UNIVERSIÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di laurea triennale di ingegneria elettronica

Implementazione di un protocollo USB per il protocollo ToLHnet su microcontrollori Tiva TM4C123G

Implementation of a USB protocol for the ToLHnet protocol on Tiva TM4C123G microcontrollers

Relatore:

Giorgio Biagetti

Tesi di laurea di:

Giovanni Di Pietrantonio

A.A. 2019/2020

Indice

INTRODUZIONE **2**

TOLHNET **3**

TIPOLOGIE DI COMUNICAZIONI **5**

COMUNICAZIONE SERIALE SINCRONA 6

COMUNICAZOINE SERIALE ASINCRONA 6

COMUNICAZIONE ISOCRONA 7

MODALITÀ DI TRASMISSIONE 8

UART E SUA IMPLEMENTAZIONE **9**

UART NEI MICROCONTROLLORI TIVA TM4C123G 9

USB **12**

VELOCITÀ DI TRASMISSIONE 12

LIVELLI DI TENSIONE 13

TRASMISSIONE DI SEGNALI 13

TRASMISSIONE DATI 16

Pacchetto di Handshake *17*

Pacchetti Token *17*

Pacchetti dati *19*

IMPLEMENTAZIONE USB SU MICROCONTROLLORI TIVA 20

CONCLUSIONE **27**

BIBBLIOGRAFIA **28**

Introduzione

Nella seguente tesi si descriveranno diversi protocolli di comunicazione seriale e come possono essere utilizzati nel progetto ToLHnet. Una trasmissione seriale è una comunicazione tra due dispositivi digitali nella quale i bit di informazione sono trasferiti lungo un unico canale di comunicazione uno di seguito all'altro e ricevuti sequenzialmente nello stesso ordine in cui sono stati inviati. Gli standard di comunicazione seriale si differenziano per caratteristiche elettriche e protocolli di trasmissione.

La tecnologia Ethernet è uno standard utilizzato per la composizione di reti locali e definisce le specifiche tecniche del livello fisico e del livello MAC della pila ISO/OSI. Il protocollo I^2C è un sistema di comunicazione seriale utilizzato tra circuiti integrati. L'USB è il protocollo attualmente più utilizzato, è utilizzato per la comunicazione e/o l'alimentazione di periferiche per computer e per qualsiasi altro dispositivo predisposto, come ad esempio smartphone, fotocamere, etc.

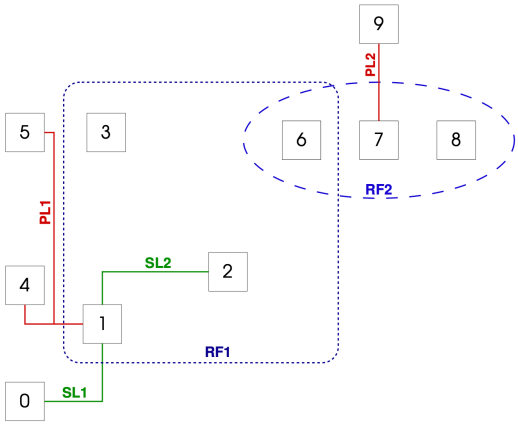
ToLHnet

Il ToLHnet (acronimo di “tree or linear hopping network”) è un protocollo di rete sviluppato per la creazione di reti miste, impiegando connessioni wireless e cablate tra i diversi nodi.

L’obiettivo principale di questo protocollo è quello di mantenere il firmaware e l’overhead introdotto dal livello di rete i più piccoli possibili. Per questo motivo è stato progettato in modo da favorire l’implementazione asimmetrica, spostando la maggior complessità su un singolo nodo speciale che sarà appunto il nodo controllore principale della rete, permettendo così agli altri nodi di essere più semplici. Ovviamente il nodo principale disporrà di memoria e potenza di elaborazione maggiori rispetto a quelle dei nodi semplici che compongono la rete e fungerà anche da gateway per permettere la comunicazione con le altre reti.

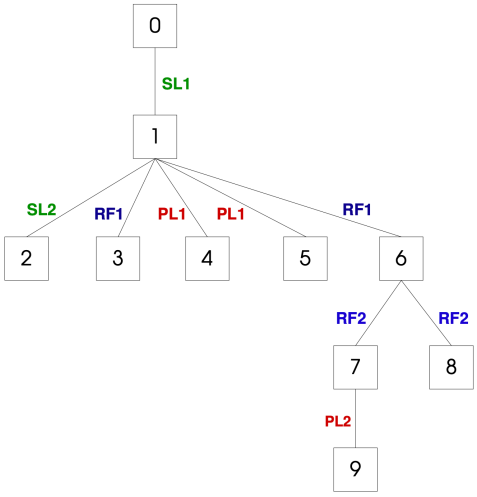
Una rete ToLHnet può comprendere più mezzi di trasmissione, che si tratti di connessioni via cavo point-to-point o collegamenti wireless.

Un esempio di rete fisica è rappresentato come in figura in cui i nodi possono essere collegati tra loro sia tramite collegamenti seriali (SL1 e SL2), collegamenti tramite powerline (PL1 e PL2) o connessioni wireless (RF1 e RF2). Un insieme di nodi connessi attraverso lo stesso mezzo di trasmissione può essere definito come un PBD – Physical Broadcast Domain (dominio di trasmissione fisico), un singolo nodo può appartenere anche a più PBD contemporaneamente come, ad esempio, i nodi 1 e 2 appartengono sia al PBD dei sistemi connessi wireless in RF1 sia al collegamento cablato seriale SR2. Questi nodi in comune permettono ai nodi non connessi direttamente tra loro di comunicare tra loro, fungendo da router. Per esempio, se il nodo 4 volesse inviare un pacchetto dati al nodo 7, questo pacchetto potrebbe passare per il nodo 1 e il nodo 6 per poi giungere a destinazione.



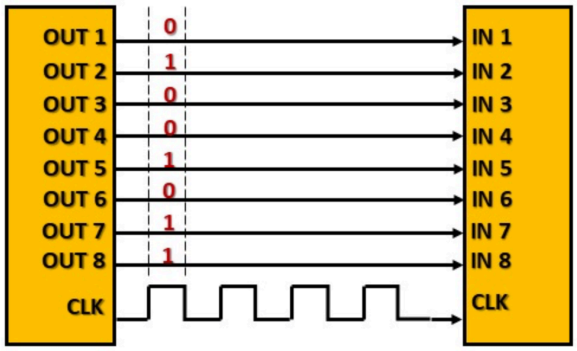
Per semplificare l'implementazione del protocollo, il ToLHnet utilizza una topologia ad albero, dove viene trasmesso un solo pacchetto per volta all'interno dell'albero in un dato momento che viaggia attraverso i rami e i nodi intermedi che fungono da router. Il nodo 0, che si trova in cima all'albero è il nodo speciale con maggior capacità rispetto agli altri nodi (lo chiameremo nodo master), oltre ad avere maggior capacità computazione degli altri nodi deve eseguire delle attività aggiuntive:

- Costruire una struttura logica ad albero partendo dalla topologia fisica.
- In accordo con la struttura ad albero logica costruire le tabelle di routing di tutti i nodi
- Configurare i nodi della rete assegnando a ciascuno di loro la propria tabella di routing
- Risolvere gli errori di rete come perdita di pacchetti, interruzione del collegamento tra nodi di rete, ecc.
- Eseguire operazioni più complesse e permettere alla rete di interfaccciarsi con l'utente e altre alte reti.

