Versione N° 1.1

LaurTec

Utilizzare la libreria USB Microchip

Esempi ed applicazioni

Autore : *Mauro Laurenti*

ID: AN4009-IT

Copyright © 2017 Mauro Laurenti

Come prescritto dall'art. 1, comma 1, della legge 21 maggio 2004 n.128, l'autore avvisa di aver assolto, per la seguente opera dell'ingegno, a tutti gli obblighi della legge 22 Aprile del 1941 n. 633, sulla tutela del diritto d'autore.
Tutti i diritti di questa opera sono riservati. Ogni riproduzione ed ogni altra forma di diffusione al pubblico dell'opera, o parte di essa, senza un'autorizzazione scritta dell'autore, rappresenta una violazione della legge che tutela il diritto d'autore, in particolare non ne è consentito un utilizzo per trarne profitto.

La mancata osservanza della legge 22 Aprile del 1941 n. 633 è perseguibile con la reclusione o sanzione pecuniaria, come descritto al Titolo III, Capo III, Sezione II.
A norma dell'art. 70 è comunque consentito, per scopi di critica o discussione, il riassunto e la citazione, accompagnati dalla menzione del titolo dell'opera e dal nome dell'autore.

AVVERTENZE

I progetti presentati non hanno la marcatura CE, quindi non possono essere utilizzati per scopi commerciali nella Comunità Economica Europea.

Chiunque decida di far uso delle nozioni riportate nella seguente opera o decida di realizzare i circuiti proposti, è tenuto pertanto a prestare la massima attenzione in osservanza alle normative in vigore sulla sicurezza.

L'autore declina ogni responsabilità per eventuali danni causati a persone, animali o cose derivante dall'utilizzo diretto o indiretto del materiale, dei dispositivi o del software presentati nella seguente opera.
Si fa inoltre presente che quanto riportato viene fornito così com'è, a solo scopo didattico e formativo, senza garanzia alcuna della sua correttezza.

L'autore ringrazia anticipatamente per la segnalazione di ogni errore.

Tutti i marchi citati in quest'opera sono dei rispettivi proprietari.

Indice

Introduzione....4
Integrati con supporto USB....4
Configurare il PIC18F4550....5
Configurare il PIC18F14K50....8
Le schede di sviluppo....10
La scheda di sviluppo Freedom III....11
La scheda di sviluppo Freedom II....12
La scheda di sviluppo Freedom Light....13
La scheda di sviluppo EasyUSB....14
La scheda di sviluppo miniCOM USB....15
La libreria Microchip MLA....16
Installare la libreria MLA....16
Organizzazione delle MLA....16
Lo stack USB Microchip....18
Uno sguardo d'insieme....18
Modifiche dello stack USB Microchip....20
Programmare con il Bootloader....21
Impostare il PID e VID....25
Driver USB....26
Driver per la Classe HID....26
Driver per la Classe CDC....26
Esempi di programmazione con la Classe CDC....28
Funzioni disponibili per la Classe CDC....28
Esempio 1: Emulazione di una porta seriale RS232....30
Esempio 2: Controllo dei pulsanti Terminal....33
Esempio 3: Controllo di 8 LED tramite Terminal....37
Esempi di programmazione con la Classe HID....40
Esempio 1: Realizzare una tastiera a due tasti....40
Esempio 2: Controllo di ingressi e uscite digitali e analogiche....43
Bibliografia....50
History....51

Introduzione

Collegare un sistema Emebedded al PC è una di quelle cose che entusiasma tutti gli appassionati di elettronica ed informatica. Negli anni 90 il tutto era facilmente ottenibile con la porta parallela utilizzata per collegare le stampanti al PC. Grazie alle istruzioni IN e OUT, offerte dai linguaggi di programmazione come il BASIC, gestire la porta parallela era questione di pochi minuti. Con gli anni la porta parallela è scomparsa dai PC come anche la porta RS232 utilizzata per i Modem. La scomparsa dei due bus dai Personal Computer è legata all'introduzione del protocollo USB (Universal Serial Bus) introdotto appunto per poter supportare molteplici applicazioni utilizzando un solo bus. L'intento è effettivamente riuscito e dopo oltre dieci anni siamo arrivati alla versione delle specifiche USB 3.1. In questo articolo si presume che il lettore abbia già letto il Tutorial "*Il Protocollo USB*" scaricabile dal sito LaurTec, per cui sono descritti subito gli esempi di programmazione per la classe CDC e HID in modalità Low Speed e Full Speed, fornendo i dettagli di programmazione necessari per adattare lo stack USB fornito dalla Microchip.

Il Tutorial fa riferimento all'ambiente di sviluppo MPLAB X e il compilatore XC8. In particolare si assume che il lettore abbia già familiarità con entrambi. Per maggiori dettagli si faccia riferimento al libro "*XC8 Step by Step*", scaricabile gratuitamente dal sito LaurTec.

Integrati con supporto USB

Sul mercato sono presenti diverse soluzioni che permettono un facile utilizzo del protocollo USB permettendo spesso di dimenticarsi totalmente dei dettagli. In particolare facendo uso della classe CDC si potrebbe anche scrivere un software come se si facesse uso di una porta RS232.
Spesso, nello sviluppo di sistemi con piccoli volumi di vendita, la decisione sull'utilizzo di un integrato o MCU è guidata da precedenti esperienze di utilizzo e/o tempo che si vuole investire. In particolare volendo sviluppare dal lato Host un semplice programma che utilizzi l'interfaccia seriale RS232, può portare alla scelta dell'utilizzo della classe CDC. Questa sarebbe la soluzione più semplice dal lato Host ma richiede l'installazione del file .inf con le relative informazioni del driver da utilizzare. Sebbene questo sia un passo non complicato, la classe HID può semplificare la fase d'installazione del driver ma complica lo sviluppo del software dal lato Host, sebbene siano presenti esempi ad hoc sviluppati da molti fornitori.

La scelta sul come sviluppare il software dal lato Host non è il solo peso sulla bilancia. In particolare anche la scelta sul come sviluppare il sistema Embedded dal lato hardware comporta dei vincoli importanti. Chi non ha esperienza con il protocollo USB ma sa utilizzare già dei microcontrollori, spesso preferisce utilizzare la sola porta UART e collegarsi con degli integrati indipendenti che supportano la classe CDC senza richiedere alcuna programmazione e conoscenza dei dettagli del protocollo USB. Di questi integrati i più famosi sono senza dubbio quelli forniti dalle seguenti società:

- FTDI (Es. FT232H, FT230X, FT231X)
- Texas Instruments (Es. TUSB3410)

- Cypress (Es. CY7C64225)

Recentemente anche la Microchip ha rilasciato un bridge USB-UART basato probabilmente su una versione programmata del PCI18F14Kxx ma che effettivamente dal lato hardware è un bridge USB a tutti gli effetti. Si capisce che l'utilizzo di un Bridge USB è la soluzione più semplice e non richiede una reale conoscenza del protocollo USB.
Qualora dal lato Hardware si voglia utilizzare un microcontrollore (MCU) le cose si complicano ma al tempo stesso si ha maggiore flessibilità visto che con la stessa MCU si possono sviluppare applicazioni per il supporto della classe CDC, HID, Mass Storage Device o altre classi USB, oltre a sistemi compositi. Microcontrollori con PHY e USB Engine sono disponibili in MCU sia ad 8, 16 che 32 bit. Tra i fornitori più noti che forniscono soluzioni complete Hardware e Software (fornendo stack USB gratuiti) si ricordano:

- Microchip (Es. PIC18F4550, PIC18F14K50, ecc.)
- ST (ST32Fxx,ecc.)
- Texas Instruments (MSP430F5xx, TM4C12x, ecc.)

In particolare le stesse società offrono per i propri integrati un programma di sub-licensing per ottenere un VID e PID unico da poter utilizzare per il proprio sistema.

Esclusi i dettagli relativi alla configurazione dei registri dedicati al protocollo, che sono gestiti dallo stack USB fornito dal fornitore della relativa MCU, ogni microcontrollore richiede delle configurazioni speciali dal lato del modulo interno del Clock che devono spesso essere fatte in maniera indipendente dallo stack. In particolare queste configurazioni ed altre specifiche vengono a dipendere anche dal particolare Hardware che circonda il microcontrollore. Nei seguenti paragrafi sono descritti alcuni dettagli dei microcontrollori che verranno descritti in maggior dettaglio negli esempi applicativi.

Configurare il PIC18F4550

Nel seguente paragrafo non si parlerà in dettaglio di ogni registro di configurazione, infatti quasi tutto è gestito dalla libreria Microchip. Ciononostante dal momento che la libreria richiede alcune impostazioni basate sul PIC utilizzato, la scheda di sviluppo e l'applicazione, è bene sapere alcune cose. Alcune delle impostazioni dei registri che seguono possono essere cambiate con opportuni `#define` presenti nella libreria Microchip ma potrebbero anche essere impostati durante la fase di inizializzazione del microcontrollore, assieme all'inizializzazione dell'applicazione principale. Partiamo con l'analizzare la circuiteria di Clock, interna al PIC18F4550 riportata in Figura 1.

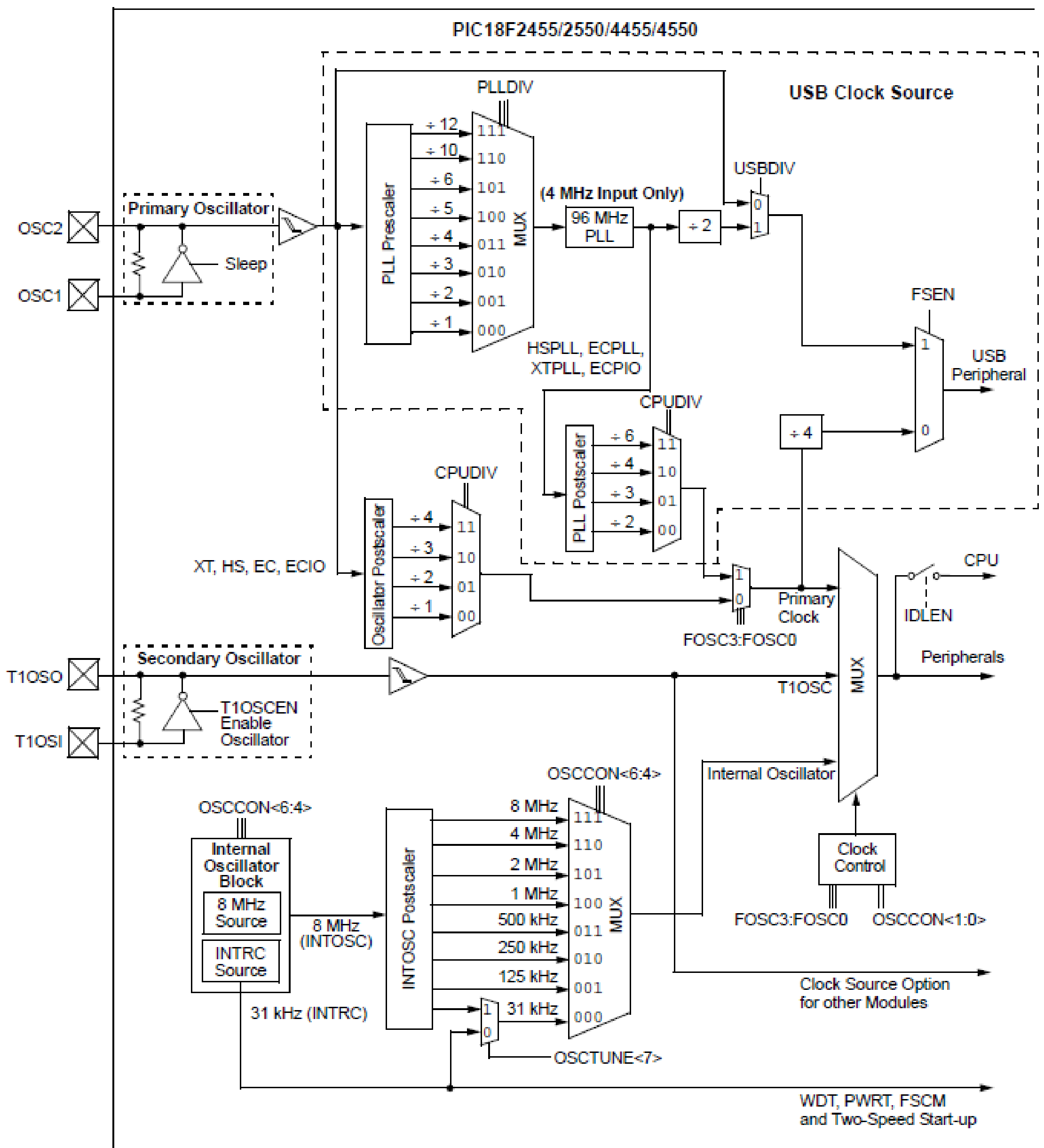


Figura 1: *Schema a blocchi della circuiteria di Clock del PIC18F4550 (prelevata dal Datasheet).*

Il PIC18F4550 possiede il supporto per due oscillatori esterni, ovvero il primario e il secondario; in aggiunta possiede il generatore di clock interno ottenuto con un oscillatore da 8MHz. L'oscillatore primario, qualora venga abilitato, viene dedicato al modulo USB. Ciononostante il Clock delle periferiche può essere derivato dal Clock in uscita dal PLL (Phase Lock Loop) come anche dall'oscillatore secondario o quello interno. Il PIC18F4550 supporta le specifiche USB 1.0 e USB 2.0 ed in particolare la modalità Low Speed e Full Speed. Qualora si voglia far uso della sola modalità Low Speed è necessario che il quarzo esterno sia di 6MHz. Se invece si vuol far uso della modalità Full Speed è possibile, per il quarzo esterno, utilizzare una più ampia scelta di valori. In particolare il valore del quarzo scelto deve essere tale per cui, diviso per uno dei valori selezionati per mezzo del multiplexer PLLDIV, (1, 2, 3, 4, 5, 6, 10, 12) il risultato sia di 4MHz. Questo è necessario poiché il PLL che segue il multiplexer, ottiene 96MHz a partire da 4MHz di

riferimento. Il Clock di 96MHz diviso 2, ovvero 48MHz viene utilizzato dal modulo USB. Il Clock da 96MHz, come detto, può essere utilizzato per generare il Clock delle periferiche, questo viene però ottenuto per mezzo di una ulteriore divisione, selezionabile tra i valori 2, 3, 4, 6 per mezzo del mux CPUDIV; questo significa che la CPU può avere un Clock massimo di 48MHz. Utilizzando questa frequenza vi renderete conto che il PIC sarà un po' più caldo del normale, ma tutto entro le specifiche. Le impostazioni ora descritte devono essere fatte per mezzo dei registri di configurazione, ovvero facendo uso della direttiva `#pragma`. Facendo uso della libreria LTlib tali configurazione vanno impostate nel file di configurazione associato al PIC. Un esempio di configurazione è :

```
#pragma config FOSC = HSPLL_HS
#pragma config PLLDIV = 5
#pragma config CPUDIV = OSC1_PLL2
```

In questa impostazione si è abilitata la modalità HS con PLL ovvero con modulo USB. Il registro PLLDIV è impostato a 5 poiché si è supposto che il quarzo utilizzato sia di 20MHz. Impostando la divisione a 5 si ha che il PLL ha al suo ingresso i 4MHz necessari per il suo corretto funzionamento.
Il Clock delle periferiche è anche ottenuto dal Clock primario, in particolare dal Clock in uscita dal PLL, il cui valore è però diviso per 2. Le impostazioni che è possibile utilizzare per i vari parametri sono riportate nel file html 18f4550.html presente nella directory:

```
[...]\Microchip\xc8\v1.41\docs\chips
```

della directory d'installazione del compilatore XC8.

