dell'alimentazione è disabilitato e ciò significa che indipendentemente dal verificarsi dell'evento NRF_DRV_POWER_USB_EVT_DETECTED il modulo USB è abilitato. L'abilitazione del modulo USB avviene dopo un tempo prestabilito pari a STARTUP_DELAY (100us).

```
else
{
    // If we are here it means that USBD_POWER_DETECTION = false.
    // In this case we have to start USB after a STARTUP_DELAY.
    NRF_LOG_INFO("No USB power detection enabled... Starting USB now");
    nrf_delay_us(STARTUP_DELAY);
    // STARTUP_DELAY is the number of microseconds to start USBD after powering up.
    if (!nrf_drv_usbd_is_enabled())
    {
        app_usbd_enable();
        APP_ERROR_CHECK(ret);
    }

    /* Wait for regulator power up */
    while (NRF_DRV_POWER_USB_STATE_CONNECTED == nrf_drv_power_usbstatus_get())
    {

    /*
      nrf_drv_power_usbstatus_get() return current USB power status.
      There are three differt return value:

      NRF_DRV_POWER_USB_STATE_DISCONNECTED        No power on USB lines detected

      NRF_DRV_POWER_USB_STATE_CONNECTED           The USB power is detected, but USB power regulator
                                                  is not ready.

      NRF_DRV_POWER_USB_STATE_READY               From the power point of view USB is ready for working

      If we are here it means that the USB power is detected, but USB power regulator is not ready.
      So this means that we just have to wait.
    */

                                /* Just waiting for the regulator */

    }
```

```
if (NRF_DRV_POWER_USB_STATE_READY == nrf_drv_power_usbstatus_get())
 // If we are here it means that from the energy point of view the USB is ready to work.
{
   if (!nrf_drv_usbd_is_started()) // Check if driver is started.
/*
 Return values:
 false     Driver is not started.
 true      Driver is started (fully functional).
 If we are here it means that driver is not started. In this case we have to start it.
*/
  {
  app_usbd_start();
 /*
  Start USB functionality.
  After calling this function USBD peripheral should be fully functional.
  All new incoming events / interrupts would be processed by the driver.
 */
  }
 }
else
{
 app_usbd_disable();
}
}
```

Il codice prosegue con la dichiarazione di alcune variabili. La loro utilità risulterà chiara in seguito.

```
int n=0;
ret_code_t ret1;
static char data1[] = {'c','i','a','o',' ','m','o','n','d','o',' ','d','a',' ','E','P','1'};
static char data2[] = {'c','i','a','o',' ','m','o','n','d','o',' ','d','a',' ','E','P','2'};
```

Fatto questo si entra nel loop infinito. La funzione app_usbd_event_queue_process() ritorna "true" se l'evento è stato processato. In tal caso non si deve far altro che aspettare che un nuovo evento si verifichi. Se la funzione ritorna "false" significa che c'è un evento ancora non processato e che quindi deve essere opportunamente gestito.
Nel seguito si distinguono tre diverse situazioni:

- flag = 1
  Come abbiamo già visto, il flag è uguale ad 1 se e soltanto se il contenuto del buffer di ricezione relativo all'istanza di classe CDC ACM è "s". Ciò significa in altri termini che l'host ha trasmesso sull'endpoint OUT (0x03) relativo all'istanza di classe CDC ACM il carattere "s". In risposta a questo evento il device "risponde" all'host trasmettendo sull'endpoint IN (0x83). In particolare mediante l'istruzione app_usbd_cdc_acm_write() è possibile scrivere sulla porta CDC ACM. Preliminarmente è però necessario utilizzare la funzione sprintf(). Il prototipo di tale funzione è il seguente:
  int sprintf(char *str, const char *format, ...);
  La funzione sprintf() permette di scrivere i caratteri nell'area puntata da str.
  Naturalmente str deve essere ampia a sufficienza per contenere i caratteri in output più il terminatore '\0'. La funzione ritorna il numero di caratteri scritti oppure un numero negativo in caso di errore. Nel nostro caso la scrittura dei caratteri avviene sul buffer di trasmissione relativo all'istanza di classe CDC ACM. Solo ora è possibile utilizzare la funzione app_usbd_cdc_acm_write().

- flag = 2
  Come abbiamo già visto, il flag è uguale a 2 se e soltanto se il contenuto del buffer di ricezione relativo all'istanza di classe CDC ACM è “b”. Ciò significa in altri termini che l'host ha trasmesso sull'endpoint OUT (0x03) relativo all'istanza di classe CDC ACM il carattere “b”. In risposta a questo evento il device “risponde” all'host trasmettendo sull'endpoint IN (0x81) associato all'interfaccia 0 relativa all'istanza di classe “myclass”. Il funzionamento è quindi cross-interface. I dati viaggiano da host a device tramite l'endpoint OUT dell'interfaccia DATA relativa all'istanza di classe CDC ACM e nel verso opposto tramite l'endpoint IN dell'interfaccia 0 relativa all'istanza myclass1.