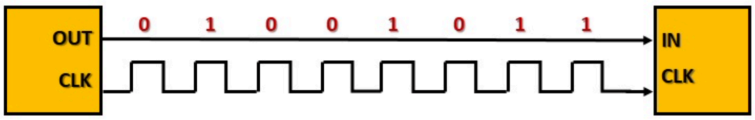


TIPOLOGIE DI COMUNICAZIONI

La comunicazione tra due dispositivi può avvenire in due diverse forme: seriale o parallelo.



La figura riportata qui sopra mostra un esempio di comunicazione in parallelo tra due dispositivi. In questo modello di comunicazione è possibile trasferire più di un bit contemporaneamente attraverso un bus dati poiché i bit di informazione viaggiano nello stesso momento su linee diverse (come nella figura qui in alto). Generalmente oltre ai cavi di trasmissione per i bit è presente un ulteriore collegamento per il clock (CLK) che ha come scopo quello di fornire la temporizzazione con cui scandire i bit.



Nella comunicazione seriale (rappresentata con la figura qui sopra) invece il flusso di dati viaggia lungo un singolo cavo e anche in questo caso, per lo stesso motivo di prima, è presente un ulteriore collegamento per il CLK.

Sembrerebbe così che la comunicazione in parallelo sia più efficiente di quella seriale, questo è vero ma essa impiega molte più linee di I/O e nei microcontrollori in genere lo spazio a disposizione è limitato e per questo motivo si preferisce optare per una comunicazione di tipo seriale. Per questo motivo nel corso del tempo si sono sviluppati diversi tipi di protocolli seriali che possono essere raggruppati in due categorie, sincroni e asincroni.

COMUNICAZIONE SERIALE SINCRONA

A livello fisico, nella comunicazione sincrona oltre ai canali di trasmissione e ricezione e presenta un ulteriore canale per il clock. Questo tipo di configurazione, rispetto all'asincrona, è più semplice e veloce poiché è già presente una sincronizzazione tra il trasmettitore e il ricevitore, di contro richiede, per l'appunto, la presenza di un canale aggiuntivo per il segnale di sincronizzazione che verrà utilizzato per temporizzare l'invio dei singoli bit. Un di questa configurazione è legato proprio alla presenza di questo segnale di sincronismo, infatti a causa della sua presenza non è possibile integrare direttamente sistemi sincroni su impianti già esistenti, come la linea telefonica o la rete elettrica, per ovviare a questo problema è necessario utilizzare dei modem adatti alla trasmissione dati di tipo sincrono connesso tramite una linea seriale sincrona al terminale.

In caso di trasmissione seriale attraverso un sistema terminale-modem è necessaria un'interfaccia che permetta di trasformare il segnale sincrono in un segnale adatto alla rete e di un computer sufficientemente potente da soddisfare le esigenze di continuità di flusso dei dati tipici di una comunicazione sincrona.

Qualsiasi protocollo venga utilizzato la trasmissione è preceduta da tra caratteri SYN (synchronous indle) della tabella ASCII, corrispondente alla sequenza binaria 0001 0110 affinché il terminale ricevente posso ben sincronizzarsi, di questi tre caratteri in genere almeno il primo viene perso mentre gli alti due dovrebbero arrivare a destinazione per permettere un controllo della bontà della linea e del sistema. Se il trasmettitore non avesse temporaneamente dati significativi da inviare ma non vuole far cadere la comunicazione sincrona, deve trasmettere il carattere DLE (data link escape) della tabella ASCII, corrispondente alla sequenza 0001 0000.

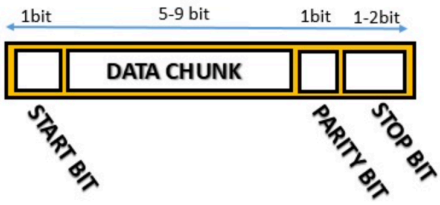
COMUNICAZOINE SERIALE ASINCRONA

Nella comunicazione asincrona il segnale di clock non è presente, risparmiamo una linea ma così è necessario implementare un metodo per la ricezione e la trasmissione dei dati. In questa modalità la sincronizzazione avviene attraverso i dati stessi, è possibile fare questo definendo alcune regole che riguardano la loro composizione valide sia per il trasmettitore che per il ricevitore, vanno definiti 4 campi: bit dati, bit di sincronizzazione, bit di controllo e baud rate. I due dispositivi per comunicare dovranno essere configurati alla stessa maniera. Nel seguito verranno descritti i campi sopracitati e come generalmente viene composto un pacchetto dati inviato tramite comunicazione asincrona.

Il **baud rate** specifica quanto velocemente i dati vengono inviati sulla linea, generalmente è indicata in bps (bit per secondo) e dal suo valore si può ricavare il tempo che un bit occupa nella trasmissione dati. Grazie a questo campo è possibile omettere il clock poichè si conosce a priori la lunghezza e la velocità di un singolo bit. La velocità di baud rate va scelta in base alla potenza computazionale dei

dispositivi che compongono la trasmissione, poiché un valore sempre più elevato di questa grandezza comporta un aumento degli errori di comunicazione.

L'operazione di impacchettamento dati prende il nome di **framing**, tra i due dispositiva va stabilita anche la "forma" della sequenza di dati che verrà trasmessa. I dati vengono suddivisi in pacchetti più piccoli di lunghezza variabile (dipende dal protocollo), questi singoli pacchetti per essere inviati devono essere inglobati in una struttura chiamata **frame** (rappresentato nella figura qui sotto).

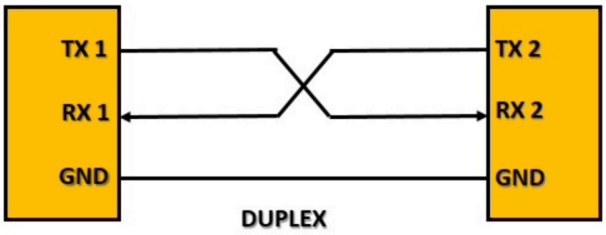


All'inizio e alla fine di un frame sono presenti i **bit di sincronizzazione** suddivisi in bit di start e bit di stop, questi bit sono importanti perché definiscono l'inizio e la fine di un frame. Subito dopo lo start bit è presente il piccolo pacchetto dati seguito dal **bit di parità** (nel caso dell'esempio) e alla "chiusura" del frame sono presenti i bit di stop. È impostante definire la lunghezza di tutte le componenti del frame per permettere al ricevitore di estrapolare correttamente solo i bit portatori di informazione. Il bit di parità è utilizzato per verificare se sono avvenuti degli errori durante la trasmissione, nel caso di controllo di parità pari se la somma dei bit 1 contenuti nel pacchetto dati è dispari il bit di parità diventerà 1 per far in modo che "in totale" i bit 1 siano pari, nel controllo a parità dispari la somma degli 1 deve essere dispari (ovviamente non è vietato sviluppare protocolli che dispongano di funzioni di controllo migliori del controllo di parità).