Oltre al Clock, un'altra impostazione importante riguarda il regolatore lineare interno presente nel PIC18F4550; questo viene utilizzato per generare i 3.3V necessari per i resistori di pull-up che definiscono la modalità Low Speed e Full Speed[1]. In particolare i resistori, dovendo essere collegati tra una delle linee dati e 3.3V, possono essere esterni al PIC, come anche il regolatore di 3.3V. Dal momento che il PIC18F4550 possiede al suo interno sia il regolatore che i resistori, perché non usarli? Le schede di sviluppo LaurTec sfruttano il regolatore lineare e i resistori interni, dunque è necessario abilitarli; il regolatore interno viene abilitato per mezzo della direttiva `#pragma`, scrivendo:

```
#pragma config VREGEN = ON
```

Le configurazioni ora descritte, ovvero facente uso della direttiva `#pragma`, vanno propriamente impostate anche facendo uso della libreria che verrà introdotta a breve. I parametri che verranno di seguito descritti sono invece gestiti dalla libreria per mezzo di costanti che starà a noi impostare. Questo diverso metodo di gestione è legato al fatto che questi secondi parametri sono contenuti in registri che è possibile facilmente cambiare durante l'esecuzione del programma.

Il registro da impostare, appena lasciato in sospeso, è quello relativo ai resistori interni di pull-up. Questo viene fatto impostando ad 1 il bit UPUEN del registro UCFG, ovvero:

[1] Si ricorda che i resistori di pull-up utilizzati sono del valore 1.5KΩ ±5%, in particolare la modalità Low Speed è segnalata collegando il resistore di pull-up tra la linea D- e +3.3V, mentre la modalità Full Speed viene identificata per mezzo di un resistore di pull-up collegato tra la linea D+ e 3.3V.

```
// Abilito i resistori di pull-up sulle linee dati USB
UCFGbits.UPUEN = 0x01;
```

Come detto questa riga di codice non sarà necessaria poiché gestita dalla libreria. Altri registri da impostare per attivare il transceiver e selezionare la relativa modalità USB sono:

```
// Abilita il transceiver interno (normalmente già attivo)
UCFGbits.UTRDIS = 0x01;

// 1 Abilita la modalità Full Speed, 0 modalità Low Speed
UCFGbits.FSEN = 0x01;

// Abilita il modulo USB
UCONbits.USBEN = 0x01;
```

In seguito sono forniti i dettagli relativi al punto della libreria dove apportare le modifiche necessarie.

Configurare il PIC18F14K50

Il PIC18F14K50 possiede come il PICK18F4550 un modulo SIE (Serial Interface Engine) per il supporto del protocollo USB. In particolare possiede al suo interno sia il transceiver che l'LDO per generare i 3.3V necessari per i resistori di pull-up per definire la modalità di comunicazione Low Speed o Full Speed. Anche i resistori di pull-up sono interni al microcontrollore. Per poter propriamente utilizzare il modulo SIE bisogna rispettivamente fornire 6MHz se si fa uso della modalità Low Speed e 48MHz se si fa uso della modalità Full Speed. In Figura 2 è riportato lo schema a blocchi del modulo Clock interno. Il PLL interno ha un fattore moltiplicativo 4x, dunque permette di raggiungere i 48MHz a partire da un quarzo di 12MHz collegato all'oscillatore primario. A seconda se il PLL sia abilitato o meno per mezzo del bit PLLEN si può anche scegliere di usare un cristallo di 48MHz.
Allo stesso modo per la modalità Low Speed è possibile utilizzare sia un cristallo da 6MHz che un cristallo da 12MHz. Facendo uso di un cristallo da 12MHz si può usare sia la modalità Low Speed (usando la divisione per 2 fornita dall'opzione USBDIV) che la modalità Full Speed, facendo uso del PLL interno. Rispetto al PCI18F4550 il PIC18F14K50 offre meno impostazioni per poter utilizzare cristalli a frequenza differente.

Il PIC18F14K50 possiede al suo interno anche un oscillatore impostabile a varie frequenze comprese tra 31KHz e 16MHz. Tale oscillatore non può essere utilizzato per il modulo USB sia per i valori forniti che l'accuratezza dello stesso. Come anche altri PIC18 il PIC18F14K50 rende disponibile anche un oscillatore secondario utilizzabile per esempio con un cristallo esterno da 32KHz, utile per realizzare un RTC (*Real Time Clock Calendar*).

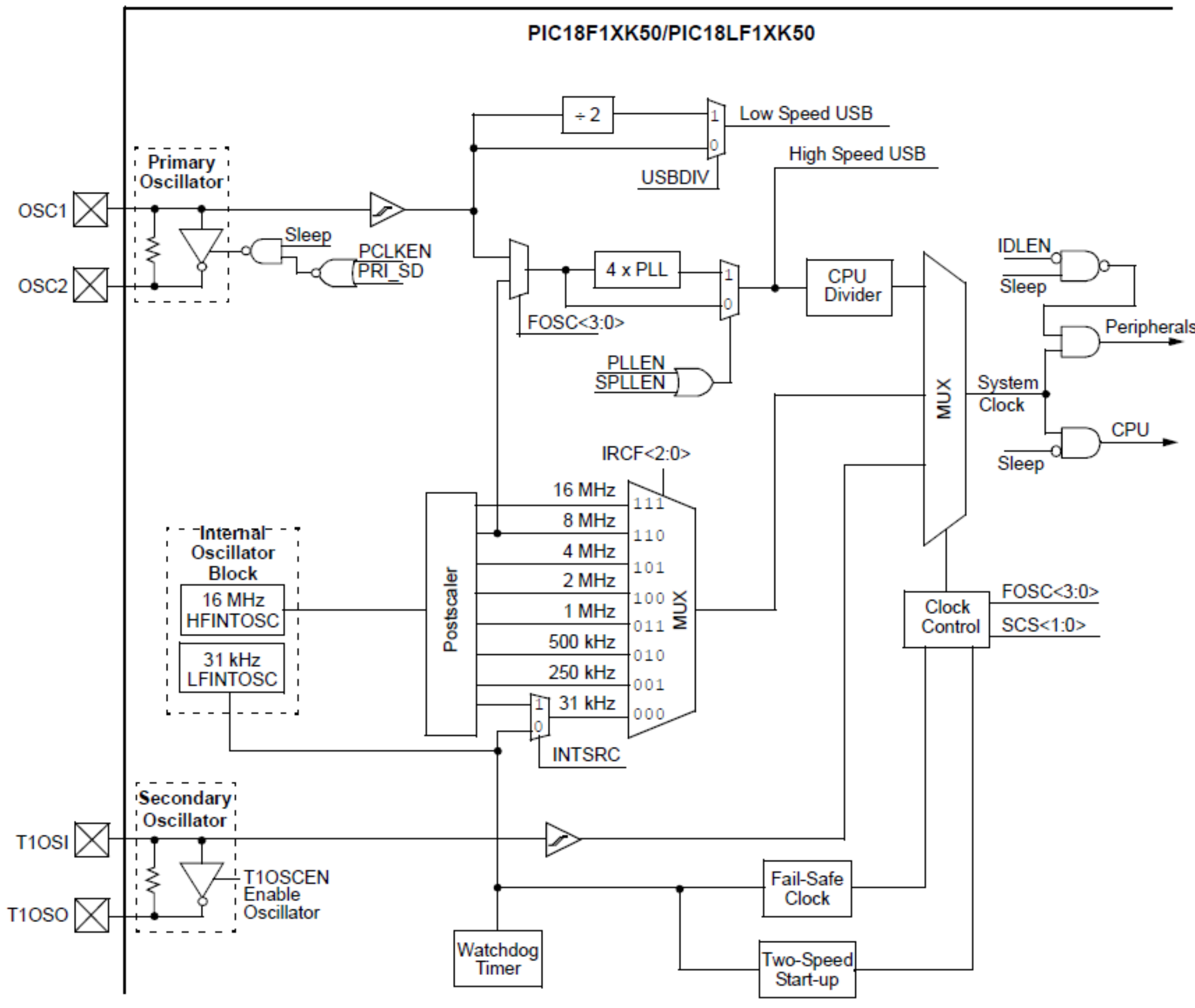


Figura 2: *Schema a blocchi della circuiteria di Clock del PIC18F14K50 (prelevata dal Datasheet).*

Le schede di sviluppo

Una volta presa la decisione su quale integrato (PHY) o MCU con porta USB utilizzare, la complessità del protocollo USB pone ancora l'esigenza di dividere lo studio e lo sviluppo di applicazioni Embedded in due problematiche. Realizzare un hardware di sviluppo e ottenere un pacchetto software per la gestione del protocollo USB. Infatti entrare nei dettagli, come avrete ben capito, non ha spesso ragione di essere, dal momento che è più importante focalizzare il proprio tempo nello sviluppo dell'applicazione e non nei singoli dettagli del protocollo USB. Dal lato USB LaurTec fornisce diverse schede di sviluppo che permettono lo sviluppo di applicazioni USB. Gli schemi elettrici e relativa documentazione sono disponibili per un download gratuito per cui possono essere prese come riferimento per lo sviluppo delle proprie applicazioni. Dal momento che le varie schede richiedono impostazioni diverse per poter utilizzare il protocollo USB, si riporta di seguito un riassunto delle varie schede disponibili e relativi integrati utilizzati.

In particolare gli esempi mostrati fanno anche uso di pulsanti e LED che devono essere opportunamente abilitati sulle varie schede. Per maggiori dettagli sulle schede di sviluppo si rimanda ai relativi Manuali Utente.

Sebbene i paragrafi che seguono mostrino una serie di schede di sviluppo, realizzare un proprio mini sistema su una scheda mille fori non è particolarmente complicato. Ciononostante, se si è alle prime armi, inserire nei propri esperimenti troppe variabili potrebbe complicare il lavoro e perdersi in problemi sia Hardware che Software.

La scheda di sviluppo Freedom III

La scheda di sviluppo Freedom III mostrata in Figura 3, permette di utilizzare il protocollo USB per mezzo del bridge USB-UART fornito dal PIC18F14K50 già programmato con lo stack USB con classe CDC. Per cui il microcontrollore principale PIC18F46K22, senza porta USB, può utilizzare il bridge USB-UART semplicemente facendo uso del modulo EUSART 1 interno alla MCU.

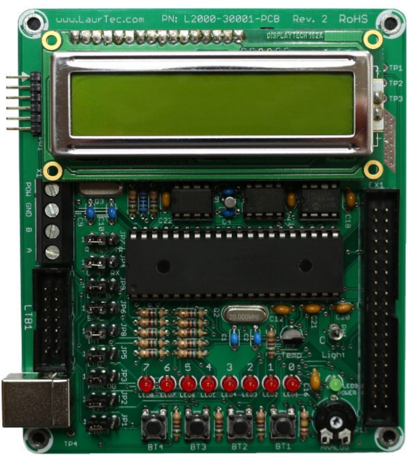


Figura 3: *Sistema di sviluppo Freedom III.*

Il bit rate utilizzato dal modulo UART deve riflettere quello utilizzato dall'Host. Il PIC18F14K50 si adatta all'Host ma il PIC18F46K22 utilizza un bit rate costante che deve essere noto a priori.
Per la posizione dei Jumper, a meno di non voler usare altre configurazioni, si può utilizzare la configurazione generica con tutto attivo:

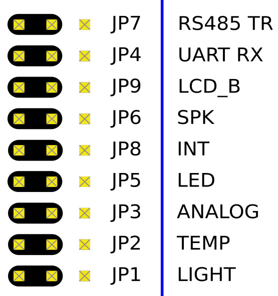


Figura 4: *Posizione dei Jumper per la scheda Freedom III.*

Il PIC18F14K50 potrebbe essere programmato anche con lo stack USB e classe HID per cui si possono sperimentare applicazioni anche con classe HID. L'interfaccia PIC18F14K50 e la MCU principale PIC18F46K22 avviene comunque per mezzo del modulo EUART 1.

La scheda di sviluppo Freedom II

La scheda di sviluppo Freedom II, riportata in Figura 5, permette di utilizzare il protocollo USB direttamente con la MCU principale, visto che non è presente alcun bridge USB-UART. Per tale ragione è necessario che la MCU principale sia il PIC18F4550.