- flag = 3
  L' host ha trasmesso sull'endpoint OUT (0x03) relativo all'istanza di classe CDC ACM il carattere “n”. Il device “risponde” all'host trasmettendo sull'endpoint IN (0x82) associato all'interfaccia 3 relativa all'istanza di classe “myclass”.
  Anche in questo caso quindi il funzionamento è cross-interface. I dati viaggiano da host a device tramite l'endpoint OUT dell'interfaccia DATA relativa all'istanza di classe CDC ACM e nel verso opposto tramite l'endpoint IN dell'interfaccia 3 relativa all'istanza myclass2.

```
while (true)  // Endless loop
{

   while (app_usbd_event_queue_process()) // Function that process events from the queue.
    {
    /*
     If we are here it means that app_usbd_event_queue_process() returns TRUE and this means
     that the event has been processed. We have to wait for a new event to be processed and
     in the meantime we are waiting.
    */
    }
   if (flag==1)
   {
    size_t size= sprintf(m_tx_buffer,"\r\nSystem status characteristic (request number:%d): ",n);
    ret1= (app_usbd_cdc_acm_write(&m_app_cdc_acm, m_tx_buffer,size));
    flag=0;
    if (ret1 == NRF_SUCCESS)
     {
      n++;
     }
   }
    // Scrittura di prova su EPIN1
   if (flag==2)
   {
    static const nrf_drv_usbd_transfer_t transfer=
    {.p_data = {.tx= &data1},.size=sizeof(data1)};
    UNUSED_RETURN_VALUE(app_usbd_ep_transfer(NRF_DRV_USBD_EPIN1, &transfer));
    flag=0;
   }
    // Scrittura di prova su EPIN2
    if (flag==3)
   {
    static const nrf_drv_usbd_transfer_t transfer=
    {.p_data = {.tx= &data2},.size=sizeof(data2)};
    UNUSED_RETURN_VALUE(app_usbd_ep_transfer(NRF_DRV_USBD_EPIN2, &transfer));
    flag=0;
   }
```

Il codice prosegue con la gestione dello spegnimento del sistema. In precedenza è stato definito l'evento BSP_EVENT_SYSOFF corrispondente al rilascio del pulsante BTN_SYSTEM_OFF. Tale occorrenza è gestita in bsp_evt_handler dove per segnalare il pervenire della richiesta di

spegnimento del sistema si pone il flag m_system_off_req = true. Nel codice che segue si testa il valore assunto da tale flag, se "true" allora si deve garantire lo spegnimento del sistema.

```
 if (m_system_off_req)
 // This flag is used in button event processing and marks the fact that system OFF should be activated.
{
 NRF_LOG_INFO("Going to system OFF due to pressing the shutdown button");
 NRF_LOG_FLUSH(); // Macro for processing all log entries from the buffer.
 // It blocks until all buffered entries are processed by the backend.
 bsp_board_leds_off();
 nrf_power_system_off();
// The function above puts the CPU into system off mode.
}
```

Il codice si conclude con la gestione della modalità di risparmio energetico.
Come abbiamo già detto, quando non viene rilevata alcuna attività sul bus per un tempo superiore a 3ms viene generato l'evento APP_USBD_EVT_DRV_SUSPEND. Questo è gestito in usbd_user_ev_handler dove si pone il flag m_usbd_suspend_state_req = true.
Il flag m_usbd_suspended è invece utilizzato per valutare lo stato del device USB.
Se i due flag hanno valore diverso occorre distinguere due casi:

- è pervenuta una richiesta di sospensione e il device non è ancora in modalità di risparmio energetico e quindi si deve provvedere a fare entrare il device in tale modalità.
- il device è in sospensione ma m_usbd_suspend_state_req = false in quanto si è verificata la condizione di "resuming". Quest'ultima si genera quando l'host ricomincia a trasmettere dati sul bus. In quest'ultimo caso si deve garantire che il device venga riportato nella normale situazione di funzionamento.

```
    if (m_usbd_suspended != m_usbd_suspend_state_req)
    // We have to process the suspend request if USBD is not suspended yet.
    /*
       m_usbd_suspend_state_req:
       flag used to mark the fact that system OFF
       should be activated from main loop.

       m_usbd_suspended:
       flag used to mark the fact that USB suspension is detected.
    */
    {
        if (m_usbd_suspend_state_req)
        {
            // If we are here it means that there is a suspend request
            // but USB is not in suspend state so we have to accept the request.
            m_usbd_suspended = nrf_drv_usbd_suspend();

        /*
            Return values
            true             USBD peripheral successfully suspended.
            false            USBD peripheral was not suspended due to resume detection.
        */
        }
        else
        {
            // If we are here it means USBD peripheral was not suspended due to resume detection.
            m_usbd_suspended = false;

        }
    }

  __SEV(); // Set Event Flag
  __WFE(); // Wait For Event
  __WFE();
 }
}
```