COMUNICAZIONE ISOCRONA

La comunicazione isocrona è un particolare caso di comunicazione asincrona su un dispositivo di tipo sincrono, è possibile effettuare una trasmissione di questo tipo solo quando la velocità della trasmissione asincrona è un quarto della trasmissione sincrona.

MODALITÀ DI TRASMISSIONE

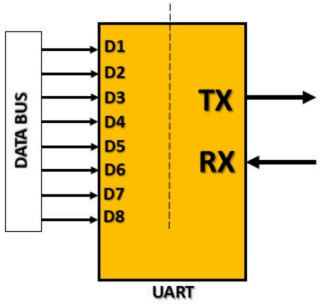


Un bus seriale consiste in due cavi che connettono tra loro due dispositivi. Ognuno di loro ha un canale per la trasmissione (Tx) e uno per la ricezione (Rx), le due linee di trasmissione che connettono i due dispositivi sono connesse ad un'estremità nel canale di ricezione di un dispositivo e l'altra estremità al canale di trasmissione dell'altro, in modo che ogni cavo sia impiegato per la trasmissione da un dispositivo all'altro. Oltre alle due linee sarà presente una terza linea per permettere ai due dispositivi di avere lo stesso valore di GND (ground/terra) in modo che i dati trasmessi abbiano livelli di voltaggio coerenti.

Un sistema di questo è chiamato **half-duplex** quando è possibile comunicare solo lungo una direzione per volta, se uno dei due dispositivi sta trasmettendo l'altro deve attendere che il primo termini la comunicazione prima di poter trasmettere a sua volta (un esempio è la comunicazione tramite walkie-talkie). Quando invece è possibile per ogni dispositivo trasmettere allo stesso momento la comunicazione è di tipo **full-duplex**. Caratteristica di queste due comunicazioni seriali è la bidirezionalità ma esistono anche sistemi monodirezionali in cui la comunicazione è possibile solo in un verso, cioè un dispositivo può solo trasmettere, mentre il secondo può solo ricevere, questo tipo di modalità è chiamata **simplex**.

UART E SUA IMPLEMENTAZIONE

L'UART (acronimo di Universal Asiynchronus Receiver/Transmitter) è una parte di circuito responsabile della comunicazione seriale.



Un UART è un circuito integrato che funge da interfaccia tra la comunicazione seriale e quella parallela. Come mostrato in figura, un UART da un lato riceve in ingresso un bus dati per una comunicazione in parallelo e dall'altro sono presenti due porte seriale, una per la trasmissione (Tx) e una per la ricezione (Rx) di dati seriali. Questi dispositivi si occupano di svolgere operazioni complesse, quali, per la trasmissione creano pacchetti dati ai quali aggiungono i bit di controllo e di sincronizzazione, per poi inviare questo frame sulla linea di Tx. Per la ricezione invece campionano i bit ricevuti in accorto con la velocità di baud rate stabilita, per poi eliminare dal frame ricevuto i bit di sincronizzazione e, dopo aver effettuato i relativi controlli, estrarre i dati utili.

UART NEI MICROCONTROLLORI TIVA TM4C123G

Nei microcontrollori di classe Tiva TM4C sono presenti 8 UART, alcune delle loro caratteristiche sono le seguenti:

- Baud rate programmabile fino a 5 Mbps
- FIFO di 16(bit)x8 sia per il Tx che per l'Rx
- Frame dati composto da: 1 bit di start, da 5 a 8 bit di informazione, 1 bit per il controllo di pairtà e da 1 a 2 bit di stop (in base a come verrà impostato)

Prima di poter utilizzare uno degli otto UART per trasmettere o ricevere dati bisogna abilitarlo e poi configurarlo. La funzione qui di seguito mostra come abilitare, il primo denominato UART0 (le loro denominazioni per questi controllori vanno da UART0 a UART7).

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
```

Ora che è stato abilitato è possibile passare all'inizializzazione dei parametri.

```
UARTConfigSetExpClk(UART0_BASE,
                    SysCtlClockGet(),
                    38400,
                    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
```

Il primo parametro della funzione (**UART0_BASE**) è l'indirizzo della porta del primo UART, il secondo paramtro (*SysCtlClockget()*) è una funzione che restituisce la frequenza di clock fornita al modulo UART, il terzo paramtro è un numero, 38400, è indica il baud rate, l'ultimo parametro serve per specificare il formato del frame che vorremmo inviare/ricevere, nello specifico, in base all'esempio qui sopra, con **UART_CONFIG_WLEN_8** si è specificato che i bit di informazione al suo interno sono 8, **UART_CONFIG_STOP_ONE** imposta un solo bit di stop (per due bit di stop si usa **UART_CONFIG_STOP_TWO**) e con **UART_CONFIG_PAR_NONE** stabiliamo che non ci saranno bit per il controllo di parità (nel datasheet sono indicati i valori per gli altri tipi di configurazione), le tre componenti vanno indicate nell'ordine scritto qui sopra e separate dal carattere '|' OR come nell'esempio. Una volta configurato, l'UART può essere utilizzato per la trasmissione e la ricezione di dati seriali. Per la ricezione, di un carattere ad esempio, rimaniamo in attesa che ne entri uno nel FIFO di ricezione:

```
while(!UARTCharsAvail(UART0_BASE)) { }
```

La funzione *UARTCharsAvail()* richiede come parameto solo l'indirizzo di una porta UART, restituisce true se è presente un carattere nel FIFO di ricezione, false altrimenti. In pratica rimane nel loop fino a quando non riceve un carattere. Una volta che ci siamo assicurati della presenza di un carattere nel FIFO dobbiamo estrarlo:

```
while(UARTCharGetNonBlocking(UART0_BASE)) { }
```

La funzione *UARTCharGetNonBlocking()* estrae dal FIFO di ricezione della porta UART specificata come parametro il carattere che ora potrà essere inserito nel buffer di output. La funzione restituisce -1 quando nel FIFO non ci sono caratteri, così che possa uscire dal loop. Invece per poter