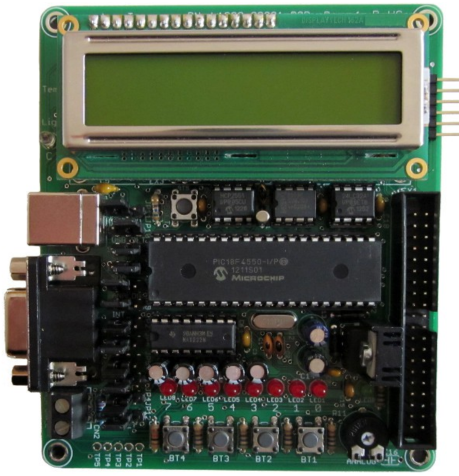


Figura 5: *Sistema di sviluppo Freedom II.*

Per poter utilizzare propriamente l'USB è necessario impostare alcuni Jumper visto che la scheda Freedom II è stata progettata per poter ospitare MCU diverse dal PIC18F4550. In particolare i Jumper che è necessario impostare sono JP7 (SDA), JP8 (SCL) e JP11 (USB_CP). L'impostazione Full Speed e Low Speed deve essere fatta per mezzo dei resistori interni al PIC, infatti Freedom II non possiede i resistori esterni, altrimenti necessari. Il tasto BT1 della scheda possiede diversamente dagli altri un resistore di pull-up al fine di supportare i bootloader Microchip senza alcuna modifica. In Figura 6 sono riportate le impostazioni base per i Jumper JP7 (SDA), JP8 (SCL) e JP11 (USB_CP).

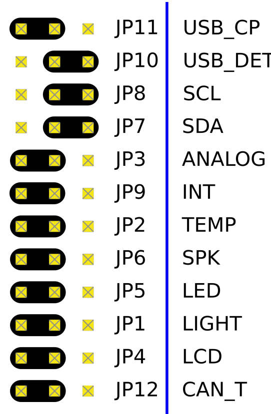


Figura 6: *Posizione dei Jumper per la scheda Freedom II.*

I Jumper JP9 (INT) e JP10 (USB_DET) vanno collegati a seconda dell'applicazione specifica.

La scheda di sviluppo Freedom Light

La scheda di sviluppo Freedom Light riportata in Figura 7, è compatibile con la scheda Microchip PICDEMTM FS USB per cui è possibile eseguire i vari esempi per PIC18F4550 presentati dalla Microchip stessa senza dover apportare alcun cambiamento.

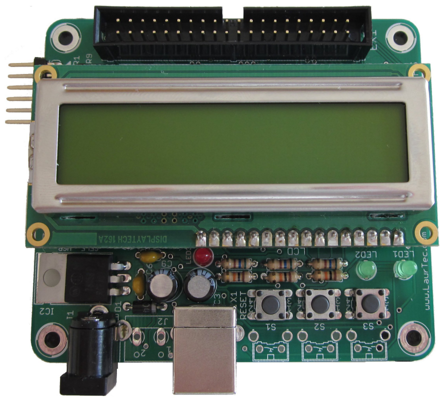


Figura 7: *Scheda di sviluppo Freedom Light.*

Ciononostante gli esempi che fanno uso del sensore di temperatura e della porta seriale non possono essere eseguiti senza l'aggiunta di Hardware esterno. La ragione è semplicemente legata al fatto che la scheda di sviluppo Freedom Light è stata pensata per poter essere utilizzata in applicazioni generiche per cui l'hardware non sempre utilizzato è stato eliminato. I Juper J1-J6 devono essere collegati come riportato in Figura 3.

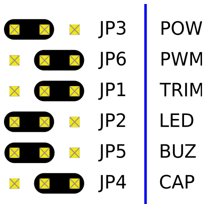


Figura 8: *Posizione dei Jumper per la scheda Freedom Light.*

La scheda di sviluppo EasyUSB

La scheda di sviluppo EasyUSB riportata in Figura 10, è compatibile con la scheda Microchip PICDEMTM FS USB per cui è possibile eseguire i vari esempi per PIC18F4550 presentati dalla Microchip stessa senza dover apportare alcun cambiamento.

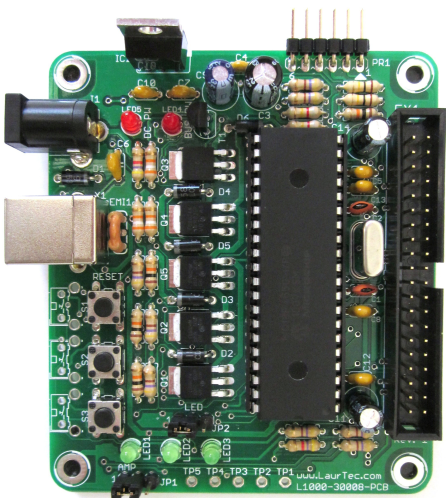


Figura 9: *Scheda di sviluppo EasyUSB.*

Ciononostante gli esempi che fanno uso del sensore di temperatura, del trimmer e della porta seriale non possono essere eseguiti senza l'aggiunta di hardware esterno.

La scheda di sviluppo EasyUSB fornisce diverse opzioni per il Power Oring. Questa flessibilità, sebbene lo stack Microchip possa essere usato senza problemi, richiede l'opportuna gestione dei diversi MOS presenti sulla scheda.

La scheda di sviluppo miniCOM USB

La scheda di sviluppo miniCOM USB è basata sul PIC18F14K50, lo stesso montato come adattatore USB-UART sulla scheda Freedom III. Le dimensioni ridotte e i connettori di espansione la rendono idonea per semplici applicazioni basate su USB. L'Hardware montato sulla scheda è semplicemente di due LED e due Pulsanti, rendendo dunque i costi ridotti.

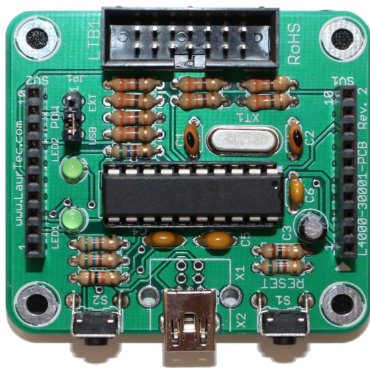


Figura 10: *Scheda di sviluppo miniCOM USB.*

La scheda possiede un solo Jumper che permette di selezionare se l'alimentazione debba essere prelevata direttamente dal bus USB o dal connettore di espansione LTB. Il connettore di espansione LTB è lo stesso presente anche sulla scheda Freedom III.
Della serie di schede miniCOM fanno anche parte la scheda miniCOM Relay, miniCOM RS232 e miniCOM Extension Board che permette di impilare la scheda e poter saldare componenti aggiuntivi come mostrato in Figura 11.

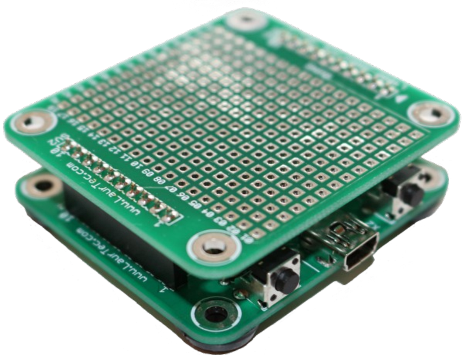


Figura 11: *Scheda di espansione miniCOM USB.*

La libreria Microchip MLA

Sebbene il numero dei registri interni presenti nei microcontrollori che supportano il protocollo USB non sia impossibile da gestire, dover leggere tutte le specifiche del protocollo USB e gestire in maniera corretta i vari segnali e tempistiche richieste dallo stesso, potrebbe richiedere molto tempo. Non è un caso che ogni azienda che fornisce microcontrollori o USB bridge fornisca anche tutto il software e driver necessari al progettista per iniziare nel minor tempo possibile. In questo modo si fornisce la possibilità di poter selezionare il proprio chip senza dover considerare il protocollo USB un ostacolo alla progettazione, alleggerendo l'utilizzo del protocollo USB. Questo non significa che tutto sia semplice, ma certamente il progettista di un'applicazione Embedded non deve essere un esperto del protocollo USB come lo deve invece essere colui che deve sviluppare lo stack USB.

Installare la libreria MLA

La Microchip, come tutte le società che producono microcontrollori con supporto USB, fornisce lo stack USB pronto per poter essere utilizzato. Lo stack è inoltre accompagnato da numerosi esempi che possono essere utilizzati come punto di partenza per le proprie applicazioni. Lo stack USB e gli esempi fanno parte delle MLA ovvero *Microchip Libraries for Applications*, che oltre allo stack USB contengono anche lo stack Ethernet, librerie grafiche ed altro. La libreria è fornita gratuitamente dalla Microchip ed è scaricabile dal link:

```
www.microchip.com/mplab/microchip-libraries-for-applications
```

La sua installazione è piuttosto semplice visto che è fornita come package da installare. A scopo precauzionale, quando si installa software di questo tipo è bene mantenere il percorso d'installazione di default visto che è quello effettivamente testato per gli esempi.

La libreria è disponibile sia per ambiente Windows, Linux e Mac. Negli esempi riportati nel Tutorial si fa riferimento all'ambiente di sviluppo Windows.

Organizzazione delle MLA

La libreria MLA viene installata nel percorso di Default:

```
C:\microchip\mla\v2016_11_07
```

Il nome della cartella finale fa riferimento alla versione della libreria stessa, il che permette di installare versioni multiple senza dover cancellare le precedenti. I vari esempi per lo standard USB scritti in ambiente MPLAB X sono contenuti nella cartella:

```
C:\microchip\mla\v2016_11_07\apps\usb
```

In particolare tale cartella contiene sia la sotto-cartella Host che Device. Per i

microcontrollori della serie PIC18 viene presa in considerazione solo la cartella Device, ma molte considerazioni possono essere utilizzate anche per gli esempi Host, sebbene basati sui microcontrollori della Microchip ad architettura 16 e 32 bit.

Per aprire i vari progetti, basta aprire l'ambiente di sviluppo MPLAB X e tramite la funzione *Open Project*, andare nella cartella Device discussa sopra. In particolare in tale cartella è presente un insieme di esempi basati su classi USB differenti, come mostrato in Figura 12:

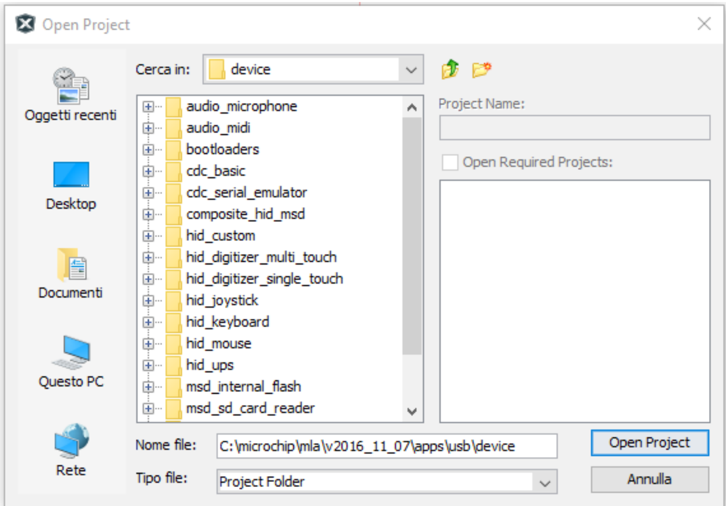


Figura 12: *Lista degli esempi USB.*

Dalla lista degli esempi è possibile subito notare che sono presenti progetti base relativi alle periferiche più comuni, per cui basta selezionare quello più vicino al proprio caso/progetto, per poter iniziare a lavorare. Si consiglia di duplicare la cartella del progetto d'interesse senza modificare il progetto originale, copiando la nuova cartella direttamente nel percorso degli esempi e cambiare il relativo nome con quello del proprio progetto. In questo modo si ha sempre la possibilità di creare nuovi progetti partendo da quelli originali della Microchip.
Una volta selezionata la cartella del progetto d'interesse troverete all'interno ulteriori progetti .x relativi ai vari microcontrollori per i quali sono stati sviluppati gli esempi. Nel caso del PIC18F4550 dovrete selezionare la scheda di sviluppo PICDEM FS USB (Es. picdem_fs_usb.x) o picdem_fs_usb_k50 per il PIC18F14K50.
Diversi esempi della microchip relativi alla Classe HID e Classe CDC possono essere caricati nelle schede di sviluppo LaurTec senza alcuna modifica. Come spiegato in seguito, volendo poi sviluppare una propria applicazione commerciale, è necessario cambiare il VID e PID dell'applicazione.

Quanto riportato nel Tutorial potrebbe cambiare al variare della versione dello Stack. Infatti negli anni lo Stack ha subito diverse modifiche ed aggiornamenti che ha portato, oltre all'introduzione di nuove funzioni, anche alla riorganizzazione di molti file e cambio di nome di funzioni.

Lo stack USB Microchip

Come abbiamo appena visto la libreria MLA fornisce molti esempi ai quali bisogna apportare poche modifiche al fine di poterli usare. In alcuni casi si potrebbero usare anche cosi come sono e modificare solo il VID e PID dell'applicazione. Nella maggioranza dei casi però si potrebbe comunque avere l'esigenza di cambiare qualche parte del programma per meglio adattarla alle proprie esigenze. Quanto segue si applica allo stack USB in generale, indipendentemente dalla classe USB utilizzata, sebbene però ci possano essere piccole variazioni.

Uno sguardo d'insieme

Lo stack USB della Microchip può essere gestito sia in polling che per mezzo delle interruzioni, in particolare, di default molti programmi fanno uso del polling. Per selezionare una modalità o l'altra bisogna commentare la riga non di interesse contenuta nel file usb_config.h:

```
#define USB_POLLING
//#define USB_INTERRUPT
```

Per poter accedere al file usb_config.h bisogna accedere la cartella Header Files nel Tab Projects dell'esplora risorse.

quando si fa uso del polling bisogna richiamare ciclicamente la funzione:

```
USBDeviceTasks();
```

Tale funzione deve essere richiamata con un periodo inferiore a 1.8ms al fine di garantire la corretta gestione dei pacchetti dati USB provenienti dall'Host ed evitare errori come l'overflow dei buffer. I programmi di esempio contengono la chiamata a tale funzione all'interno di un ciclo `while` infinito contenuto nella funzione `main`. Da quanto appena detto si capisce che scrivere un programma all'interno della funzione `main`, al fine di arricchire le funzioni software dell'applicazione, richiede una certa attenzione. In particolare non si possono usare cicli delay bloccanti che impedirebbero l'aggiornamento dello stack o il controllo ricorsivo dei Flag delle periferiche. Il modo migliore di operare e aggiungere nuove funzioni sarebbe per mezzo di state machine. Lo stack stesso possiede diverse state machine il cui stato viene aggiornato all'interno della funzione `USBDeviceTasks`, da cui si capisce anche l'importanza di doverla chiamare in maniera frequente. Gli esempi presentati dalla Microchip, all'interno del ciclo `while` dove viene richiamata la funzione `USBDeviceTasks`, viene eseguita ciclicamente anche la funzione relativa all'applicazione stessa. Per esempio per l'applicazione del Mouse viene richiamata la funzione `APP_DeviceMouseTasks`. Qualora si volesse modificare l'applicazione, si potrebbe modificare la funzione Task richiamata nel ciclo `while` o semplicemente scrivere il proprio programma all'interno del ciclo `while` stesso. Per semplici esempi utilizzerò anche questo approccio anche se non è consigliato per applicazioni più complesse.