3 - Il progetto: firmware host-side

3.1 - Libreria LIBUSB

La libreria libusb fornisce l'accesso generico ai dispositivi USB. Questa facilita notevolmente la produzione di applicazioni che comunicano con l'hardware USB. Tra le altre cose, la libreria permette di cercare un dispositivo in base al vendorID (VID) e al productID (PID) o in base alla classe. Nel nostro caso il device appartiene ad una classe customizzata (vendor specific) per cui la ricerca del device avviene lato host tramite VID e PID (specificati in "sdk_config.h" nella porzione di codice mostrata qui sotto).

```
// <i> Note: This value is not editable in Configuration Wizard.
// <i> Vendor ID ordered from USB IF: http://www.usb.org/developers/vendor/
#ifndef APP_USBD_VID
#define APP_USBD_VID 0x1915
#endif

// <s> APP_USBD_PID - Product ID.

// <i> Note: This value is not editable in Configuration Wizard.
// <i> Selected Product ID
#ifndef APP_USBD_PID
#define APP_USBD_PID 0x520F
#endif
```

3.2 - La strategia

Esistono due diverse filosofie per realizzare un programma per accedere ad un dispositivo che tipicamente è più lento della CPU dove risiede il programma stesso:

- interfaccia I/O sincrona: dall'invio del comando all'esecuzione dello stesso il programma è "congelato" (blocking API). L'interfaccia I/O sincrona consente di eseguire un trasferimento USB con una singola chiamata a funzione. Quando la chiamata alla funzione ritorna, il trasferimento è completato ed è possibile analizzare i risultati. Il vantaggio principale di questo modello è la semplicità, tuttavia, questa interfaccia ha i suoi limiti. Infatti, l'applicazione è congelata all'interno di libusb_bulk_transfer() fino al completamento della transazione e resterà inattiva nel frattempo. Un altro problema è che legando il thread con quella singola transazione non è possibile eseguire operazioni di I/O con più endpoint e/o più dispositivi contemporaneamente, a meno che non si ricorra alla creazione di un thread per transazione. Inoltre, non è possibile annullare il trasferimento dopo che la richiesta è stata inviata.
- interfaccia I/O asincrona: dall'invio del comando a quando viene eseguito, l'host può eseguire altre operazioni (non-blocking). E' tipicamente più complicata rispetto l'interfaccia sincrona in quanto si deve tenere traccia dell'ordine dei comandi dati. Sebbene sia un'interfaccia più complessa, risolve tutti i problemi descritti sopra per l'interfaccia sincrona. In questo caso, piuttosto che utilizzare funzioni che si bloccano fino al completamento dell'operazione di I/O, l'interfaccia asincrona presenta funzioni non bloccanti che iniziano un trasferimento e ritornano immediatamente. L'applicazione passa alla funzione non bloccante un puntatore alla funzione di callback che libusb chiamerà con i risultati della transazione

quando è stata completata. I trasferimenti che sono stati inviati tramite le funzioni non bloccanti possono essere annullati con una chiamata alla funzione libusb_cancel_transfer(). La natura non bloccante di questa interfaccia consente di eseguire simultaneamente I/O su più endpoint e/o su più dispositivi. Proprio per quest'ultima considerazione, nel presente lavoro di tesi si è scelto di utilizzare un interfaccia I/O di questo tipo.

3.3 - Il codice

Nella prima parte del codice si definiscono alcune sigle che verranno poi richiamate nel prosieguo. E' molto importante sottolineare la definizione del contesto. “libusb_context” è una struttura che rappresenta una sessione libusb. E' possibile definire sessioni multiple ciascuna delle quali associata ad un modulo che utilizza indipendentemente la libreria. Specificando la sessione si prevengono interferenze tra due diversi “utenti” libusb. Per applicazioni in cui esiste un solo “utente” libusb è legale passare NULL a tutte le funzioni che richiedono un puntatore di contesto purché si sia sicuri che nessun altro codice tenterà di utilizzare libusb dallo stesso processo. Quando si passa NULL, verrà utilizzato il contesto predefinito. Questo viene creato la prima volta che un processo chiama libusb_init() e termina con libusb_exit().
Si definisce inoltre “devh”, un puntatore ad una struttura tipo “libusb_device_handle”. Questo viene essenzialmente utilizzato per gestire le operazioni di I/O.

```
#include <signal.h>
#include <stdio.h>
#include <poll.h>
#include <libusb-1.0/libusb.h>


#define USB_VENDOR_ID       0x1915      /* USB vendor ID used by the device */
#define USB_PRODUCT_ID      0x520F      /* USB product ID used by the device */
#define USB_ENDPOINT_IN1    (LIBUSB_ENDPOINT_IN  | 1)   /* endpoint address */
#define USB_ENDPOINT_OUT1   (LIBUSB_ENDPOINT_OUT | 1)   /* endpoint address */
#define USB_ENDPOINT_IN2    (LIBUSB_ENDPOINT_IN  | 2)   /* endpoint address */
#define USB_ENDPOINT_OUT2   (LIBUSB_ENDPOINT_OUT | 2)   /* endpoint address */

static libusb_context *ctx = NULL;
struct libusb_device_handle *devh = NULL;
```

Il codice prosegue con la dichiarazione di alcune variabili globali la cui utilità risulterà chiara nel seguito. Si definisce inoltre il prototipo della funzione sighandler. Questa permette di uscire dall'applicazione alla pressione di “^C” da tastiera.

```
#define LEN_IN_BUFFER 1024*8

static uint8_t in_buffer1[LEN_IN_BUFFER];
static uint8_t in_buffer2[LEN_IN_BUFFER];

// OUT-going transfers (OUT from host PC to USB-device)
struct libusb_transfer *transfer_out = NULL;

// IN-coming transfers (IN to host PC from USB-device)
struct libusb_transfer *transfer_in1 = NULL;
struct libusb_transfer *transfer_in2 = NULL;

int do_exit = 0;

// Function Prototypes:
void sighandler(int signum);

enum {
    out_deinit,
    out_release,
    out
} exitflag;
```