```
UARTCharPut(UART0_BASE, 'c'));
```

trasmettere un carattere è soltanto necessario posizionarlo all'interno del FIFO di trasmissione usando la seguente funzione (che in questo caso si prepara a trasmettere il carattere 'c').

Una volta svolte tutte le operazioni è sempre consigliabile disabilitare l'UART, questo è possibile farlo utilizzando la seguente funzione inserendo come parametro l'indirizzo dell'UART da voler disabilitare, sempre nel caso del primo UART:

```
UARTDisable(UART0_BASE);
```

USB

L'USB (Universal Serial Bus) è uno standard per la comunicazione seriale, utilizzato anche per l'alimentazione di periferiche per computer o altri dispositivi.

Questo standard definisce le specifiche per cavi, connettori, protocolli per la connessione, la comunicazione e l'alimentazione tra dispositivi. La tensione di alimentazione standard del protocollo USB è di 5V ma i dispositivi che seguono il protocollo USB Power Delivery (UPD) possono avere tensioni di alimentazione maggiori, anche fino a 50V. Lo standard prevede che il connettore porti anche un cavo di alimentazione (VBUS) per periferiche a basso consumo.

La comodità di questo protocollo consiste nella possibilità di aggiornare dispositivi sprovvisti di USB attraverso schede PCI (Peripheral Component Interconnect) o PC Card. Inoltre, le porte USB sono dotate del supporto Plug and Play e supportano i dispositivi hot swap così da permettere ad un utente di poter connettere un dispositivo senza la necessità di installarlo e dando la possibilità di connetterlo o disconnetterlo anche mentre il sistema è avviato. Esistono diverse tipologie di USB, l'USB di tipo A e di tipo B permettono di collegare periferiche che richiedono il trasferimento dei dati ma non segnali video, a differenza dell'USB di tipo C.

VELOCITÀ DI TRASMISSIONE

Esistono varie versioni del protocollo USB, da essa dipende la velocità di trasmissione, nel caso dell'USB 2.0 il bit-rate è di 60MB/s (da ridividere tra tutti i dispositivi collegati alla porta), mentre per i dispositivi USB 3.0 la velocità massima di scrittura varia tra 70-90MB/s mentre di lettura invece 90-110MB/s. In base al bit-rate i protocolli USB vengono suddivisi nelle seguenti categorie:

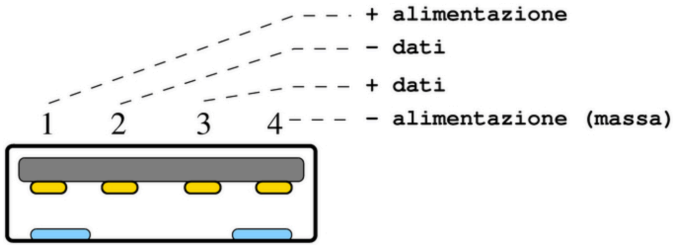
- **Low-speed(LS):** definita anche USB 1.0, la velocità è di 1,5Mbit/s.
- **Full-speed(FS):** bit-rate di 12Mbit/s, è la velocità di trasmissione standard di un USB 1.0. Tutti gli hub USB possono funzionare a questa velocità.
- **Higt-speed(HS):** velocità di 60MB/s, introdotta nel 2001, nota comunemente come USB 2.0. Tutti i dispositivi HS hanno il connetto identico a quello dell'USB 1.0 e sono retrocompatibili con esso.
- **SuperSpeed(SS):** raggiungono la velocità di 5Gbit/s, introdotta nel 2010, chiamata anche USB 3.0, in genere i dispositivi a tale velocità sono retrocompatibili con i precedenti e introducono un nuovo tipo di cablaggio e permettono il funzionamento Full-duplex.
- **SuperSpeed+(SS+):** definito USB 3.1 permette di raggiungere il bit-rate di 10Mbit/s, usando due linee dati ulteriori anche i 20 Mbit/s (in questo caso si tratta di USB 3.2).

Per i dispositivi a LS o FS il tempo minimo di una transizione è di 1ms, il tempo di transizione di un HS invece varia da 940ns a 984ns, in dipendenza dal fatto che il pacchetto di interrupt sia da 1 o 4 byte.

LIVELLI DI TENSIONE

I segnali USB vengono trasmessi utilizzando un segnale differenziale su un cavo dati a doppino intrecciato con impedenza caratteristica di 90 Ω ± 15%. In base alle tipologie di trasmissione i livelli energetici sono i seguenti:

- **LS e FS:** utilizzano una singola coppia di fili, denominati D- e D+, in half-duplex. I livelli di tensione per il livello logico passo sono tra 0 e 0.3 V, mentre per il livello logico alto tra 2.8 e 3.6 V.
- **HS:** usa una coppia di fili come i precedenti ma con una convenzione elettrica diversa. Il livello logico basso si ottiene con una tensione tra -10 e 10 mV mente quello alto tra 360 e 440 mV, inoltre è presenta un terminatore per adattare la linea.
- **SS:** aggiunge un'ulteriore coppia di cavi intrecciati e schermati. Questi servono per permettere la comunicazione full-duplex, per la half-duplex si continua ad utilizzare la configurazione precedente.
- **SS+:** utilizza una maggiore velocità di trasmissione dati e/o la line aggiuntiva dei connettori type-C.



TRASMISSIONE DI SEGNALI

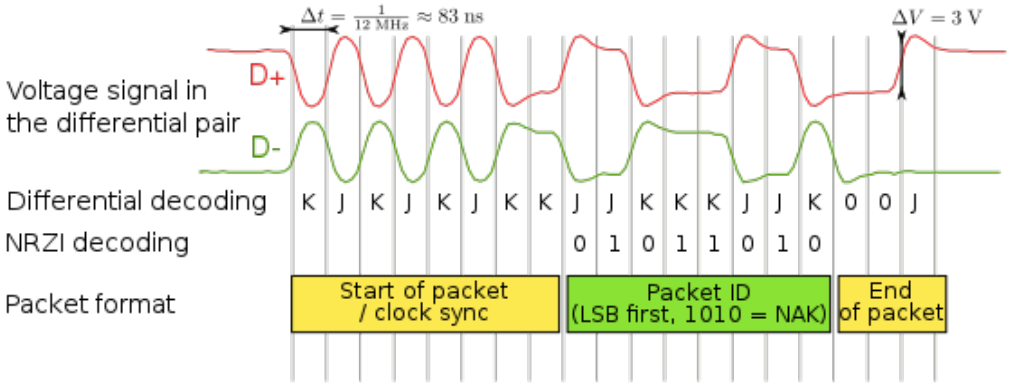
Una comunicazione USB avviene sempre tra un host o hub e un device, l'host include un resistore di pull-down di 15kΩ per ciascuna delle linee dati. Quando nessun dispositivo è connesso entrambe le linee dati vengono portate a livello basso in uno stato chiamato "*single-ended zero"* (SE0) che sta ad indicare che il dispositivo in quella porta non è connesso alcun dispositivo (o se c'è un dispositivo connesso va resettato).

Per semplificare la discussione verrà utilizzata la seguente terminologia:

Segnale	Stato di transizione linea	Descrizione	Valori linee USB bassa velocità		Valori linee USB alta velocità	
			D+	D-	D+	D-
J	uguale allo stato di linea inattiva	stato che si presenta durante una trasmissione, indica anche quando la linea è inattiva	basso	alto	alto	basso
K	inverso di J	stato che si presenta durante una trasmissione	alto	basso	basso	alto
SE0	single-ended zero	sia D+ che D- sono a livello basso, indica che il dispositivo e disconnesso e/o che va resettato	basso	basso	basso	basso
SE1	single-ended one	non dovrebbe verificarsi, indica un errore	alto	alto	alto	alto

Nei dispositivi a bassa velocità (trasmissioni *low-bandwitch*) quando sono inattivi è la linea D- ad avere lo stato alto, per i dispositivi ad alta velocità invece (trasmissioni *full-bandwitch*) è D+ ad essere a livello alto. Il segnale K ha sempre polarità opposta a quella di J. Nel segnale SE1 (*single-ended one*) sia D+ che D- sono a livello alto, questo stato avverte dell'avvenimento di un errore e non dovrebbe mai verificarsi.

Per la trasmissione dati l'USB utilizza la codifica NRZ: il bit 0 viene trasmesso alternando le linee dati dal segnale J al segnale K o viceversa, il bit 1 viene trasmesso rimanendo sullo stesso livello energetico. Per evitare che si persa la sincronizzazione nel caso in cui si abbiano lunghe sequenze di 1 vengono utilizzati bit di stuffing. Ogni sei bit 1 viene aggiungo un bit di stuffing 0, in questo modo è possibile che venga riscontrata un errore anche rilevando sette bit 1 consecutivi. Nel protocollo USB 3.0 viene utilizzata una codifica di dati diversa per poter gestire le velocità di trasmissione più elevate.