Una variabile di sistema che è importante conoscere è:

```
USBDeviceState
```

Nelle ultime versioni dello stack è possibile trovarla con il nome della macro:

```
USBGetDeviceState()
```

infatti è definita come:

```
#define USBGetDeviceState() USBDeviceState
```

Tale variabile viene aggiornata in funzione dello stato di configurazione del sistema. I valori che assume questa variabile sono:

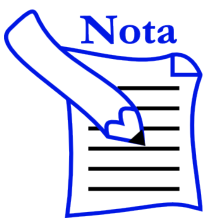
POWERED_STATE
Il cavo USB è collegato ma il sistema non è stato configurato dall'Host.

DEFAULT_STATE
Il sistema è in stato di Default per permettere la comunicazione da parte dell'Host.

ADDRESS_STATE
L'Host ha iniziato la comunicazione con il sistema per poterlo configurare e al tempo stesso configurare il sistema operativo. Questa è anche la fase in cui Windows® avvia la sequenza di caricamento del driver.

CONFIGURED_STATE
Il sistema è propriamente configurato e il sistema operativo ha caricato il driver opportuno.

Per poter leggere la variabile `USBDeviceState` si procede allo stesso modo con cui si farebbe con una variabile normale, dunque si può fare un controllo per mezzo dell'istruzione `if`. E' bene ricordare che non bisogna utilizzare l'istruzione `while` al fine di attendere un cambio di stato. Questa procedura sarebbe infatti di tipo bloccante, dunque la variabile non verrebbe mai aggiornata dal servizio di aggiornamento della libreria, causando dunque lo stallo del modulo USB.



Oltre agli stati sopra descritti, sono anche presenti: DETACHED_STATE e ATTACHED_STATE che però non devono essere utilizzati su Freedom II e Freedom III poiché le schede non supportano l'alimentazione da USB. Tali stati sono supportati invece da EasyUSB. In particolare queste due costanti indicano quando il cavo USB è collegato e scollegato.

Altre funzioni specifiche verranno trattate negli esempi che seguiranno.

Modifiche dello stack USB Microchip

Lo stack USB è predisposto già per diverse modifiche al fine di poter esser facilmente adattato a diverse esigenze progettuali. In particolare molte delle modifiche che è possibile apportare consistono semplicemente nel commentare o meno un determinato `#define` al fine di permettere una compilazione mirata di determinate funzioni o influenzare eventuali test condizionali. Di seguito sono riportate alcune delle modifiche che può essere necessario effettuare al fine di poter adattare lo stack alle varie schede di sviluppo.

Modifica 1

Decidere se usare la modalità Polling o la modalità Interrupt commentando il relativo `#define` non di interesse:

```
#define USB_POLLING
//#define USB_INTERRUPT
```

Il `#define` lo si trova nella cartella usb del progetto, file usb_config.h

Modifica 2

Decidere se usare la modalità Full Speed o Low Speed commentando il relativo `#define`:

```
#define USB_SPEED_OPTION USB_FULL_SPEED
//#define USB_SPEED_OPTION USB_LOW_SPEED
```

Il `#define` lo si trova nella cartella usb del progetto, file usb_config.h. A meno di non avere esigenze particolari si può impostare la modalità Full Speed.

Modifica 3

A seconda dell'Hardware utilizzato può essere necessario abilitare i resistori di pull-up interni al microcontrollore al fine di impostare la relativa modalità Low Speed o Full Speed. In particolare per le schede LaurTec devono essere abilitati. Tale opzione è anche quella di default:

```
#define USB_PULLUP_OPTION USB_PULLUP_ENABLE
//#define USB_PULLUP_OPTION USB_PULLUP_DISABLED
```

Il `#define` lo si trova nella cartella usb del progetto, file usb_config.h

Modifica 4

Il modulo del clock deve essere propriamente impostato al fine di avere il clock giusto per far funzionare il modulo USB. La libreria è già impostata per lavorare con un quarzo di 20MHz ed avere un clock per le periferiche direttamente dal clock del PLL. Le impostazioni presenti nel file system.c per il PIC18F4550 sono:

```
#pragma config PLLDIV   = 5          // 20 MHz crystal
#pragma config CPUDIV   = OSC1_PLL2
#pragma config USBDIV   = 2          // Clock source from 96MHz PLL/2
#pragma config FOSC     = HSPLL_HS
```

Se si dovesse adattare il programma per microcontrollori differenti dal PIC18F4550 è necessario cambiare le relative configurazioni. In particolare si ricorda che in base alla frequenza del quarzo è necessario impostare in maniera opportuna il divisore al fine di avere in ingresso al PLL la frequenza richiesta dallo stesso. Tale frequenza, pari a 4MHz per il PIC18F4550, potrebbe essere differente per altri modelli.

Modifica 5

Oltre alla circuiteria di clock è necessario abilitare il regolatore interno al PIC18F4550, in maniera da permettere il corretto funzionamento del modulo USB. La libreria attuale possiede il regolatore già abilitato, dunque non è necessario fare alcuna modifica, ma accertarsi di avere la seguente riga di codice su ON:

```
#pragma config VREGEN    = ON       //USB Voltage Regulator
```

A questo punto, con le poche modifiche descritte, qualora necessarie, la libreria è già pronta per operare con le schede di sviluppo LaurTec o simili. Naturalmente oltre alle modifiche ora apportate, il progetto può essere ripulito da tutto quel codice che non deve essere utilizzato. La pulizia della libreria sarà cosa facile una volta presa esperienza con la stessa, in ogni modo si consiglia di compilare il progetto ad ogni modifica apportata e testarlo nuovamente sulla scheda di sviluppo. In questo modo eviterete di eliminare parti di codice che vi sono sembrate “di troppo”. Personalmente non ho l'abitudine di ripulire il codice in maniera tale da poter riportare facilmente ogni programma nelle versioni più aggiornate della libreria senza dover ripulire ogni volta il codice.

Programmare con il Bootloader

Aggiungere la porta USB al proprio sistema Embedded non fornisce solamente un ottimo canale di comunicazione con il PC ma anche un ottimo modo per aggiornare il Firmware della propria applicazione. Questa opportunità, qualora si abbia la porta USB già a disposizione è bene sempre tenerla disponibile, infatti i programmi non saranno quasi mai esenti da problemi anche se effettueremo test rigorosi. Ciononostante questa opzione non viene fornita gratuitamente al prezzo di una semplice opzione da attivare. Per permettere l'aggiornamento del Firmware della nostra applicazione via USB è necessario programmare nel microcontrollore un bootloader ovvero un programma che viene avviato ad ogni accensione del sistema e controlla determinate condizioni di avvio. Se vengono verificate, avvia l'aggiornamento del Firmware, altrimenti avvia il programma principale. Si capisce che normalmente, non volendo aggiornare il Firmware ad ogni accensione, gran parte delle volte il bootloader avvierà il programma principale. Quando viene avviato il programma principale, il bootloader è quasi come se non fosse presente, infatti tutte le risorse del microcontrollore sono disponibili per la nostra applicazione. L'aver però detto “quasi come” fa ben intendere che qualcosa del bootloader bisogna tener a mente. In effetti, sebbene le risorse siano effettivamente tutte disponibili, la nostra applicazione (Firmware) deve essere opportunamente compilata affinché possa essere supportato il bootloader. Infatti il bootloader viene compilato come un qualunque programma, per cui il linker lo posiziona in maniera tale che la prima istruzione del bootloader coincida con il Reset Vector. Oltre a questo gli Interrupt Vector ad alta e bassa priorità posizionati all'indirizzo 0x08 e 0x016 sono assegnati al bootloader. Questo significa che la nostra applicazione non può utilizzarli. Considerando che il bootloader

occupa inoltre un certo spazio in memoria, generalmente inferiore a 0x1000 byte, tale spazio non deve essere utilizzato dall'applicazione principale. Riassumendo il bootloader, pur liberando le risorse nel momento in cui non è in esecuzione, impone i seguenti limiti:

- Gli Interrupt Vector devono essere liberati.
- Il Reset Vector deve essere spostato
- Lo spazio di memoria flash usata dal bootloader deve essere mantenuta libera.

Oltre ai vincoli sopra citati è importante tenere a mente che le configurazioni usate per la propria applicazione non creino conflitto con il bootloader. Conflitti tipici potrebbero essere associati al modulo del clock o utilizzo del pin di Reset. Eventuali conflitti di configurazione potrebbero non far funzionare correttamente il bootloader impedendo di aggiornare il Firmware.

Il bootloader può essere compilato come un qualunque programma mentre l'applicazione deve essere compilata modificando alcuni parametri del linker, al fine di spostare gli Interrupt Vector e il Reset Vector, e mantenere libera la memoria flash usata già dal bootloader. Al fine di evitare che il linker faccia uso della zona di memoria Flash occupata dal bootloader, bisogna impostare il parametro *ROM Ranges* tra le configurazioni del progetto facendo uso del seguente range:

```
ROM ranges: default,-0-FFF,-1006-1007,-1016-1017
```

come mostrato in Figura 13 .

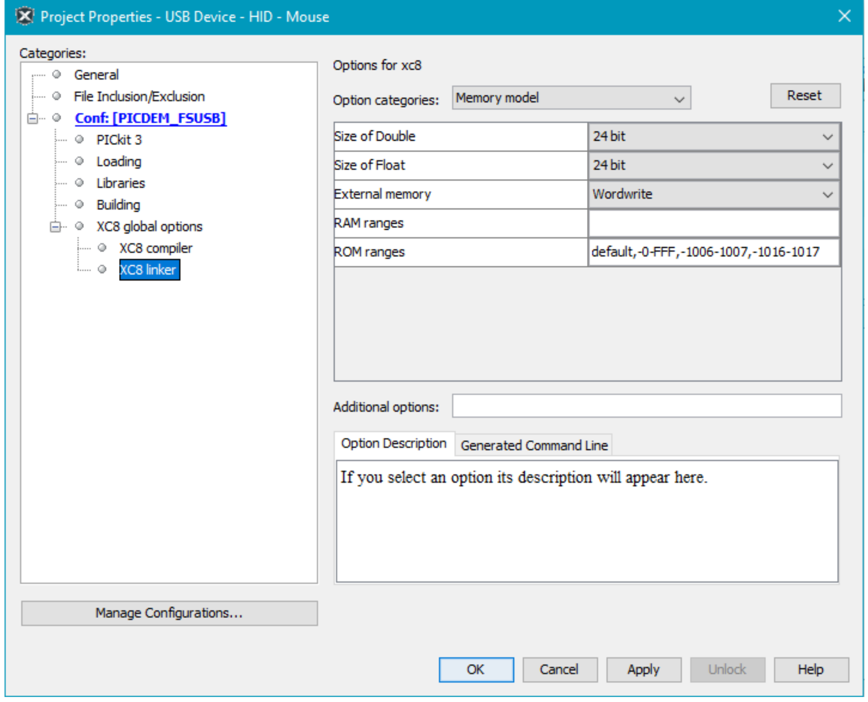


Figura 13: *Offset di 0x1000 per spostare gli Interrupt Vector e il Reset Vector.*

In particolare tale impostazione informa il linker di utilizzare la memoria di default escludendo i range di memoria del bootloader e dei nuovi Interrupt Vector.

Oltre a queste impostazioni bisogna anche informare il Linker di traslare tutto il codice, inclusi gli Interrupt Vector 0x08 e 0x18. Questo viene fatto per mezzo del parametro Codeoffset da impostare a 0x1000, come mostrato in Figura 14.

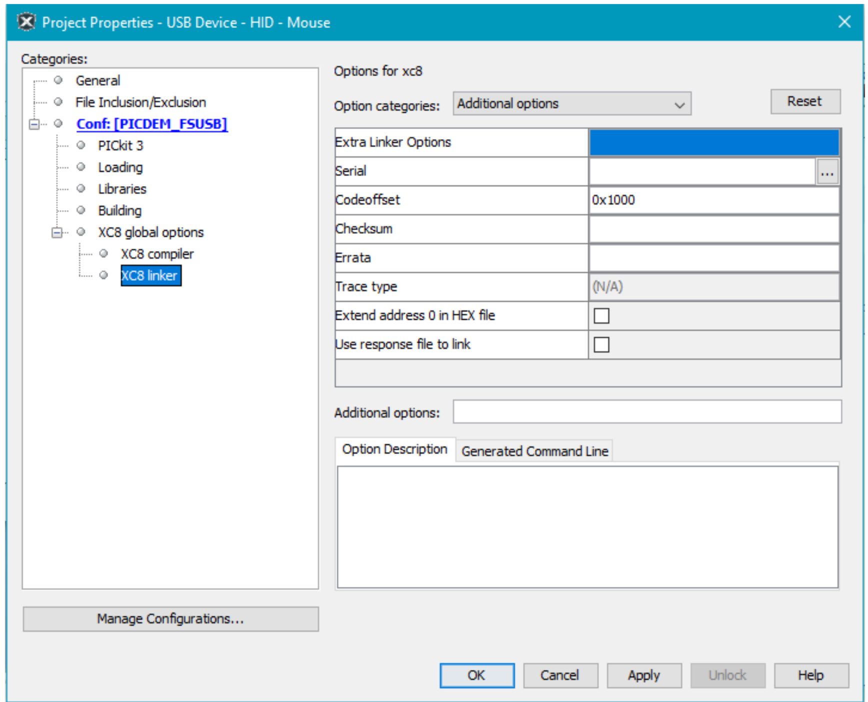


Figura 14: *Impostazione code offset.*

Per verificare se le impostazioni appena descritte hanno effetto, si può compilare il programma e vedere il file .hex. In particolare vedendo gli indirizzi di inizio del programma si noterà, nel caso di utilizzo del bootloader, di un salto alla locazione di memoria 0x1000.