Prima di proseguire con la restante parte di codice è preliminarmente necessario analizzare gli step di una transfer asincrona in accordo con la libreria libusb.
Un'operazione di I/O asincrona può essere decomposta in cinque passi fondamentali:

1. Allocazione: questo step prevede l'allocazione della memoria per il trasferimento USB (transfer). In questa fase la transfer è "vuota" cioè senza dettagli sul tipo di I/O per la quale verrà utilizzata. L'allocazione viene eseguita con la funzione libusb_alloc_transfer().
2. Riempimento: questo step prevede il riempimento della transfer con le informazioni relative al tipo e alla direzione del messaggio, il buffer dei dati, la funzione di callback, ecc. Per fare questo si possono utilizzare le funzioni di supporto: libusb_fill_control_transfer(), libusb_fill_bulk_transfer() e libusb_fill_interrupt_transfer().
3. Sottomissione: dopo la fase di allocazione e di riempimento la transfer può essere sottomessa usando la funzione libusb_submit_transfer(). Questa ritorna immediatamente e può essere considerata come attivazione della richiesta di I/O in background.
4. Gestione del completamento: dopo che un trasferimento è stato inviato si può incorrere in una delle seguenti situazioni:
    - il trasferimento viene completato (ovvero alcuni dati sono stati trasferiti);
    - il trasferimento ha un timeout e il timeout scade prima che tutti i dati vengano trasferiti;
    - il trasferimento non riesce a causa di un errore;
    - il trasferimento viene annullato.

   Ciascuna di queste causerà il richiamo della funzione di callback di trasferimento specificata dall'utente. Spetta alla funzione di callback determinare quale di quanto sopra è effettivamente accaduto e agire di conseguenza. Al callback specificato dall'utente viene passato un puntatore alla struttura libusb_transfer che è stata utilizzata per impostare e inviare il trasferimento.
5. Deallocazione: quando un trasferimento è stato completato (cioè la funzione di callback è stata invocata), è consigliato liberare il trasferimento. I trasferimenti vengono deallocati con libusb_free_transfer(). Lo step di deallocazione è consigliato se non deve essere rinviata una nuova transfer. In caso contrario, piuttosto che la deallocazione, lo step da compiere è la risottomissione. Allocazione, riempimento e invio sono tutti separati anche se ragionevolmente potrebbero essere combinati in un'unica operazione. Il motivo della separazione è consentire di inviare nuovamente i trasferimenti senza doverli riallocare ogni volta.

Analizzati gli step di una transfer asincrona si è in grado di illustrare il prosieguo del codice.
Questo seguita con la definizione delle due funzioni di callback: cb_out e cb_in.
La prima è quella di OUT callback e verrà richiamata quando la trasmissione dati da host a device è completata.

```
// Out Callback
//   - This is called after the Out transfer has been received by libusb
void cb_out(struct libusb_transfer *transfer)
{
    fprintf(stderr, "cb_out: status =%d, actual_length=%d\n",
        transfer->status, transfer->actual_length);
}
```

Fatto questo si passa alla definizione della funzione di IN callback (cb_in). Questa verrà richiamata quando la trasmissione dati da device ad host è completata.
E' all'interno di questa funzione che viene eseguita la risottomissione della transfer avvenuta sull'endpoint effettivamente utilizzato per la transfer precedente.

```
// In Callback
//   - This is called after the command for version is processed.
//     That is, the data for in_buffer IS AVAILABLE.
void cb_in(struct libusb_transfer *transfer)
{

    int len = transfer->actual_length;
    printf("\nReceived data from EP %d:", (int) transfer->endpoint);
    for (int i = 0; i < len; ++i)
        printf("%c", transfer->buffer[i]);
    printf("\n");
    //submit the next transfer
    if  ( (int) transfer->endpoint == USB_ENDPOINT_IN1) libusb_submit_transfer(transfer_in1);
    else libusb_submit_transfer(transfer_in2);

}
```

Il codice prosegue con il main(). Inizialmente viene definita una struttura di tipo sigaction che verrà riempita nel seguito e sarà utilizzata per captare la pressione di “^C” da tastiera.
Successivamente viene utilizzata la funzione libusb_init() per inizializzare la libreria e per creare il contesto. Questa deve essere chiamata all'inizio del programma, prima che tutte le funzioni libusb vengano utilizzate.

```
int main(void)
{
    struct sigaction sigact;
    int r = 1;  // result
    int i;

    //init libUSB
    r = libusb_init(NULL);

    if (r < 0) {
        fprintf(stderr, "Failed to initialise libusb\n");
        return 1;
    }
```