L'immagine qui sopra mostra un esempio di trasmissione USB FS (USB 1.0) composta dai seguenti campi:

- **Pacchetto di sincronizzazione:** si parte con lo stato J, che sta ad indicare l'inattività della linea, con D+ e D- rispettivamente alto e basso. La sequenza di sincronizzazione corrisponde al ad un byte 0000 0011, che si ottiene effettuando le commutazioni KJKJ KJKK, l'ultima coppia di bit 1 (che si presenta come un doppio stato K) sta ad indicare la fine della sincronizzazione e l'inizio del frame dati USB, per frame dati grandi (trasmissione *high-bandwitch*) la sincronizzazione arriva anche ad essere di 4 byte.
- **End of packet (EOP):** indica la fine del frame. Si rappresenta con due stati SE0 in cui entrambe le linee sono a livello basso. Successivamente la linea torna allo stato di inattività portandosi allo stato J.
- **Bus reset:** non è presente in figura. Un ripristino del bus USB viene effettuato con un segnale SE0 prolungato per un tempo di 10-20 ms.

Per i protocolli USB 2.x il device deve segnalare all' host che necessita di maggior larghezza di banda, per farlo viene utilizzato un protocollo di reset chiamato *chirping* (cinguettio). Un dispositivo USB 2.0 si collega prima come un dispositivo FS, ma dopo il bus reset posiziona D- a livello alto, ottenendo uno segnale K quindi, questo sta ad indicare che il dispositivo collegato è di tipo HS. Se l'host è compatibile per questo tipo di trasmissione allora in risposta al device invia un segnale di *chirping* che consiste in una sequenza di stati J e K alternati. Il device deve ricevere almeno tre serie di JK per avere conferma che l'host è compatibile per trasmissioni HS. I device che utilizzano USB 3.0 hanno una linea aggiuntiva oltre a quella utilizzata dall'USB 1.x e 2.x, quindi questa segnalazione non è necessaria.

TRASMISSIONE DATI

Una comunicazione USB avviene attraverso lo scambio di pacchetti. Dopo. La sincronizzazione tutti i pacchetti sono costituiti da byte, il primo byte che viene trasmesso è sempre un byte di identificazione del pacchetto, chiamato PID. In realtà il PID è di 4 bit ma il byte che forma è composto prima dalla sequenza invertita di questi 4 bit seguiti dal loro complemento bit a bit, questa ridondanza è utilizzata per la rilevazione degli errori inoltre impedisce la presenza di lunghe sequenze di 1 poiché non sarà possibile averne più di quattro consecutivi. Nella tabella seguente sono indicati i possibili valori di PID e le relative informazioni in base al suo valore.

Tipo	Valore PID	Byte trasmesso	Nome	Descrizione
Riservato	0000	0000 1111		
Token	1000	0001 1110	SPLIT	Stream di dati suddiviso (USB 2.0)
	0100	0010 1101	PING	Controlla se l'endpoint può accettare dati (USB 2.0)
Speciale	1100	0011 1100	PRE	Preambolo trasmissione a larghezza di banda ridotta (USB 1.0)
handshake			ERR	Errore trasmissione (USB 2.0)
	0010	0100 1011	ACK	Pacchetti dati accettati
	1010	0101 1010	NAK	Pacchetti dati non accettati, richiesta di ritrasmissione
	0110	0110 1001	NYET	Dati non ancora pronti
	1110	0111 1000	STALL	Impossibile trasmettere; eseguire ripristino
Token	0001	1000 0111	OUT	Indirizzo per trasferimento host-to-device
	1001	1001 0110	IN	Indirizzo per il trasferimento device-to-host
	0101	1010 0101	SOF	Marcatore inizio frame
	1101	1011 0100	SETUP	Indirizzo di controllo per il trasferimento host-to-device
Data	0011	1100 0011	DATA0	Numero di pacchetto dati pari
	1011	1101 0010	DATA1	Numero di pacchetto dati dispari
	0111	1110 0001	DATA2	Pacchetto dati per il trasferimento isocrono di un pacchetto dati (USB 2.0)
	1111	1111 0000	MDATA	Pacchetto dati per il trasferimento isocrono di un pacchetto dati (USB 2.0)

I pacchetti sono suddivisi in tre categorie principali, ciascuno con formato diverso e controllo CRC con in testa sempre la sequenza di sincronizzazione. Nel seguitosi si userà la dicitura EOP per indicare il campo *End Of Packet* che sarà sempre uguale ad una sequenza di due SE0 (indicati con 0 per comodità) e un J.

Pacchetto di Handshake

Sinc	PID	EOP
KJ KJ KJ KK	XXXX XXXX	00J

Questo pacchetto è costituito dal solo byte PID, il rilevamento degli errori avviene secondo la forma complementata precedentemente descritta, è un pacchetto che generalmente è inviato in risposta ai pacchetti dati. I tre tipi "principali" sono ACK, utilizzato per confermare la ricezione del pacchetto, NAK, che indica che il pacchetto non è stato accettato dal ricevitore e che va ritrasmesso e STALL che indica l'impossibilità di trasmettere il pacchetto. Con il protocollo USB 2.0 sono stati aggiunti altri handshake: NYET e ERR. Con NYET indichiamo che, in caso di trasmissione suddivisa su più pacchetti, la comunicazione non è ancora stata completata, con ERR invece che questa trasmissione non è riuscita. Si utilizza NYET anche quando si vuole comunicare all'host che il pacchetto dati è stato ricevuto ma il device non è più in grado di ricevere pacchetti a causa del buffer pieno, dopo aver ricevuto questo messaggio l'host può mandare in messaggio PING per verificare le condizioni del dispositivo in modo da poter inviare il pacchetto successivo quando il buffer si sarà liberato, diversamente riceverebbe un pacchetto NAK.

Pacchetti Token

Un pacchetto token è costituito dal PID, seguito da 2 byte di payload, 11 bit di indirizzo e il controllo CRC-5. Questo tipo di pacchetti viene inviato esclusivamente dall'host, mai dai dispositivi. I pacchetti token utilizzati nelle trasmissioni USB 1.0 sono i seguenti:

- **IN** e **OUT** sono costituiti da 7 bit per indicare il device con cui sta comunicando e da altri 4 bit per la funzione di utilizzo (in caso di device multifunzione), rispettivamente dicono al device se devono trasmettere un pacchetto DATAx o riceverlo. Con un token **IN** l'host si aspetta di ricevere come risposta dal device un handshake NAK o STALL oppure il pacchetto DATAx, in quest'ultimo caso invierà come risposta un ACK. Subito dopo un token **OUT** l'host invierà il pacchetto DATAx al device, aspettandosi come risposta un handshake tra ACK, NAK NYET o STALL.
- La funzione di **SETUP** è molto simile a quella di OUT ma viene utilizzato solo per configurare il dispositivo. È sempre seguito da un pacchetto DATA0 lungo 8 byte standardizzato.