Come detto il bootloader è la prima applicazione ad essere eseguita all'avvio del microcontrollore e controlla se sono verificate le condizioni di aggiornamento del Firmware altrimenti viene avviato il programma principale. Le condizioni di avvio sono specifiche per l'applicazione e possono essere cambiate secondo le proprie esigenze. Per gli integrati di riferimento utilizzati nel seguente Tutorial si ha:

PIC18F4550

Il bootloader della Microchip fa uso del controllo del pin PORTB bit 4. Se il pin viene trovato a massa il bootloader entra in modalità aggiornamento Firmware, mentre se trovato a 1, viene avviato il programma principale. La condizione di avvio del bootloader è il tasto sul pin RB4 che deve essere premuto prima di alimentare il Microcontrollore o premere il tasto di Reset di riavvio. Quando l'alimentazione è disponibile, o si è rilasciato il tasto di Reset, è possibile rilasciare il tasto sul pin RB4.

PIC18F14K50

Il bootloader della Microchip disattiva tramite le configurazioni del microcontrollore il pin di Reset, utilizzandolo come standard I/O. La condizione di avvio del bootloader è il tasto di Reset da premere prima di fornire l'alimentazione al Microcontrollore. Quando l'alimentazione è disponibile è possibile rilasciare il tasto di Reset.

Le schede di sviluppo LaurTec basate sul PIC18F4550 e PIC18F14K50 possiedono un pulsante con resistore di pull-up sul pin RB4 e di Reset, supportando i relativi bootloader senza richiederne alcuna modifica.

Dopo aver compreso che il bootloader impone dei vincoli di compilazione sull'applicazione principale vediamo qualche dettaglio in più sulla sequenza di programmazione.

Considerando un microcontrollore vuoto, ovvero con la memoria flash cancellata, bisogna programmare il bootloader nel microcontrollore facendo uso di un programmatore, come per esempio il PICKIT 3. La programmazione può essere fatta sia dall'ambiente di sviluppo MPLAB X direttamente dopo la compilazione del bootloader sia caricando direttamente il file .hex generato dalla compilazione.
Il progetto del bootloader è fornito dalla Microchip e si trova tra i progetti USB sotto la cartella:

```
[...]\mla\v2016_11_07\apps\usb\device\bootloaders\firmware\pic18_non_j
```

Si fa notare che la versione gratuita del compilatore XC8, avendo un numero limitato di ottimizzazioni, non riesce a compilare il bootloader nei limiti di memoria di 0x1000 byte. Infatti a causa dell'ottimizzazione ridotta questo eccede i limiti e creerebbe errori se si fanno uso degli spostamenti di memoria appena citati. Per evitare questo bisogna necessariamente abilitare la versione PRO (gratuita per un tempo limitato di prova) o far uso direttamente dei file precompilati .hex (che include PID e VID Microchip).

Una volta installato il bootloader, è possibile caricare il programma principale facendo uso del bootloader stesso. Non è possibile programmare l'applicazione direttamente dall'ambiente di sviluppo dopo la compilazione del programma. Facendo infatti in questo modo verrebbe cancellata la memoria Flash, cancellando il bootloader precedentemente programmato.

Una volta caricato il bootloader e il programma principale, potrebbe essere letta la memoria flash esportandola in un unico file HEX. Questo potrebbe essere utile qualora si vogliano programmare più microcontrollori con il bootloader e l'applicazione, programmando nel microcontrollore un file unico. La Microchip fornisce anche il programma che permette di unire due file HEX in un solo file HEX, evitando di dover usare la procedura sopra citata.

Per poter programmare un file HEX dal PC all'interno della scheda in cui è stato preventivamente caricato il bootloader, è necessario far uso di un'applicazione dedicata. Normalmente chi fornisce il bootloader fornisce anche questa applicazione dedicata al

bootloader. La cosa importante da tenere a mente è che il bootloader hanno un VID e PID dedicato che differisce dal VID e PID del programma. In questo modo quando il bootloader viene avviato, si identifica al PC con il proprio VID e PID permettendo all'applicazione di riconoscere la volontà dell'utente di voler aggiornare il proprio Firmware. In particolare il bootloader della Microchip, basato su classe HID, è identificato dal seguente VID, PID:

- VID = 0x04D8
- PID = 0x0000

I bootlaoder forniti per le schede LaurTec sono identificati dal seguente VID e PID.

- VID = 0xF750
- PID = 0x04D8

dal momento che i codici forniti da Microchip e LaurTec sono differenti, anche il software da utilizzare per caricare il Firmware dal PC deve essere opportunamente cambiato per i codici diversi, altrimenti il bootloader non verrebbe riconosciuto. In particolare il progetto LaurTec PIC bootloader, scaricabile gratuitamente dal sito LaurTec, fornisce sia i bootloader modificati con i nuovi VID e PID LaurTec che l'applicazione per il PC modificato. Per maggiori dettagli sull'utilizzo del software di aggiornamento si rimanda alla guida del progetto stesso.

Impostare il PID e VID

Per poter utilizzare i programmi sviluppati con lo Stack USB bisogna sempre impostare il VID (Vendor ID) e PID (Product ID) contenuti nel file descriptor. In particolare per test in laboratorio si può far uso del VID e PID forniti dalla Microchip o quelli LaurTec, mentre per applicazioni commerciali bisogna richiedere dei propri codici VID e PID al sito USB.org o far uso di codici forniti per sublicensing. I codici Microchip sono già scritti nelle applicazioni di esempio per cui non richiedono alcuna modifica. Per fare esperienza e comprendere tutti i passi necessari, si possono anche cambiare i codici facendo uso di quelli LaurTec. I VID e PID utilizzati, al fine di permettere l'utilizzo dei driver corretti e discriminare il bootloader sono differenti:

PIC18 Bootloader
- PID = 0xFC5D
- VID = 0x04D8

Applicazioni Classe CDC
- PID = 0xF5B9
- VID = 0x04D8

Applicazioni Classe HID
- PID = 0xF750
- VID = 0x04D8

Il VID e il PID possono essere trovati all'intero del file usb_descriptors.c (cartella del

progetto Source Files → usb), come sotto riportato (valori evidenziati in rosso).

```
const USB_DEVICE_DESCRIPTOR device_dsc=
{
    0x12,                   // Size of this descriptor in bytes
    USB_DESCRIPTOR_DEVICE,  // DEVICE descriptor type
    0x0200,                 // USB Spec Release Number in BCD format
    0x00,                   // Class Code
    0x00,                   // Subclass code
    0x00,                   // Protocol code
    USB_EP0_BUFF_SIZE,      // Max packet size for EP0, see usb_config.h
    MY_VID,                 // Vendor ID
    MY_PID,                 // Product ID: Mouse in a circle fw demo
    0x0003,                 // Device release number in BCD format
    0x01,                   // Manufacturer string index
    0x02,                   // Product string index
    0x00,                   // Device serial number string index
    0x01                    // Number of possible configurations
};
```

In particolare i valori sono definiti all'interno del file usb_config.h presente nella cartella usb del progetto

```
#define MY_VID 0x04D8
#define MY_PID 0x0000
```

Nel file usb_descriptors.c sono presenti anche gli altri descriptor per mezzo dei quali è possibile impostare alcune delle proprietà del dispositivo USB. Tra i principali si ricorda la corrente massima che il dispositivo può assorbire e la descrizione del dispositivo stesso.

Driver USB

Una volta realizzato il dispositivo USB e collegato al PC bisogna fare in modo che l'Host lo possa riconoscere. Sebbene il protocollo USB è stato concepito per permettere di nascondere molte delle fasi di inizializzazione del dispositivo stesso, bisogna comunque accertarsi che il driver per il nostro dispositivo sia propriamente installato. Per gli esempi che verranno mostrati a breve si prendono in considerazione i due casi in cui si faccia uso della classe HID e la classe CDC.

Driver per la Classe HID

I sistemi embedded che fanno uso della Classe HID hanno il vantaggio di non richiedere l'istallazione di alcun driver, infatti i sistemi operativi moderni possiedono i driver per i dispositivi HID. In particolare l'Host, una volta identificato il dispositivo USB come periferica HID, associa al nuovo VID e PID il driver idoneo. Le applicazioni dal lato del PC per poter comunicare con il sistema embedded devono solo conoscere il VID e PID del dispositivo al fine di poterlo riconoscere tra i vari dispositivi USB installati.

Driver per la Classe CDC

Per un sistema embedded che faccia uso della Classe CDC, sebbene non sia richiesto alcun driver come per la classe HID, è necessario fornire un file di configurazione .ini per

specificare il driver da utilizzare, anche se già presente nel sistema operativo. Il file .ini è fornito tra i file di progetto della libreria Microchip e nel progetto *LaurTec PIC bootloader*. Tale file deve essere aggiornato a seconda del VID e PID utilizzato per il proprio progetto con classe CDC. Il formato con cui si trovano il VID e PID nel file .ini è USB\VID_xxxx&PID_yyyy. Un esempio è:

```
%DESCRIPTION%=DriverInstall, USB\VID_04D9&PID_000A
```

che è appunto il VID e PID utilizzato dalla Microchip all'interno del file .inf.

In questi file, oltre al VID e PID è possibile anche cambiare gli identificatori che descrivono il tipo di dispositivo e il produttore.

Il Driver per la classe CDC viene richiesto solo al primo collegamento del dispositivo. È importante notare che il VID e PID da utilizzare devono essere diversi da quelli utilizzati nelle prove o dispositivi con classe HID altrimenti il sistema operativo all'associazione del driver potrebbe installare/collegare il driver di Classe errata e il dispositivo non funzionerebbe. Si potrebbe anche verificare che il sistema operativo non trovi affatto il driver idoneo a causa dell'ambiguità VID e PID.

Esempi di programmazione con la Classe CDC

Per le applicazioni in cui si vuole avere un'interfaccia grafica sul PC facile da programmare, probabilmente la Classe CDC è la migliore da usare visto che emula una porta seriale RS232. Infatti per la gestione della porta seriale RS232 sono presenti molte librerie e Visual Studio .NET di Microsoft, la include di default. Un altro vantaggio di questa Classe, è che simulando la porta seriale RS232 permette anche di utilizzare gli strumenti di Debug di questo protocollo, ovvero i vari software di gestione della porta seriale. In particolare si ricorda *RS232 Terminal* scaricabile gratuitamente dal sito LaurTec.

Dal lato dello stack USB, la libreria Microchip fornisce diverse funzioni e macro che è possibile richiamare al fine di poter gestire la porta USB come un semplice modulo UART integrato nel microcontrollore. Per vedere i dettagli di queste funzioni basta aprire un qualunque esempio basato su classe CDC. Gli esempi presentati di seguito, sebbene facciano riferimento al PIC mostrato in testa all'esempio stesso, sono piuttosto generici e possono essere riutilizzati anche per altri modelli supportati dallo stack USB.

Funzioni disponibili per la Classe CDC

La libreria USB Microchip, ovvero il Framework, mette a disposizione diverse funzioni per mezzo delle quali è possibile leggere e scrivere attraverso la porta USB come se fosse una porta seriale. Queste possono essere trovate nel file usb_device_cdc.h, mentre la loro implementazione è nel file usb_device_cdc.c. Per una descrizione dettagliata è bene far riferimento al file .h, nel quale, per ogni funzione, sono descritti esempi e modalità d'uso.
Alcune delle funzioni che verranno a breve descritte, sono in realtà delle macro, per cui sono dichiarate solo nel file usb_device_cdc.h ovvero non hanno una implementazione del file .c. Le funzioni e macro che la libreria mette a disposizione sono:

`void CDCTxService(void)`

Sebbene questa funzione non richieda parametri in ingresso e non restituisca alcun valore, deve essere richiamata una volta ad ogni ciclo del loop presente nel main al fine di aggiornare la state machine interna, e permettere la trasmissione dei dati eventualmente presenti nel buffer.

`USBUSARTIsTxTrfReady()`

Questa funzione è in realtà una macro, e permette di controllare se il modulo USB è pronto per poter inviare dei dati. Questa macro deve essere richiamata prima di effettuare una qualunque trasmissione dati. La macro restituisce 1 se il modulo USB può trasmettere dati, 0 altrimenti. Questa funzione non deve essere richiamata/controllata per mezzo dell'istruzione `while`, visto che si creerebbe una istruzione bloccante che impedirebbe al resto della libreria di funzionare correttamente, dunque di aggiornare lo stato delle variabili.

Tipo: Macro
Parametri: void
Restituisce:

1: Il modulo USB può trasmettere dati.
0: Il modulo USB non può momentaneamente trasmettere dati.

void putUSBUSART(uint8_t *data, uint8_t Length)

Questa funzione permette di scrivere dati dalla RAM al modulo USB. Il puntatore `data` rappresenta il puntatore alla variabile o array in cui è contenuta l'informazione da trasmettere. Tale funzione deve essere utilizzata quando è nota la lunghezza dell'array, infatti il secondo parametro da passare rappresenta proprio la lunghezza dell'array. In particolare permette anche di inviare dati di valore 0x00. Qualora si passasse una variabile semplice di tipo `char`, la lunghezza è 1. Si noti che il tipo della variabile `length` e `data` sono di tipo `uint8_t`, definito come:

```
#ifndef uint8_t
typedef unsigned char uint8_t;
#define uint8_t uint8_t
```

Tipo: Funzione
Parametri:
`*data`: Puntatore alla struttura dati residente in memoria RAM.
`length`: Lunghezza della struttura dati (numero di byte ≤ 255) .
Restituisce: void

uint8_t getsUSBUSART(uint8_t *buffer, uint8_t len)

Questa funzione, viene utilizzata per leggere i dati dal modulo USB. La funzione accetta come parametri il puntatore alla struttura dati della variabile o array in cui scrivere i dati, e la lunghezza, ovvero il numero di byte che ci si aspetta siano presenti. Nel caso in cui siano presenti più byte verranno letti solo quelli richiesti (i rimanenti possono essere letti in letture successive), mentre qualora non siano presenti i byte richiesti, vengono restituiti solo quelli presenti. La funzione non è bloccante, ovvero non aspetta per l'arrivo di tutti i byte. Restituisce 0 se non sono presenti byte da leggere.