Il codice prosegue con l'apertura del device. Questo avviene mediante la funzione libusb_open_device_with_vid_pid(). Questa è utilizzata per trovare un device in accordo con una particolare combinazione di vendorID (VID) e productID (PID). La funzione ritorna un device handle per il primo device trovato o NULL se non viene trovato alcun device.

```
    //open the device with vendorID (VID) e productID (PID)
    devh = libusb_open_device_with_vid_pid(ctx,
        USB_VENDOR_ID, USB_PRODUCT_ID);
    if (!devh) {
        perror("device not found");
        return 1;
    }
```

Lo step successivo consiste nel richiedere le due interfacce, l'interfaccia 0 e l'interfaccia 3 a ciascuna delle quali corrisponde una coppia di bulk endpoint.
Prima di fare questo è però necessario utilizzare la funzione libusb_detach_kernel_driver().
Questa serve per fare in modo che quando l'interfaccia viene richiesta non sia stata già reclamata da un driver del kernel.

```
    libusb_detach_kernel_driver (devh,0); // Interface 0
    libusb_detach_kernel_driver (devh,3); // Interface 3
```

E' possibile ora richiedere le due interfacce mediante l'istruzione libusb_claim_interface().

Questa ritorna 0 nel caso di successo o un valore negativo in caso di errore.
Gli errori possono essere di diverso tipo, tra questi:

- LIBUSB_ERROR_NOT_FOUND (-5): nel caso in cui l'interfaccia non esistesse;
- LIBUSB_ERROR_BUSY (-6): se il programma o un driver ha già richiesto l'interfaccia. E' questo l'errore a cui si potrebbe incorrere se non si utilizzasse l'istruzione libusb_detach_kernel_driver();
- LIBUSB_ERROR_NO_DEVICE (-4): se il device è stato disconnesso;

Nel caso in esame, al verificarsi di un errore si impostano due flag, uno serve per identificare il motivo per il quale è avvenuto l'errore (exitflag), l'altro per identificare che si è verificato un errore per il quale l'applicazione non può proseguire e quindi deve terminare (do_exit).
Se non si verifica alcun errore e quindi il valore restituito da libusb_claim_interface() è zero allora si procede con gli step che permettono di implementare una transfer asincrona (allocazione, riempimento e sottomissione).

```
//claim the interface
r = libusb_claim_interface(devh, 0);
if (r < 0) {
    fprintf(stderr, "usb_claim_interface 0 error %d\n", r);
    exitflag = out;
    do_exit = 1;
} else  {
    printf("Claimed interface 0\n");
    // allocate transfer of data IN (IN to host PC from USB-device)
    transfer_in1  = libusb_alloc_transfer(0);
    libusb_fill_bulk_transfer( transfer_in1, devh, USB_ENDPOINT_IN1,
        in_buffer1,  LEN_IN_BUFFER1,  // Note: in_buffer is where input data written.
            cb_in, NULL, 0); // no user data
    //submit the transfer, all following transfers are initiated from the CB
    r = libusb_submit_transfer(transfer_in1);
  }
```

Quanto fatto sopra è relativo all'interfaccia 0 e va ovviamente ripetuto anche per l'interfaccia 3.

```
r = libusb_claim_interface(devh, 3);
if (r < 0) {
    fprintf(stderr, "usb_claim_interface 0 error %d\n", r);
    exitflag = out;
    do_exit = 1;
} else  {
    printf("Claimed interface 3\n");
    // allocate transfer of data IN (IN to host PC from USB-device)
    transfer_in2  = libusb_alloc_transfer(0);
    libusb_fill_bulk_transfer( transfer_in2, devh, USB_ENDPOINT_IN2,
        in_buffer2,  LEN_IN_BUFFER2,  // Note: in_buffer is where input data written.
            cb_in, NULL, 0); // no user data
    //submit the transfer, all following transfers are initiated from the CB
    r = libusb_submit_transfer(transfer_in2);
  }
```

A questo punto viene riempita la struttura sigact di tipo sigaction precedentemente definita all'inizio del main().

```
// Define signal handler to catch system generated signals
// (If user hits CTRL+C, this will deal with it.)
sigact.sa_handler = sighandler;  // sighandler is defined below. It just sets do_exit.
sigemptyset(&sigact.sa_mask);
sigact.sa_flags = 0;
sigaction(SIGINT, &sigact, NULL);
sigaction(SIGTERM, &sigact, NULL);
sigaction(SIGQUIT, &sigact, NULL);
```

All'inizio del codice viene incluso il file header della libreria standard signal.h. Questo consente l'utilizzo delle funzioni per la gestione dei segnali fra processi. Un processo può ricevere un segnale sia al verificarsi di un evento hardware che software e può intercettare e gestire un segnale tramite la funzione sigaction(). Questa permette di definire l'azione corrispondente all'occorrenza di un particolare segnale come ad esempio SIGINT (pressione “^C”). Nel caso in esame la gestione dei segnali SIGINT, SIGTERM e SIGQUIT passa alla funzione sighandler definita al termine del main(). Questa si limita ad impostare il flag do_exit ad 1.
Si entra ora nella parte nevralgica del codice. Questa è essenzialmente basata sull'utilizzo della funzione poll() che monitora un set di file descriptors e attende che un evento si verifichi su uno di questi. Il prototipo della funzione poll() è:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Il set di file descriptors che la poll() deve monitorare è specificato nel parametro fds che ha il seguente formato:

```
struct pollfd
{
    int fd; /* descrittore di file */
    short events; /* eventi richiesti */
    short revents; /* eventi restituiti */
};
```