- Ogni millisecondo (nelle trasmissioni ad alta velocità) l'host USB trasmette un token **SOF** (**S**tart **O**f **F**rame) dove al posto dei bit di indirizzo c'è un numero pari al numero di frame trasmessi. È utilizzato anche per la sincronizzazione nelle trasmissioni isocrone e per interrompere la trasmissione di dati. I dispositivi USB 2.0 ricevono 7 token SOF aggiuntivi per frame.
- Con **PING** l'host chiede al dispositivo se è in grado di ricevere o meno un pacchetto OUT o DATAx. Viene utilizzato dopo che il dispositivo ha risposto con handshake NAK o NYET. Quando riceve queste risposte l'host attende per un certo lasso di tempo prima di inviare nuovamente un PING a cui il device, a seconda dei casi, risponderà ACK, NAK o STALL.
- **SPLIT** viene utilizzato per suddividere la transazione dati. L'host USB se si trova a comunicare con un dispostivi USB più lento, invece che rimanere ad aspettare quel device invia un token SPLIT all' hub più vicino seguito da uno o due pacchetti dati e, quando è richiesta una risposta, invia un secondo token SPLIT. Questo pacchetto in aggiunta contiene 7 bit per il numero dell'hub, 12 bit per i flag di controllo e 5 bit per il controllo CRC-5.

Nel seguito verranno rappresentati i formati dei vari pacchetti.

Formato token OUT, IN, SETUP e PING

Campo	Sinc	PID	indirizzo	ENDP	CRC-5	EOP
Numero bit		8	7	4	5	
Segnale	KJ KJ KJ KK	XXXX XXXX	XXXX XXX	XXXX	XXXXX	00J

Con **ENDP** si indica l'endpoint, il buffer del device che trasmette/riceve.

Formato token SOF

Campo	Sinc	PID	Numero del frame	CRC-5	EOP
Numero bit		8	11	5	
Segnale	KJ KJ KJ KK	XXXX XXXX	XXXX XXXX XXX	XXXXX	00J

Formato token SSPLIT (start-split trasmission) e CSPLIT (compleate-split trasmission)

0 = SSPLIT 1 = CSPLIT	Sinc	PID	Indirizzo Hub	S/C	Numero porta	S	E U	EP	CRC-5	EOP
Numero bit		8	7	1	7	1	1	2	5	
Segnale	KJ KJ KJ KK	XXXX XXXX	XXXX XXX	X	XXXX XXX	X	X	XX	XXXXX	00J

Con il campo S/C si indica se la trasmissione è stata completata oppure no, con S/C = 0 indichiamo un pacchetto SSPLIT che indica che è iniziata una trasmissione SPLIT (o che comunque non è ancora conclusa), con S/C = 1 indichiamo un paccheto CSPLIT per segnala che la trasmissione è conclusa. Quando il campo S (quello dopo il campo numero porta) è uguale ad 1 la trasmissione è a bassa velocità, è 0 quando la trasmissione è ad alta velocità. Dopo il campo S abbiamo il campo E in caso di SSLIT per indicare la fine del carico utile a piena velocità o il campo U in caso di CSPLIT che è un campo riservato che corrisponde sempre ad un bit 0. In fine con EP indichiamo il tipo di pacchetto: 00 = controllo, 01 = isocrono, 10 = bulk, 11=interrupt.

Pacchetti dati

Campo	Sinc	PID	Dati	CRC-16	EOP
Numero bit		8	0-8192	16	
Segnale	KJ KJ KJ KK	XXXX XXXX	(XXXX XXXX) * Nbyte	XXXX XXXX XXXX XXXX	00J

Un pacchetto dati è costituito dal PID, seguito da un pacchetto di dati di informazione di lunghezza variabile da 1 a 1024 byte e dal controllo CRC a 16 bit. In realtà la lunghezza massima del campo dati è di 1024 bye solo per i protocolli higt-speed, per i full-speed questo campo è di 64 byte mentre per i low-speed di soli 8 byte. Esistono due tipologie principali di pacchetti dati: DATA0 e DATA1. Un pacchetto dati deve essere sempre preceduto da un token di indirizzo e seguito sempre da un handshake di ACK per la conferma ricezione. Se un dispositivo USB non dovesse ricevere l'ACK di conferma per i dati che ha trasmesso non sa se il pacchetto è stato ricevuto o meno, il pacchetto potrebbe essere andato perso durante la trasmissione oppure ad essere andato perso potrebbe essere stato il pacchetto di handshake, in entrambi i casi il trasmettitore provvedere al rinvio del pacchetto. Per risolvere problema della duplicazione dei pacchetti il dispositivo in ricezione tiene traccia dell'ultimo pacchetto DATAx che è stato ricevuto, così che, se dovesse ricevere una copia di un pacchetto già ricevuto, lo riconoscerà e lo ignorerà (dopo un pacchetto di tipo DATA0 seguirà un pacchetto DATA1 che verrà seguito a sua volta da un pacchetto DATA0, quindi non posso ricevere due DATA0 o DATA1 di seguito). Se il controllo CRC dovesse rivelare errori il destinatario non provvederà ad inviare un ACK, così che il trasmettitore, non vedendolo arrivare, provvederà a trasmettere una copia di tale pacchetto.

Quando un dispositivo viene ripristinato con un token SETUP si aspetta sempre di ricevere un pacchetto di 8 byte standardizzato.

Con l'USB 2.0 sono stati introdotti nuovi tipi di pacchetti dati: DATA2 e MDATA. Questi due sono utilizzati solo dai dispositivi che effettuano trasmissioni high-bandwitch isocrone.

IMPLEMENTAZIONE USB SU MICROCONTROLLORI TIVA

I trasferimenti USB possono essere impostati per essere di varie tipologie in base anche a come viene implementata la trasmissione. Si possono implementare trasferimenti di tipo *isocrono* in cui alla periferica è garantita una quantità di banda (bit trasmessi in 125 µs o 1 ms) predeterminata, questo trasferimento è utilizzato principalmente per trasmissioni audio o video, in questo tipo di trasferimento il controllo degli errori è disabilitato. Nelle trasmissioni di tipo *interrupt* viene stabilita una latenza massima (ad esempio di 1 ms). I trasferimenti di tipo *bulk* (trasferimento implementato nel seguito) sono trasferimenti che avvengono alla massima velocità possibile dopo aver oddisfatto le richieste di tipo isocrono e interrupt, viene utilizzato in tutti i casi in cui è richiesto un trasferimento di dati il più veloce possibile (come ad esempio quando trasferiamo dati su un hard-disk esterno).

Per implementare la trasmissione (di tipo bulk) USB su microcontrollori Tiva TM4C123G per prima cosa bisogna permettere al sistema di identificare il microcontrollore come dispositivo USB, per farlo bisogna definire dei descrittori in base al tipo di funzionalità, questi descrittori rappresentano:

- Le lingue supportate dal dispositivo
- Una stringa identificativa del produttore (Texas Instrument)
- Una stringa identificativa del prodotto
- Una stringa numerica per identificare il dispositivo tra quelli collegati al sistema (a nostra discrezione)
- Una stringa per la descrizione dell'interfaccia
- Una stringa di descrizione della configurazione

Un esempio di configurazione dei descrittori, nel caso di trasmissione di tipo Bulk e nel caso sia supportata una sola lingua, è scritto in codice qui di seguito.