Tipo: Funzione
Parametri:
`*buffer`: Puntatore alla struttura dati, residente in memoria RAM, dove scrivere i dati letti.
`len`: Numero di byte da leggere (numero di byte ≤ 255) .
Restituisce: Numero di byte letti. Vale 0 se non sono presenti dati da leggere.

void putrsUSBUSART(const char *data)

Questa funzione permette di scrivere una stringa terminata dal carattere '\0' memorizzata in memoria Flash (stringa costante). Prima di chiamare tale funzione è necessario accertarsi che il modulo USB possa accettare richieste di trasmissione dati. La funzione richiede solo il puntatore alla struttura che contiene i dati da trasmettere, ovvero l'indirizzo della memoria flash dove è contenuta la stringa d'interesse. La lunghezza non è richiesta poiché la stringa è terminata dal carattere '\0'.

Tipo: Funzione
Parametri:
`*data`: Puntatore alla struttura dati, residente in memoria Flash.
Restituisce: void

```
void putsUSBUSART(char *data);
```

Questa funzione si comporta come la funzione precedente tranne che per il fatto che il puntatore è di tipo "normale", ovvero ad una struttura dati presenti in RAM. Anche in questo caso la stringa deve essere terminata con il carattere '\0'.

Tipo: Funzione
Parametri:
`*data`: Puntatore alla struttura dati, residente in memoria RAM.
Restituisce: void

Esempio 1: Emulazione di una porta seriale RS232

Microcontrollore: PIC18F14K50
Scheda di sviluppo: miniCOM USB
Bootloader: Non presente

Come prima applicazione vediamo un esempio di emulazione della porta seriale RS232, ovvero creiamo un dispositivo che ci permette di realizzare la funzione di un cavo adattatore USB-UART. Per cui scrivendo sul PC dei dati su *RS232 Terminal* o altro programma di Debug, i dati vengono inviati tramite porta USB al microcontrollore e restituiti direttamente in uscita alla porta UART del microcontrollore, allo stesso bitrate impostato sul PC. Collegando assieme la linea TX e RX del modulo UART si effettua un eco del segnale ovvero sul terminal di Debug viene mostrato nuovamente il carattere (byte) inviato.

Per realizzare questa applicazione, non bisogna in realtà scrivere alcuna riga di codice visto che tra gli esempi Microchip è presente il progetto:

```
cdc_serial_emulator
```

che svolge appunto quanto appena detto.
Per verificare l'applicazione non bisogna far altro che caricare il progetto all'interno dell'ambiente di sviluppo MPLAB X in particolare l'esempio:

```
low_pin_count_usb_development_kit_pic18f14k50.x
```

Successivamente è bene compilare per mezzo del comando *Clean and Build Main Project* al fine di effettuare una compilazione con nuovi file intermedi. Successivamente potete collegare il programmatore alla scheda miniCOM USB, che deve essere preventivamente collegata già alla porta USB al fine di essere alimentata. Effettuati i collegamenti potete scaricare il programma all'interno del microcontrollore.

Se tutto è andato a buon fine il sistema operativo riconoscerà la scheda di sviluppo come adattatore USB-UART ovvero creerà una COM port che sarà possibile aprire con un Terminal.

Al primo collegamento della scheda, a seconda del sistema operativo, potrebbe o meno essere richiesto il driver. Per essere certi che il dispositivo sia propriamente installato bisogna andare tra le periferiche di sistema e controllare che non sia presente un punto

esclamativo sulla periferica associata alla scheda di sviluppo, come mostrato in Figura 15.

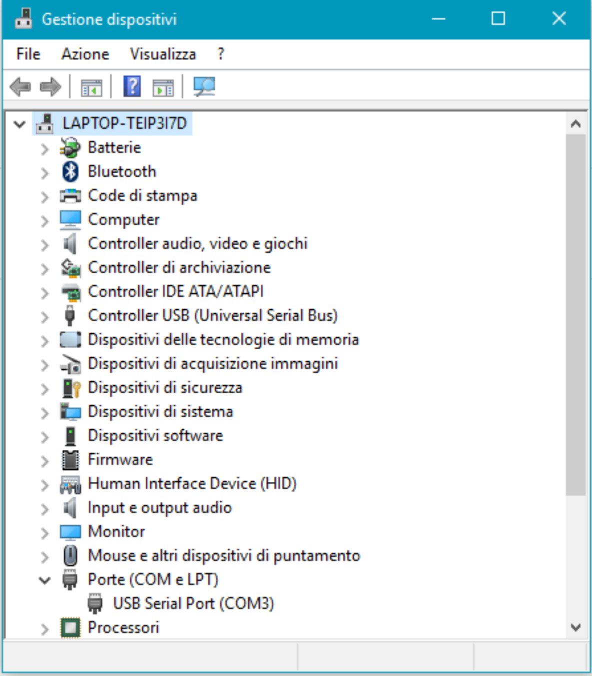


Figura 15: *Porta COM3 creata al collegamento della scheda.*

Nel caso in cui sulla periferica visualizzata nel gruppo Porte (COM e LPT), sia presente un punto esclamativo, bisogna installare manualmente il driver ovvero il file .inf con le informazioni relative al VID, PID e driver da utilizzare. Per fare questo basta cliccare sulla periferica con il punto esclamativo e selezionare la voce "Aggiornamento software driver..."

Se non avete cambiato il VID e PID troverete il file inf, nella directory:

```
[…]\mla\v2016_11_07\apps\usb\device\cdc_serial_emulator\utilities\inf
```

se avete fatto uso del VID e PID di LaurTec il file .inf opportunamente modificato lo trovate nel progetto LaurTec_PIC_Bootloader, scaricabile dal sito LaurTec, nella cartella:

```
LaurTec_PIC_Bootloader_2.0.0\04 - Drivers\CDC - RS232 Emulation - inf
```

Una volta verificato che il sistema riconosce propriamente il dispositivo è possibile aprire il software *RS232 Terminal* ed inviare dei caratteri al dispositivo. In particolare ogni volta che si invia un carattere non viene visualizzato nulla come risposta a meno di non collegare assieme la linea TX (pin 3) e RX (pin 1) del connettore LTB1 presente nella scheda di sviluppo miniCOM USB. In particolare abilitando la funzione echo su *RS232 Terminal*, i caratteri sono mostrati due volte, ovvero un echo del Terminal e un echo da parte della scheda, come mostrato in Figura 16.

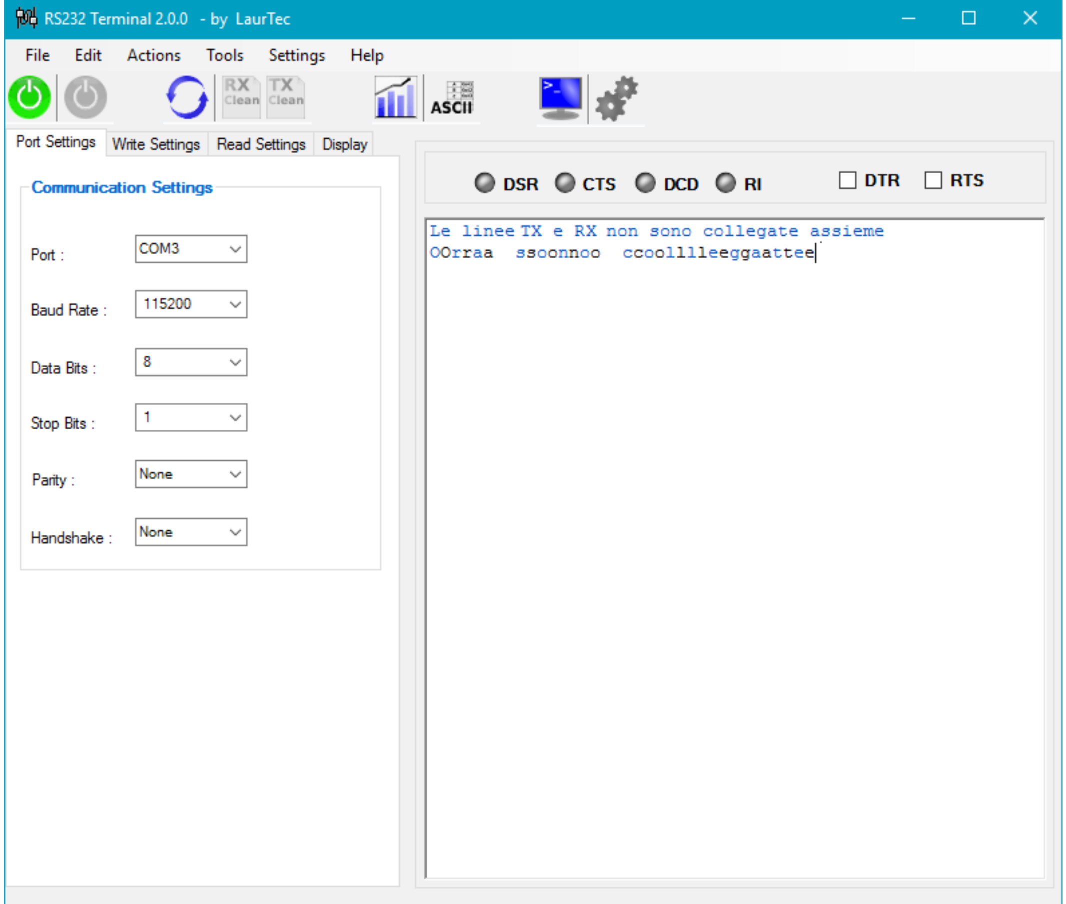


Figura 16: *Esempio di echo ottenuto via software e collegando la linea TX e RX assieme.*

Il Baud Rate (bit rate) della porta può essere variato a piacimento sulla finestra di *RS232 Terminal*. I bit in uscita dal modulo UART della scheda miniCOM USB rifletteranno automaticamente il Baud Rate selezionato.

L'esempio appena mostrato può essere caricato anche sulla scheda Freedom III visto che fa uso del PIC18F14K50, ma al fine di poterlo caricare in maniera opportuna bisogna compilare il codice per il bootloader e caricare il codice stesso facendo uso dell'applicazione *LaurTec PIC Bootloader*. Oltre a cambiare le impostazioni del Linker, è necessario anche cambiare il VID e PID con quelli LaurTec.

Esempio 2: Controllo dei pulsanti Terminal

Microcontrollore: PIC18F4550
Scheda di sviluppo: Freedom II, Freedom Light
Bootloader: Non presente

In questo secondo esempio scriveremo un programma ad hoc per poter leggere dei pulsanti e visualizzare il pulsante premuto sul Terminal.

Prima di procedere è bene fare una copia del progetto che andremo a modificare, ovvero:

```
cdc_basic
```

Si può rinominare il progetto in

```
02_Esempio_CDC_basic
```

Dal progetto si deve poi aprire l'esempio:

```
picdem_fs_usb.x
```

Analizzando il programma si può vedere che la funzione main è piuttosto semplice e non fa altro che inizializzare il modulo USB ed entrare in un loop infinito in cui viene eseguita la funzione:

```
APP_DeviceCDCBasicDemoTasks();
```

La nostra applicazione deve sostituire tale chiamata, ovvero si deve togliere la chiamata alla funzione di esempio e mettere quella alla propria funzione.

Nel caso di un semplice esempio come quello sotto, si può anche scrivere il tutto all'interno della funzione main. Nella funzione main si può anche aggiungere l'inizializzazione dei pin utilizzati per i pulsanti. Apportando le dovute modifiche la funzione main del progetto può essere cambiata con la seguente:

```
void main(void){

  // Variabile da trasmettere contenente il testo
  unsigned char dataOUT[11];

  //0: sistema non pronto
  //1: sistema pronto
  unsigned char system_is_ready = 0;

  //0: non premuto
  //1: premuto
  unsigned char BT1_pressed = 0;
  unsigned char BT2_pressed = 0;

  //Inizializzazione del modulo USB
  SYSTEM_Initialize(SYSTEM_STATE_USB_START);
  USBDeviceInit();
  USBDeviceAttach();
```

```
// Imposto PORTA tutti ingressi
LATA = 0x00;
TRISA = 0xFF;

// Imposto PORTB tutti ingressi
LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi
LATC = 0x00;
TRISC = 0xFF;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

// Abilita i resistori di pull-up sulla PORTB
INTCON2bits.RBPU = 0x00;

// Inizializzazione Testo
dataOUT[0] = 'B';
dataOUT[1] = 'U';
dataOUT[2] = 'T';
dataOUT[3] = 'T';
dataOUT[4] = 'O';
dataOUT[5] = 'N';
dataOUT[6] = ':';
dataOUT[7] = ' ';
dataOUT[9] = 13;
dataOUT[10] = '\0';

while(1){

    SYSTEM_Tasks();

    //Controllo se il modulo USB è propriamente inizializzato
    //Se non è inizializzato non controlla i pulsanti ed effettua un altro
    //controllo dall'inizio
    if( USBGetDeviceState() < CONFIGURED_STATE ){
        system_is_ready = false;
    } else {
        system_is_ready = true;
    }

    //Controllo se il modulo è in stato suspend
    if( USBIsDeviceSuspended()== true ) {
        system_is_ready = false;
    } else {
        system_is_ready = true;
    }

    //Controllo i pulsanti solo se il sistema è pronto
    if (system_is_ready == true){

        // Controllo pressione BT1
        if (PORTBbits.RB4 == 0 )
            BT1_pressed = true;
```

```
      if(USBUSARTIsTxTrfReady() && PORTBbits.RB4 == 1 && BT1_pressed == true) {
          dataOUT[8] = '1';
          putsUSBUSART(dataOUT);
          BT1_pressed = false;
       }


      // Controllo pressione BT2
      if (PORTBbits.RB5 == 0 )
         BT2_pressed = true;

      if(USBUSARTIsTxTrfReady() && PORTBbits.RB5 == 1 && BT2_pressed == true) {
          dataOUT[8] = '2';
          putsUSBUSART(dataOUT);
          BT2_pressed = false;
       }

      }

      CDCTxService();

      }
    }
```

Il programma inizializza il modulo USB per mezzo delle seguenti funzioni:

```
//Inizializzazione del modulo USB
  SYSTEM_Initialize(SYSTEM_STATE_USB_START);
  USBDeviceInit();
  USBDeviceAttach();
```

per poi inizializzare le varie porte del microcontrollore.
Si fa notare che sebbene non sia utilizzata la PORTD, il LED0 lampeggia visto che è controllato dallo stack, in particolare dalla funzione:

```
void APP_LEDUpdateUSBStatus(void);
```

mantenendo la funzione, ma cancellando il suo contenuto, viene liberato il LED anche per applicazioni generiche.