Il campo events è un parametro di input, una maschera di bit che specifica gli eventi a cui l'applicazione è interessata per il file descriptor fd.
Il campo revents è un parametro di output, riempito dal kernel con gli eventi realmente accaduti.
Il chiamante deve specificare il numero di elementi nell'array fds in nfds.
L'argomento timeout specifica il numero di millisecondi per cui poll() dovrebbe bloccarsi in attesa che un file descriptor sia pronto. Nel caso in esame alla poll() si passa il timeout che viene ritornato dalla funzione libusb_get_next_timeout(), dopo aver fatto alcune operazioni per ottenerlo in millisecondi.
In caso di successo, poll() restituisce un valore non negativo corrispondente al numero di elementi nella struttura pollfd i cui campi revents sono stati impostati ad un valore diverso da zero.
La funzione che la libreria libusb mette a disposizione per recuperare un elenco di file descriptors da interrogare nel main loop è la libusb_get_pollfds(). Questa restituisce un elenco terminato da NULL in caso di successo o NULL in caso di errore. La libusb_get_pollfds() ritorna strutture di tipo libusb_pollfd (che non ha il campo revents) mentre la poll() vuole per argomento una struttura di tipo pollfd (descritta sopra nel dettaglio). Per questo motivo, per utilizzare la funzione poll() è necessario copiare il contenuto della struttura di tipo libusb_pollfd in una struttura di tipo pollfd. Questo è quello che si fa nel ciclo for mostrato nella porzione di codice seguente.

```
    printf("Entering loop to process callbacks...\n");
    const struct libusb_pollfd** usbfd= libusb_get_pollfds(ctx);
    struct pollfd myfds[3];
    int k;
    static struct timeval tv;
    tv.tv_sec=0;
    tv.tv_usec=0;
    int timeout_ms=0;
    const struct libusb_pollfd** usbfd= libusb_get_pollfds(ctx);
    for  (k = 0; k < 3; ++k)
    {
        if(usbfd[i] == NULL) break;
        printf("\nvalore di k:%d",k);
        myfds[k].fd = usbfd[k]->fd;
        myfds[k].events = usbfd[k]->events;
        myfds[k].revents = 0;
    }
```

Dopo aver fatto quanto sopra descritto è possibile utilizzare la funzione poll() come illustrato nel codice seguente. Come detto in precedenza, questa restituisce un valore non negativo corrispondente al numero di elementi nella struttura pollfd i cui campi revents sono stati impostati ad un valore diverso da zero. Per questo motivo si controlla il valore ritornato dalla poll() e se questo è positivo si chiama la funzione libusb_handle_events_timeout(). Questa gestisce ogni evento pendente controllando se il timeout è scaduto e l'attività dei file descriptors. Fatto questo si chiama la funzione libusb_handle_events_completed() che blocca il programma fino a quando l'evento pendente non risulta opportunamente gestito dalla funzione di callback.

```
if(1){
    while (!do_exit)
    {
      int res=libusb_get_next_timeout(ctx,&tv);
      if (res==1)
      {
         timeout_ms= (tv.tv_sec*1000)+ (tv.tv_usec/1000);
         if (tv.tv_usec % 1000) timeout_ms++;
      }else if (res==0) printf("\nNo pending timeout...");
      int retvalpoll=poll(myfds,k,timeout_ms);
      if (retvalpoll>0) libusb_handle_events_timeout(ctx,&tv);
      r =  libusb_handle_events_completed(ctx, NULL);
      if (r < 0)
      {   // negative values are errors
            exitflag = out_deinit;
            break;
      }
    }
}
 else{
      // This implementation uses a blocking call and aquires a lock to the event handler
      struct timeval timeout;
      timeout.tv_sec   = 0;        // seconds
      timeout.tv_usec = 100000;  // ( .1 sec)
      libusb_lock_events(ctx);
      while (!do_exit) {
           r = libusb_handle_events_locked(ctx, &timeout);
           if (r < 0){    // negative values are errors
                exitflag = out_deinit;
                break;
           }
      }
      libusb_unlock_events(ctx);
 }
```