```
/****************** Lingue supportate ******************/

const uint8_t g_pui8LangDescriptor[] =
        {
          4,
          USB_DTYPE_STRING,
          USBShort(USB_LANG_EN_US)
        };

/******************* ID produttore *********************/
```

```
const uint8_t g_pui8ManufacturerString[] =
            {
              (17 + 1) * 2,
              USB_DTYPE_STRING,
              'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
              't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
            };

/******************** ID prodotto **********************/

const uint8_t g_pui8ProductString[] =
            {
              (19 + 1) * 2,
              USB_DTYPE_STRING,
              'G', 0, 'e', 0, 'n', 0, 'e', 0, 'r', 0, 'i', 0, 'c', 0, ' ', 0, 'B', 0,
              'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
              'e', 0
            };

/***************** Stringa numerica **********************/

const uint8_t g_pui8SerialNumberString[] =
            {
              (8 + 1) * 2,
              USB_DTYPE_STRING,
              '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
            };

/****************** Stringa interfaccia ******************/

const uint8_t g_pui8DataInterfaceString[] =
            {
              (19 + 1) * 2,
              USB_DTYPE_STRING,
              'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
              'a', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
              'a', 0, 'c', 0, 'e', 0
            };

/***************** Stringa configurazione ****************/
```

```
const uint8_t g_pui8ConfigString[] =
        {
          (23 + 1) * 2,
          USB_DTYPE_STRING,
          'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
          'a', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
          'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0
        };

/*************** Tabella dei descrittori *****************/

const uint8_t * const g_ppui8StringDescriptors[] =
        {
          g_pui8LangDescriptor,
          g_pui8ManufacturerString,
          g_pui8ProductString,
          g_pui8SerialNumberString,
          g_pui8DataInterfaceString,
          g_pui8ConfigString
        };

/*************** Numero dei descrittori *****************/

#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

Queste stringhe sono valide per un trasferimento dati di tipo Bulk, quindi in caso si volesse configurare il microcontrollore per applicazioni diverse vanno modificate, il numero dei descrittori da inserire dipende dal numero di lingue supportate, in totale quindi saranno (1+(5*numero_lingue)) e vanno scritti prima tutti i descrittori della prima lingua per poi scrivere quelli della successiva. È possibile anche implementare di trasmissione e ricezione, si spiegherà prima come procedere nel caso siano assenti.

Sempre per le trasmissioni bulk, dopo aver definito i descrittori bisogna definire una struttura *tUSBBulkDevice* e inizializzarne i parametri in base all'applicazione che si sta sviluppando, questa struttura è composta nel seguente modo:

```
typedef struct
        {
          const uint16_t ui16VID;
          const uint16_t ui16PID;
          const uint16_t ui16MaxPowermA;
          const uint8_t ui8PwrAttributes;
          const tUSBCallback pfnRxCallback;
          void *pvRxCBData;
          const tUSBCallback pfnTxCallback;
          void *pvTxCBData;
          const uint8_t *const *ppui8StringDescriptors;
          const uint32_t ui32NumStringDescriptors;
          tBulkInstance sPrivateData;
        }
        tUSBDBulkDevice
```

- **ui16VID:** l'ID del venditore presente nella tabella dei descrittori
- **ui16PID:** l'ID del prodotto presente nella tabella dei descrittori
- **ui16MaxPowermA:** la massima potenza del device in mA
- **ui8PwrAttributes:** indica se il dispositivo è autoalimentato o alimentato tramite bus
- **pfnRxCallback:** puntatore alla funzione di callback di ricezione, richiamata quando viene ricevuto un pacchetto
- **pvRxCBData:** puntatore fornito dal client mandato come primo parametro quando viene richiamata la funzione di callback di ricezione (pfnRxCallback)
- **pfnTxCallback:** puntatore alla funzione di callback di trasmissione, richiamata quando viene trasmesso un pacchetto
- **pvTxCBData:** puntatore fornito dal client mandato come primo parametro quando viene richiamata la funzione di callback di trasmissione (pfnTxCallback)
- **ppui8StringDescriptors:** puntatore alla tabella dei descrittori, i descrittori devono essere nell'ordine scritto nel codice precedente, in caso si più lingue vanno scritti prima tutti quelli relativi alla lingua 1, poi quelli per la lingua 2 e così fino all'ultima lingua.
- **ui32NumStringDesciptors:** numero dei descrittori di stringa, si ricorda che deve valere [1+(5*numero_lingue)]
- **sPrivateData:** i dati dell'istanza privata del dispositivo. Questa non deve essere modificata da nessun codice esterno al dispositivo configurato per la trasmissione bulk.