L'applicazione inizia poi in maniera ciclica a controllare se il modulo USB è pronto, ed in particolare viene aggiornata la seguente variabile:

```
system_is_ready
```

Quando la variabile vale true (ovvero 1) vengono controllati i tasti. In particolare al fine di evitare che la stringa di pressione venga inviata in maniera continua, viene impostato un flag per rilevare la pressione del tasto e successivamente, se il modulo USB può trasmettere dati e il tasto è stato rilasciato, viene inviata la stringa che contiene il testo BUTTON: e il relativo tasto premuto, 1 o 2.
La stringa consiste in un array di 11 caratteri, in particolare il carattere 8 viene aggiornato in base alla pressione del tasto al valore '1' o '2'. Il carattere 9 viene impostato al valore numerico 13, che corrisponde al carattere ASCII del tasto di ritorno a capo. Il carattere 10 dell'arrary è impostato con il carattere di fine stringa '\0'.

Si noti che alla fine del programma viene richiamata la funzione:

```
CDCTxService();
```

che permette di aggiornare la *state machine* dello stack USB.

Collegando il sistema al PC, dopo aver programmato la scheda, alla pressione dei tasti BT1 e BT2 viene visualizzato il relativo testo, come riportato in Figura 17. Il bit rate della porta seriale può essere impostato ad un qualunque valore, visto che lo stack si adatta in automatico a qualunque valore supportato.

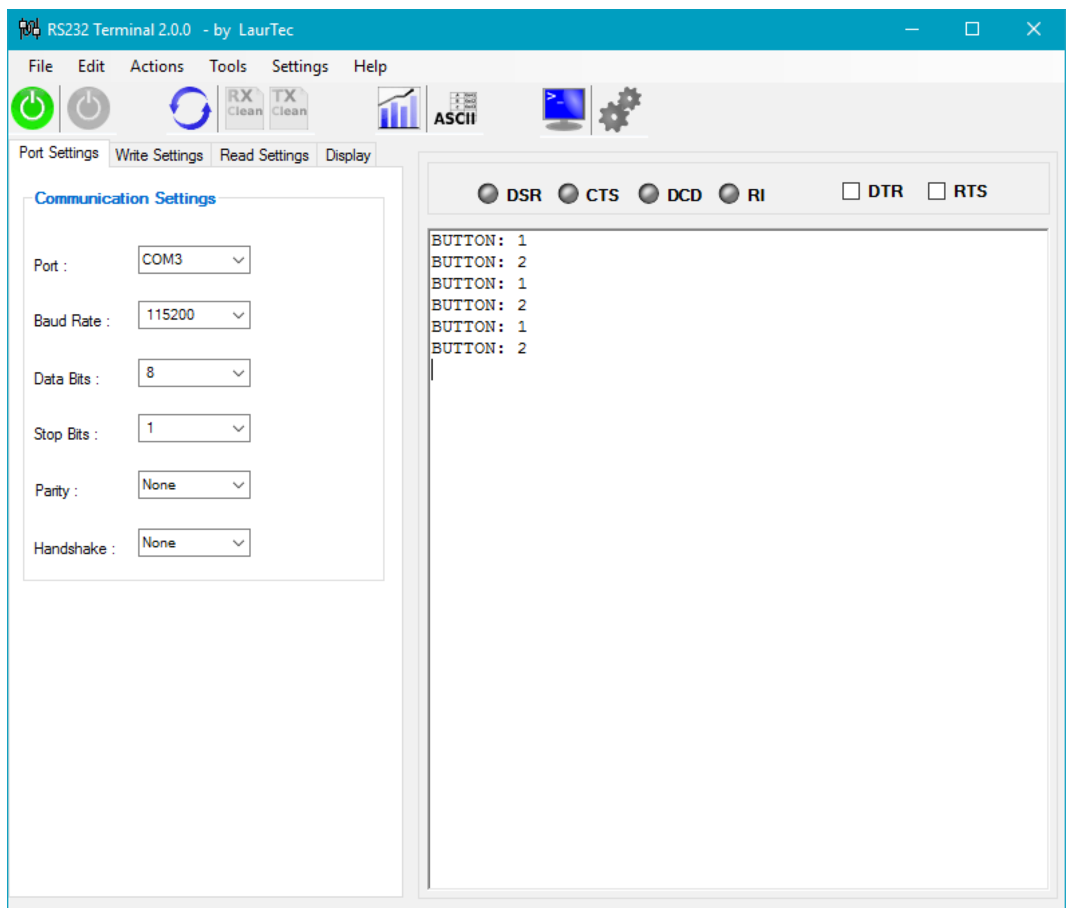


Figura 17: *Esempio di dati visualizzati su RS232 Terminal alla pressione dei pulsanti.*

In maniera analoga all'esempio mostrato, potrebbero essere inviate anche altre informazioni, come per esempio un array contenente le misure di un ADC. L'importante è che ogni controllo ed eventuale misura non sia di tipo bloccante e permetta il corretto funzionamento dello stack.

L'esempio mostrato fa uso dello stack USB in modalità Interrupt, per cui la parte relativa al polling è stata eliminata, ovvero la chiamata alla funzione `USBDeviceTasks();`.

Esempio 3: Controllo di 8 LED tramite Terminal

Microcontrollore: PIC18F4550
Scheda di sviluppo: Freedom II, Freedom Light
Bootloader: Non presente

In questo ultimo esempio vediamo come controllare per mezzo di un Terminal, 8 LED, o anche 8 Relay, semplicemente inviando dei caratteri alla porta seriale. In particolare il protocollo utilizzato è il seguente:

- Il primo byte è un header fisso pari a 0xAA
- Il secondo byte determina quale LED accendere; ad ogni LED è associato un bit.

Per esempio:
- Per accendere il LED 0 bisogna inviare 0xAA 0x01.
- Per accendere il LED 1 bisogna inviare 0xAA 0x02.
- Per accendere il LED 4 e 1 bisogna inviare 0xAA 0x12.

Questo esempio mostra ancora una volta come sia semplice gestire la porta USB dal lato PC qualora il dispositivo sia riconosciuto tramite porta seriale virtuale. Come nel precedente esempio bisogna creare una copia del progetto originale:

```
cdc_basic
```

Si può rinominare il progetto in

```
03_Esempio_CDC_basic
```

Dal progetto si deve poi aprire l'esempio:

```
picdem_fs_usb.x
```

Per evitare che il LED presente sulla PORTD bit 0 sia controllato dalla funzione APP_LEDUpdateUSBStatus bisogna commentare tutto il suo contenuto, lasciando però attiva la funzione stessa.

Il file main per la gestione del nostro nuovo programma può essere come sotto riportato:

```
void main(void){

    //0: sistema non pronto
    //1: sistema pronto
    unsigned char system_is_ready = 0;

    unsigned char header_received = 0;

    unsigned char data_received = 0;
    unsigned char number_of_bytes = 0;
```

```
//Inizializzazione del modulo USB
SYSTEM_Initialize(SYSTEM_STATE_USB_START);
USBDeviceInit();
USBDeviceAttach();

// Imposto PORTA tutti ingressi
LATA = 0x00;
TRISA = 0xFF;

// Imposto PORTB tutti ingressi
LATB = 0x00;
TRISB = 0xFF;

// Imposto PORTC tutti ingressi
LATC = 0x00;
TRISC = 0xFF;

// Imposto PORTD tutte uscite
LATD = 0x00;
TRISD = 0x00;

// Imposto PORTE tutti ingressi
LATE = 0x00;
TRISE = 0xFF;

while(1){

    SYSTEM_Tasks();

    //Controllo se il modulo USB è propriamente inizializzato
    //Se non è inizializzato non controlla i pulsanti ed effettua un altro
    //controllo dall'inizio
    if( USBGetDeviceState() < CONFIGURED_STATE ){
        system_is_ready = false;
    } else {
        system_is_ready = true;
    }

    //Controllo se il modulo è in stato suspend
    if( USBIsDeviceSuspended()== true ) {
        system_is_ready = false;
    } else {
        system_is_ready = true;
    }

    //Controllo i pulsanti solo se il sistema è pronto
    if (system_is_ready == true){

        // Leggo un byte dal buffer in ingresso
        number_of_bytes = getsUSBUSART (&data_received, 1);

        if (number_of_bytes != 0){
            if (header_received == true){
                LATD = data_received;
                header_received = false;
            }

            if (data_received == 0xAA){
                header_received = true;
            }
        }
```

```
        }

        CDCTxService();
    }
}
```

Per quanto riguarda l'inizializzazione, il programma è simile all'esempio precedente. Successivamente, verificato che il modulo USB è propriamente inizializzato e non è nella modalità Suspend, il programma inizia a leggere i dati in arrivo dal PC, inviati per mezzo di un Terminal. In particolare i dati nel buffer sono letti un byte alla volta, ma passando un array si potrebbero anche leggere un numero maggiore di byte, qualora presenti nel buffer. In particolare l'istruzione sotto, oltre che a leggere un eventuale byte, assegna alla variabile `number_of_bytes` il numero di byte letti, che nel caso specifico o vale 0 o 1.

```
// Leggo un byte dal buffer in ingresso
number_of_bytes = getsUSBUSART (&data_received, 1);
```

Questo controllo permette di accertarsi che i controlli successivi siano effettuati solo se durante il ciclo di lettura è stato effettivamente letto un byte dal buffer.
Le istruzioni di controllo che seguono permettono di verificare il primo e secondo byte secondo il semplice protocollo precedentemente creato.

Il protocollo sebbene abbia un header byte che permette di limitare i problemi da eventuali disturbi, non copre tutte le casistiche derivanti dalla gestione di LED o Relay in maniera affidabile. Un'analisi dettagliata del problema e casistiche potrebbe essere un buon esercizio per estendere l'applicazione.

Per testare il programma si è fatto uso dell'applicazione *RS232 Terminal* impostando il formato dei dati in HEX. In particolare facendo uso dello strumento Macro, è possibile assegnare ad ogni tasto il codice per l'accensione del LED d'interesse, come mostrato in Figura 18.

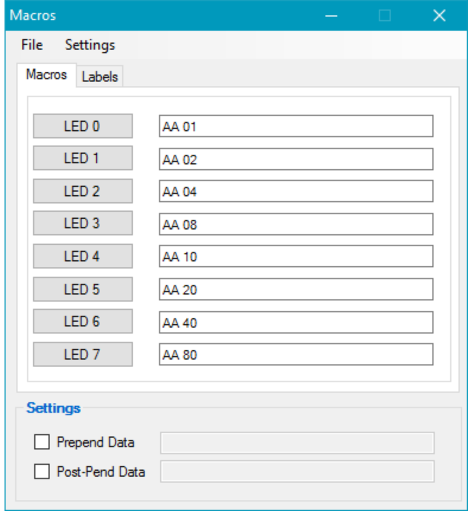
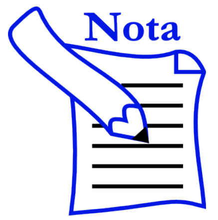


Figura 18: *Esempio di utilizzo dello strumento Macro di RS232 Terminal.*

Esempi di programmazione con la Classe HID

La classe HID, diversamente dalla classe CDC, permette di realizzare dispositivi che non richiedono nessun file .ini da installare. Sebbene per entrambe le classi il driver risulta in generale già installato, da un punto di vista dell'utilizzatore, un dispositivo HID risulta certamente di più facile installazione. Tra le periferiche tipiche che rientrano nella classe HID vi sono il mouse e la tastiera, ma la stessa classe può essere utilizzata anche per trasmettere informazioni generiche che rappresentano dati di altra natura. Negli esempi che seguono verranno mostrati un esempio che modifica il progetto di una tastiera e un progetto generico di classe HID. La scelta della classe HID complica leggermente gli aspetti della progettazione del software dal lato del PC, infatti il dispositivo deve essere gestito questa volta come dispositivo USB e non più come semplice porta seriale COM, per la quale per esempio l'ambiente di sviluppo .NET possiede il supporto con oggetti dedicati e di facile utilizzo. Con la classe HID bisogna includere delle librerie .dll che permettano di gestire lo stack ad alto livello. Su internet sono disponibili diversi progetti di librerie USB *open source*. Da questo punto di vista, sebbene lo sviluppo dell'applicazione possa essere più complicato, in realtà si possono usare i progetti base sviluppati dalla Microchip anche dal lato PC, adattandoli alle proprie esigenze.



Gli esempi fanno uso di "numeri magici" ovvero di numeri scritti direttamente nel codice sorgente. Questa pratica non è una buona regola di programmazione, ma al fine di avere un codice compatto la si è preferita alla definizione di costanti da scrivere in punti diversi. Questo avrebbe richiesto il mettere in evidenza diversi punti del codice in cui dover apportare le modifiche.

Esempio 1: Realizzare una tastiera a due tasti

Microcontrollore: PIC18F4550
Scheda di sviluppo: Freedom II, Freedom Light
Bootloader: Non presente

In questo esempio viene modificato il progetto base della tastiera:

```
hid_keyboard
```

In particolare si vuole realizzare una semplice tastiera che ci permetta per esempio di scorrere un documento avanti ed indietro, ovvero implementi i tasti di navigazione "Page Up", "Page Down".

Come nel precedente esempio bisogna creare una copia del progetto originale, in particolare si può rinominare il progetto in:

```
01_Esempio_tastiera
```

Dal progetto si deve poi aprire l'esempio:

```
picdem_fs_usb.x
```

se si vuole utilizzare il PIC18F4550 o il progetto:

```
low_pin_count_usb_development_kit_pic18f14k50.x
```

qualora si voglia utilizzare il PIC18F14K50.