Il main() prosegue con la gestione della cancellazione delle transfer.
L'istruzione condizionale if interpreta come true un qualunque valore diverso da 0 (o NULL nel caso di puntatori). “transfer_out” è un puntatore ad una struttura di tipo libusb_transfer ed è messo a NULL all'inizio del codice e poi successivamente viene allocato. Dato che la parte finale di terminazione viene eseguita anche se c'è stato un errore nell'aprire la porta USB e quindi quando il trasferimento non è stato ancora mai allocato, il controllo con l'if serve per evitare di cancellare un trasferimento mai allocato. Rimuovendo l'if si avrebbe la cancellazione della transfer indipendentemente se questa sia stata o meno allocata. La cancellazione di una transfer precedentemente non allocata restituisce l'errore LIBUSB_ERROR_NOT_FOUND.
L'utilizzo dell'istruzione condizionale if serve per evitare tutto ciò.
Quanto detto con riferimento alla cancellazione della transfer_out può essere esteso al caso della transfer_in1 e della transfer_in2.
La cancellazione di una transfer avviene mediante l'istruzione libusb_cancel_transfer().
Questa ritorna zero in caso di cancellazione completata con successo.

```
    // If these transfers did not complete then we cancel them.
    // Unsure if this is correct...
    if (transfer_out) {
        r = libusb_cancel_transfer(transfer_out);
        if (0 == r){
            printf("transfer_out successfully cancelled\n");
        }
        if (r < 0){
            exitflag = out_deinit;
        }

    }
    if (transfer_in1) {
        r = libusb_cancel_transfer(transfer_in1);
        if (0 == r){
            printf("transfer_in1 successfully cancelled\n");
        }
        if (r < 0){
            exitflag = out_deinit;
        }
    }
    if (transfer_in2) {
        r = libusb_cancel_transfer(transfer_in2);
        if (0 == r){
            printf("transfer_in2 successfully cancelled\n");
        }
        if (r < 0){
            exitflag = out_deinit;
        }
    }
```

Il main() prosegue e si conclude con il controllo del flag exitflag.

```
    switch(exitflag){
    case out_deinit:
        printf("at out_deinit\n");
        libusb_free_transfer(transfer_out);
        libusb_free_transfer(transfer_in1);
        libusb_free_transfer(transfer_in2);
    case out_release:
        libusb_release_interface(devh, 0);
        libusb_release_interface(devh, 3);
    case out:
        libusb_close(devh);
        libusb_exit(NULL);
    }
    return 0;
}
```

Terminato il main(), il codice si conclude con la funzione sighandler() che come detto in precedenza, quando richiamata, si limita ad imporre il flag do_exit=1.

```
// This will catch user initiated CTRL+C type events and allow the program to exit
void sighandler(int signum)
{
    printf("sighandler: EXIT \n");
    do_exit = 1;
}
```

4 - Il funzionamento

Per verificare la corretta enumerazione del device, da terminale è possibile utilizzare il comando:

```
nicholas@nicholas-VirtualBox:~$ lsusb -v -d 1915:520f
```

L'opzione “-v” sta per “verbosity” e viene utilizzata per visualizzare l'output in modo dettagliato.
L'opzione “-d [VID] [PID]” serve per mostrare solo il dispositivo avente quella particolare combinazione di productID e vendorID. Entrambi gli ID sono forniti in esadecimale.
Il risultato del comando di cui sopra è illustrato qui di seguito:

```
nicholas@nicholas-VirtualBox:~$ lsusb -v -d 1915:520f

Bus 002 Device 003: ID 1915:520f Nordic Semiconductor ASA nRF52 USB composite device

Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        64
  idVendor           0x1915 Nordic Semiconductor ASA
  idProduct          0x520f
  bcdDevice            1.00
  iManufacturer           1
  iProduct                2
  iSerial                 3
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength       0x0079
    bNumInterfaces          4
    bConfigurationValue     1
    iConfiguration          4
    bmAttributes         0xc0
      Self Powered
    MaxPower              100mA
    Interface Association:
      bLength                 8
      bDescriptorType        11
      bFirstInterface         1
      bInterfaceCount         2
      bFunctionClass          2 Communications
      bFunctionSubClass       2 Abstract (modem)
      bFunctionProtocol       0
      iFunction               0
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           2
      bInterfaceClass       255 Vendor Specific Class
      bInterfaceSubClass      0
      bInterfaceProtocol      0
      iInterface              0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
```

```
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x01  EP 1 OUT
      bmAttributes            2
        Transfer Type            Bulk
        Synch Type               None
        Usage Type               Data
      wMaxPacketSize     0x0040  1x 64 bytes
      bInterval               0
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       0
    bNumEndpoints           1
    bInterfaceClass         2 Communications
    bInterfaceSubClass      2 Abstract (modem)
    bInterfaceProtocol      0
    iInterface              0
    CDC Header:
      bcdCDC               1.10
    CDC Call Management:
      bmCapabilities       0x03
        call management
        use DataInterface
      bDataInterface          2
    CDC ACM:
      bmCapabilities       0x02
        line coding and serial state
    CDC Union:
      bMasterInterface        1
      bSlaveInterface         2
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x84  EP 4 IN
      bmAttributes            3
        Transfer Type            Interrupt
        Synch Type               None
        Usage Type               Data
      wMaxPacketSize     0x0040  1x 64 bytes
      bInterval              16
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        2
    bAlternateSetting       0
    bNumEndpoints           2
    bInterfaceClass        10 CDC Data
    bInterfaceSubClass      0
    bInterfaceProtocol      0
    iInterface              0
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x83  EP 3 IN
      bmAttributes            2
        Transfer Type            Bulk
        Synch Type               None
        Usage Type               Data
      wMaxPacketSize     0x0040  1x 64 bytes
```

```
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x03  EP 3 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        3
      bAlternateSetting       0
      bNumEndpoints           2
      bInterfaceClass       255 Vendor Specific Class
      bInterfaceSubClass      0
      bInterfaceProtocol      0
      iInterface              0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x82  EP 2 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x02  EP 2 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
```