Successivamente a questa operazione bisogna aggiungere funzioni per la gestione degli eventi di ricezione e trasmissione in modo da gestire tutti i messaggi che richiedono una risposta particolare.

Per un generico dispositivo bulk, in ricezione, solo l'evento **USB_EVENT_RX_AVAILABLE** (evento che si verifica quando viene ricevuto un pacchetto) necessita di essere gestito. Quando questo evento si verifica bisogna controllare la quantità di dati ricevuti richiamando la funzione **USBDBulkRxPacketAvaible()** e leggerli utilizzando **USBDBulkPacketRead()**. Facendo ciò si conferma la ricezione del pacchetto da parte dell'host in modo che quest'ultimo possa spedire il pacchetto successivo. Se la funzione non è in grado di leggere il pacchetto restituisce 0 e tenta di nuovo la lettura alcuni millisecondi dopo.

Altri eventi che è meglio gestire se si utilizzano buffer in ricezione e trasmissione sono **USB_EVENT_CONNECTED** e **USB_EVENT_DISCONNECTED** che indicano quando il dispositivo viene connesso o disconnesso dall'applicazione in modo che il buffer venga svuotato o che in caso di dispositivo disconnesso ogni tentativo di trasmissione verso l'host venga bloccato immediatamente come un errore.

Per la trasmissione non ci sono eventi che devono essere necessariamente gestiti ma è consigliata la gestione dell'evento **USB_EVENT_TX_COMPLETE** poiché permette al device bulk di capire quando l'host ha ricevuto correttamente il pacchetto in modo da poterne inviare un altro.

Predisposto il device per la risposta agli eventi non resta che inizializzarlo come dispositivo USB e posizionarlo sul bus in modo da permettergli di comunicare, la funzione che consente di fare questo è **USBDBulkInit()**. Se questa funzione restituisce un valore di ritorno diverso da NULL allora il dispositivo è pronto per comunicare.

Una volta ricevuto l'evento **USB_EVENT_CONNECTED** il device è connesso all'host ed è pronto per comunicare, è possibile trasmettere un pacchetto usando la funzione **USBDBulkPacketWrite()**, quando il pacchetto inviato sarà ricevuto dall'host si verificherà l'evento **USB_EVENT_TX_COMPLETE**.

Di seguito sono illustrate le funzioni precedenti.

- USBDBulkRxPacketAvailable()

```
uint32_t USBDBulkRxPacketAvailable(void *pvBulkDevice)
```

- **pvBulkDevice:** puntatore restituito dalla funzione USBDBulkInit()

La funzione restituisce la dimensione (in byte) del pacchetto che deve essere ancora letto, 0 se non ci sono dati da elaborare.

- USBDBulkPacketRead()

```
uint32_t USBDBulkPacketRead(void *pvBulkDevice,
                            uint8_t *pi8Data,
                            uint32_t ui32Length,
                            bool bLast)
```

- **pvBulkDevice:** puntatore restituito dalla funzione USBDBulkInit()
- **pi8Data:** punta ad un buffer in cui vengono restituiti i dati ricevuti
- **ui32Length:** la dimensione del buffer puntato da pi8Data, se il dispositivo rileva che l'intero pacchetto è stato letto, l'host viene informato che può inviare un nuovo pacchetto
- **bLast:** indica se il dispositivo effettuerà un ulteriore chiamata per leggere i dati

La funzione restituisce il numero di byte letti.

- USBDBulkPacketWrite()

```
uint32_t USBDBulkPacketWrite(void *pvBulkDevice,
                             uint8_t *pi8Data,
                             uint32_t ui32Length,
                             bool bLast)
```

- **pvBulkDevice:** puntatore restituito dalla funzione USBDBulkInit()
- **pi8Data:** punta al primo byte del pacchetto che deve essere trasemsso
- **ui32Length:** il numero dei byte da trasmettere. La dimensione massima di questo parametro è 64 (la dimensione massima di un pacchetto USB). Tentare di inviare un pacchetto di dimensione maggiore farà restituire alla funzione il valore 0 (non invia niente)
- **bLast:** indica se devono essere scritti altri dati prima che il pacchetto debba essere trasmesso, se "true" il dispositivo effettuerà un ulteriore chiamata della funzione, se "false" non ci saranno altre chiamate e il dispositivo programmerà la trasmissione.

La funzione restituisce il numero di byte inviati oppure 0 se la trasmissione non è andata a buon fine.

- USBDBulkInit()

```
void *USBDBulkInit( uint32_t ui32Index,
                    tUSBDBulkDevice *psBulkDevice)
```

- **ui32Index:** indice del controller USB
- **psBulkDevice:** punta ad una struttura che contiene i parametri per il funzionamento del dispositivo

La funzione restituisce NULL in caso di errore o un puntatore void da utilizzare nelle altre funzioni per i dispositivi configurati come device bulk.

Conclusione

Sono stati descritte le varie versioni del protocollo USB in base alla velocità di trasmissione. Il protocollo si è evoluto nel tempo introducendo nuove tipologie di pacchetti per la gestione delle comunicazioni e per le nuove applicazioni per cui viene utilizzato (come la trasmissione video), mostrandone le potenzialità.

È stato mostrato una prima implementazione di una configurazione USB su un microcontrollore di classe Tiva C4M123G, ma per poter funzionare e comunicare con un host è necessaria ancora la configurazione della gestione degli eventi.

Bibbliografia

[1] Texas instruments, *Tiva™ C Series TM4C123G LaunchPad Evaluatio board – User's guide*, aprile 2013.

[2] Texas Instruments, *Tiva™ TM4C123GH6PM Microcontroller - data sheet*, revisione del 12 giugno 2014.

[3] Texas Instruments, *TivaWare™ Peripheral Driver Library – user's guide*, revisione del 25 luglio 2016.

[4] Texas Instruments, *TivaWare™ Utilities Library – user's guide*, revisione del 7 febbraio 2014.

[5] Texas Instruments, *TivaWare™ USB Library – user's guide*, revisione del 7 febbraio 2014.

[6] DII – Università politecnica delle Marche, *ToLHnet – Tree or Linear Hopping network*, 9 settembre 2014.

[7] Wkipedia.org, *Trasmissione seriale*, (https://it.wikipedia.org/wiki/Trasmissione_seriale), 5 novembre 2020.

[8] Wikipedia.org, *Ethernet*, (https://it.wikipedia.org/wiki/Ethernet#Indirizzo_Ethernet), 29 gennaio 2021.

[9] Wikipedia.org, *I^2C*, (https://it.wikipedia.org/wiki/I²C), 19 dicembre 2020.

[10] *Il protocollo i2c*, (https://sites.google.com/site/sitodelguru/elettronica/il-protocollo-i2c)

[11] Wikipedia.org, *USB (Communications)*, (https://en.wikipedia.org/wiki/USB_(Communications)#Data_packets), 20 ottobre 2020.