Dopo aver verificato la corretta enumerazione del device si apre la porta seriale CDC ACM utilizzando picocom (da terminale: $ picocom /dev/ttyACM0). Come suggerisce il nome, picocom è un programma di emulazione di terminale analogo per funzionamento a PuTTy o Teraterm. Nell'immagine qui di seguito notiamo nella parte sinistra il terminale aperto con picocom e nella parte destra l'output del programma host-side.

```
nicholas@nicholas-VirtualBox: ~

local echo is  : no
noinit is      : no
noreset is     : no
hangup is      : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        :
omap is        :
emap is        : crcrlf,delbs,
logfile is     : none
initstring     : none
exit_after is  : not set
exit is        : no

Type [C-a] [C-h] to see available commands
Terminal ready

*** local echo: yes ***
s
System status characteristic (request number:0):
b
n
```

generato in seguito alla pressione di "s"

```
nicholas@nicholas-VirtualBox: ~

nicholas@nicholas-VirtualBox:~$ sudo ./host_async_twointv2
[sudo] password di nicholas:
Claimed interface 0
Claimed interface 3
Entering loop to process callbacks...

No pending timeout...
Received data from EP 129:ciao mondo da EP1

No pending timeout...
Received data from EP 130:ciao mondo da EP2
```

generato in seguito alla pressione di "b"

generato in seguito alla pressione di "n"

Il risultato è esattamente quello aspettato stando al programma host-side e a quello device-side.

Per valutare nel dettaglio la natura dei pacchetti sul bus USB è possibile utilizzare Wireshark. Questo è un programma open source per l'USB Sniffing cioè per l'analisi dei pacchetti trasmessi sul bus. In questo modo è possibile valutare in maniera dettagliata i pacchetti che viaggiano sul bus durante la fase di enumerazione del device ed i pacchetti dati che host e device si scambiano ad enumerazione avvenuta.

5 – Conclusioni e sviluppi futuri

Il presente lavoro ha portato ad approfondire i fondamenti del protocollo USB, a partire da nozioni prettamente teoriche fino ad arrivare all'implementazione di un device composito.
La parte iniziale della tesi si concentra sulle nozioni teoriche principali relative al protocollo, tra cui le principali revisioni, i componenti del bus USB, le specifiche elettriche, meccaniche, di alimentazione, il formato dati e la periferica USB in generale.
Il lavoro di tesi prosegue e si completa con la realizzazione di un device composito che supporta un'istanza di classe CDC ACM e due diverse istanze di classe personalizzata.
La scelta di realizzare una classe ad hoc piuttosto che utilizzarne una già implementata e messa a disposizione dall'USB-IF si è dimostrata appropriata in quanto permette una più facile gestione del device host-side.
Se da un lato la classe CDC ACM è supportata in ambiente Linux e Windows 10, dall'altro la classe personalizzata essendo realizzata ad hoc per lo scopo dovrà essere gestita in maniera opportuna host-side.
Parte integrate del lavoro di tesi è quindi lo studio della libreria standard C libusb e delle funzioni principali che questa mette a disposizione. Attraverso queste infatti si è resa possibile la scrittura di un firmware host-side per la gestione delle istanze di classe personalizzata.
Lo step successivo da compiere oltre quanto descritto nel lavoro di tesi che ho presentato è quello di adattare il firmware device-side e quello host-side a quello che attualmente è in essere nel dispositivo già implementato per un progetto d'ateneo. Questo è stato sommariamente descritto nell'introduzione della presente tesi. Un'esposizione più dettagliata è disponibile consultando la fonte [1] della bibliografia.

Bibliografia

1. G. BIAGETTI, P. CRIPPA, L: FALASCHETTI, C. TURCHETTI, *A Multi-Channel Electromyography, Electrocardiography and Inertial Wireless Sensor Module Using Bluetooth Low-Energy,* Electronics Journal, Ancona, Italia, 2020.

2. J. AXELSON, *USB Complete: Everything You Need to Develop Custom USB Peripherals,* 4th edition*, Lakeview Research LLC*, Madison, Wisconsin, USA, 2009.

3. K. SIDDHARTH, P. MAHETA, R. J. RAJESH, D. RASHMI RANJAN, *Novel multi-interface USB prototype device for merging commonly used peripheral devices*, IEEE, Pune, India, 2015.

4. COMPAQ, HEWLETT-PACKARD, INTEL, LUCENT, MICROSOFT, NEC, PHILIPS, Universal Serial Bus Specification, Revision 2.0, 27 Aprile 2000.

5. S. GIUSTO, *Il protocollo USB,* Fare elettronica, 2007.

6. *Defined Class Codes,* USB-IF, 2006.

7. *Class definitions for Communication Devices 1.2*, USB-IF , 2007.

8. *Universal Serial Bus 2.0 Cables and Connectors Class Document*, USB-IF, 2007.

9. *nRF52840 product specification*, Nordic Semiconductor Infocenter, 2016.

Sitografia

1. *USB in Nutshell: making sense of USB standard,* beyondlogic.org

2. armKEIL website: armkeil.com

3. Nordic Semiconductor ASA website: nordicsemi.com

4. libusb website: libusb.info

5. Wireshark User's Guide version 3.5.0: wireshark.org