Una volta aperto il progetto, si può vedere che il main è piuttosto semplice e viene richiamata poi la funzione:

```
APP_KeyboardTasks();
```

all'interno della quale avviene la gestione dei tasti. Al fine di soddisfare le specifiche USB per una periferica HID, sono effettuati diversi controlli, che nel nostro caso non sono di molto interesse.

Il codice che bisogna modificare, ovvero dove dobbiamo inserire il controllo dei nostri tasti è il seguente:

```
if(HIDTxHandleBusy(keyboard.lastINTransmission) == false)
    {
        /* Clear the INPUT report buffer.  Set to all zeros. */
        memset(&inputReport, 0, sizeof(inputReport));

        if(BUTTON_IsPressed(BUTTON_USB_DEVICE_HID_KEYBOARD_KEY) == true)
        {
            if(keyboard.waitingForRelease == false)
            {
                keyboard.waitingForRelease = true;

                /* Set the only important data, the key press data. */
                inputReport.keys[0] = keyboard.key++;

                //In this simulated keyboard, if the last key pressed
                exceeds the a-z + 0-9,
                //then wrap back around so we send 'a' again.
                if(keyboard.key == 40)
                {
                    keyboard.key = 4;
                }
            }
        }
        else
        {
            keyboard.waitingForRelease = false;
        }
```

osservando il codice ci si rende conto che la lettera 'a' non è rappresentata dal codice ASCII 97 (0x61) con cui si è soliti lavorare. Infatti l'USB Implementers Forum per le tastiere, utilizza un altro codice i cui dettagli sono mostrati nel documento ufficiale:

```
HID usage Tables
```

In particolare l'alfabeto da 'a' alla 'z' sono contenuti tra i codici 4-29 (decimale). I numeri tra 1,2,3,...9, 0 sono rappresentati dall'intervallo 30-39.

Nel caso dell'esempio specifico, i tasti/codici d'interesse sono:

- Page Down: 78
- Page Up: 75

Per controllare i due tasti, il codice deve essere modificato nel seguente modo:

```
if(HIDTxHandleBusy(keyboard.lastINTransmission) == false){

        /* Clear the INPUT report buffer.  Set to all zeros. */
        memset(&inputReport, 0, sizeof(inputReport));

        //Page Up is pressed
        if(PORTBbits.RB4 == 0x00){

            if(keyboard.waitingForRelease == false){

                keyboard.waitingForRelease = true;
                inputReport.keys[0] = 75;

            }

          //Page Down is pressed
        } if (PORTBbits.RB5 == 0x00){

            if(keyboard.waitingForRelease == false){

                keyboard.waitingForRelease = true;

                inputReport.keys[0] = 78;
            }
        }
        else{

            keyboard.waitingForRelease = false;
        }
```

Si noti che il codice relativo nel tasto viene inserito all'interno dell'array:

```
inputReport.keys[]
```

tale array è composto da 6 byte [0..5] e contiene la lista dei tasti premuti che bisogna inviare. Se per esempio se si premesse il tasto 'a' si dovrebbe scrivere:

```
inputReport.keys[0] = 4;
```

se successivamente, tenendo premuto 'a' si dovesse anche premere 'b' l'array dovrebbe essere impostato nel seguente modo:

```
inputReport.keys[0] = 4;
inputReport.keys[1] = 5;
```

se si dovesse lasciare il tasto 'a', il nuovo array dovrebbe avere il valore solo per 'b'

```
inputReport.keys[0] = 5;
```

Da quanto appena detto si capisce che tale array permette di inviare anche le combinazioni di tasti complesse che sono assegnate a funzioni speciali.

Per far funzionare correttamente il codice, bisogna anche aggiungere il seguente codice nella funzione main :

```
//Enable the switches on the PORTB
LATB = 0x00;
TRISB = 0xF0;

// Enable the pull-up Resistors on PORTB
INTCON2bits.RBPU = 0x00;
```

In particolare questo permette di impostare come input RB4 e RB5 (oltre ad RB6 e RB7 se si volesse usare anche questi) ed attivare i resistori di pull-up.

Il codice base fa già uso dei tasti RB4 e RB5 per cui non sono richieste altre modifiche oltre a quanto mostrato. Qualora si vogliano gestire altri tasti potrebbe essere necessario modificare altre funzioni, come per esempio

```
BUTTON_IsPressed ()
BUTTON_Enable ()
```

e definire eventualmente altre costanti, come fatto già per i due tasti BUTTON_S2 e BUTTON_S3.

Esempio 2: Controllo di ingressi e uscite digitali e analogiche

Microcontrollore: PIC18F4550
Scheda di sviluppo: Freedom II, Freedom Light
Bootloader: Non presente

In questo esempio viene preso come riferimento il progetto base generico HID:

```
hid_custom
```

Come nel precedente esempio bisogna creare una copia del progetto originale, che si può rinominare in:

```
02_Esempio_Gestione_IO_e_Analog
```

Dal progetto si deve poi aprire l'esempio:

```
picdem_fs_usb.x
```

se si vuole utilizzare il PIC18F4550. L'applicazione di esempio, senza lacuna modifica permette di:

- Accendere e spegnere il LED posto sul bit 1 della PORTD.
- Leggere il tasto posto sul pin 4 di PORTB.

- Leggere l'ingresso analogico AN0 (Sensore Light di Freedom II).

Il main ancora una volta viene utilizzato solo per l'inizializzazione del modulo USB e richiamare in maniera ciclica la funzione:

```
APP_DeviceCustomHIDTasks();
```

I dettagli della funzione sono riportati sotto. In particolare la funzione non ha modifiche ma è stata impaginata diversamente dalla funzione originale.

```
void APP_DeviceCustomHIDTasks(){

    /* If the USB device isn't configured yet, we can't really do anything
     * else since we don't have a host to talk to.  So jump back to the
     * top of the while loop. */
    if( USBGetDeviceState() < CONFIGURED_STATE ){
        return;
    }

    /* If we are currently suspended, then we need to see if we need to
     * issue a remote wakeup.  In either case, we shouldn't process any
     * keyboard commands since we aren't currently communicating to the host
     * thus just continue back to the start of the while loop. */
    if( USBIsDeviceSuspended()== true ){
        return;
    }

    //Check if we have received an OUT data packet from the host
    if(HIDRxHandleBusy(USBOutHandle) == false){

        //We just received a packet of data from the USB host.
        //Check the first uint8_t of the packet to see what command the host
        //application software wants us to fulfill.

        //Look at the data the host sent, to see what kind of application
        //specific command it sent.
        switch(ReceivedDataBuffer[0]){

            //Toggle LEDs command
            case COMMAND_TOGGLE_LED:
                LED_Toggle(LED_USB_DEVICE_HID_CUSTOM);
                break;

            //Get push button state
            case COMMAND_GET_BUTTON_STATUS:
                //Check to make sure the endpoint/buffer is free before we
                //modify the contents
                if(!HIDTxHandleBusy(USBInHandle)){

                    //Echo back to the host PC the command we are
                    //fulfilling in the first uint8_t.
                    //In this case, the Get Pushbutton State command.
                    ToSendDataBuffer[0] = 0x81;

                    //pushbutton not pressed, pull up resistor on circuit
                    //board is pulling the PORT pin high
                    if(BUTTON_IsPressed(BUTTON_USB_DEVICE_HID_CUSTOM) ==
                                                                 false){
                        ToSendDataBuffer[1] = 0x01;
```

```
                    } else{
                            ToSendDataBuffer[1] = 0x00;
                    }
                    //Prepare the USB module to send the data packet to the host
                    USBInHandle = HIDTxPacket(CUSTOM_DEVICE_HID_EP,
                                   (uint8_t*)&ToSendDataBuffer[0],64);
                }
                break;

            //Read POT command.  Uses ADC to measure an analog voltage on
            //one of the ANxx I/O pins, and returns the result to the host
            case COMMAND_READ_POTENTIOMETER:{

                uint16_t pot;

                //Check to make sure the endpoint/buffer is free before we
                //modify the contents
                if(!HIDTxHandleBusy(USBInHandle)) {

                    pot = ADC_Read10bit(ADC_CHANNEL_POTENTIOMETER);

                    //Echo back to the host the command we are fulfilling
                    //in the first uint8_t.
                    //In this case, the Read POT (analog voltage) command.
                    ToSendDataBuffer[0] = 0x37;
                    ToSendDataBuffer[1] = (uint8_t)pot; //LSB
                    ToSendDataBuffer[2] = pot >> 8;     //MSB


                    //Prepare the USB module to send the data packet to the host
                    USBInHandle = HIDTxPacket(CUSTOM_DEVICE_HID_EP,
                                   (uint8_t*)&ToSendDataBuffer[0],64);
                }
            }

            break;
        }

        //Re-arm the OUT endpoint, so we can receive the next OUT data packet
        //that the host may try to send us.
        USBOutHandle = HIDRxPacket(CUSTOM_DEVICE_HID_EP,
                       (uint8_t*)&ReceivedDataBuffer, 64);
    }
}
```

Il programma controlla in maniera ciclica se il buffer in ingresso abbia ricevuto dei dati, controllando lo stato del buffer richiamando la funzione:

```
HIDRxHandleBusy(USBOutHandle)
```

Quando sono ricevuti dei byte, viene controllato il seguente buffer:

```
ReceivedDataBuffer[]
```

In particolare il primo byte del buffer viene considerato come chiave che descrive i byte successivi. Questa caratteristica non appartiene al protocollo USB ma è semplicemente una scelta del progetto. In particolare con la classe HID sono spediti pacchetti di dati di 64 byte e in questo esempio di dispositivo generico si può scegliere come formattarli.

Il valore che assume il primo byte è tra quelli definiti nella seguente struttura dati definita come:

```
typedef enum
{
    COMMAND_TOGGLE_LED = 0x80,
    COMMAND_GET_BUTTON_STATUS = 0x81,
    COMMAND_READ_POTENTIOMETER = 0x37
} CUSTOM_HID_DEMO_COMMANDS;
```

Si capisce che è possibile cambiare i comandi con altri valori ed eventualmente definire un header di trasmissione anche più complesso.

Il valore del primo byte del buffer è controllato per mezzo del costrutto condizionale `switch case`, in particolare:

```
case COMMAND_TOGGLE_LED
```

In questo caso viene semplicemente invertito lo stato del LED. Modificando la funzione

```
LED_Toggle
```

si potrebbero per esempio controllare 4-8 LED o relay, facendo uso di un secondo byte, piuttosto che effettuare un semplice toggle.

```
case COMMAND_GET_BUTTON_STATUS
```

In questo secondo caso viene controllata la pressione di un tasto e inviato il valore dello stesso al PC per mezzo del buffer:

```
ToSendDataBuffer[]
```

Anche in questo caso il buffer ha un header che precede il valore del secondo byte che contiene l'informazione da trasmettere. Il buffer in uscita viene effettivamente preparato per essere inviato solo dopo la chiamata della seguente funzione:

```
 USBInHandle = HIDTxPacket(CUSTOM_DEVICE_HID_EP,
               (uint8_t*)&ToSendDataBuffer[0],64);
```

In particolare il valore restituito dalla funzione è relativo al buffer del modulo USB utilizzato per la reale trasmissione dell'array e deve essere memorizzato al fine di poter far un controllo sullo stesso se la trasmissione è terminata o meno. Tale controllo è possibile farlo per mezzo del seguente controllo:

```
if(!HIDTxHandleBusy(USBInHandle))
```

L'ultimo caso del case:

```
case COMMAND_READ_POTENTIOMETER
```

Permette di leggere il canale AN0 del modulo ADC, che nel caso della scheda Freedom II

è collegato al sensore di luce.

Il Firmware appena descritto invia le informazione al PC, il quale deve avere un'applicazione ad hoc che possa comunicare con la periferica USB. In particolare nella cartella *utilities* sono presenti diversi progetti che possono adempiere allo scopo e sono stati sviluppati per vari ambienti di sviluppo, tra cui .NET e QT. Non volendo sviluppare alcuna riga di codice si può eseguire il programma contenuto nella seguente cartella:

```
[...]\02_Esempio_Gestione_IO_e_Analog\utilities\plug_and_play_example\wind
ows\bin
```

se la programmazione del microcontrollore va a buon fine il programma riconosce la scheda in automatico e inizia la visualizzazione dei dati del modulo ADC e del pulsante, come riportato in Figura 19.

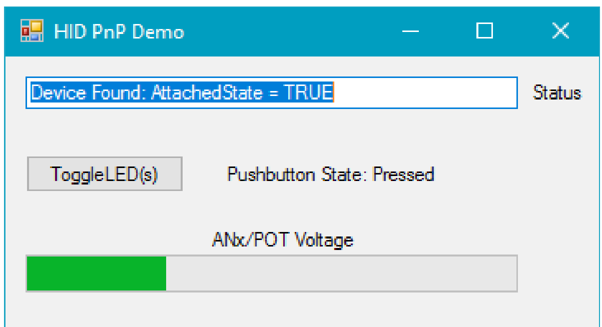


Figura 19: *Esempio dell'applicazione HID in esecuzione.*

A seconda della propria esperienza si può scegliere un ambiente di sviluppo piuttosto che un altro. L'ambiente QT offre la possibilità di scrivere un programma portabile su ogni sistema operativo, mentre il .NET vincola l'utilizzo in ambiente Windows. Gli esempi .NET sono per la release del 2008, ma li ho personalmente ricompilati anche in ambiente .NET 2015.

Indice Alfabetico

A
ADDRESS_STATE....19
APP_DeviceMouseTasks....18
ATTACHED_STATE....19
B
BASIC....4
bootloader....21 e seg.
C
CDC....5
CDCTxService....28, 36
classe CDC....11
Classe CDC....26
Classe HID....26
CONFIGURED_STATE....19
CPUDIV....7, 20
Cypress....5
D
DEFAULT_STATE....19
DETACHED_STATE....19
driver....26
DriverInstall....27
E
EasyUSB....14, 19
Ethernet....16
F
file .hex....23
file .ini....27
file HEX....24
file usb_config.h....20
Firmware....22
FOSC....7, 20
Framework....28
Freedom II....12, 19
Freedom III....11, 15, 19
Freedom Light....13
FSEN....8
FT232H....4
FTDI....4
Full Speed....20
G
getsUSBUSART....29
H
HID....5
HS....7
HSPLL_HS....20
I
IN....4
Interrupt Vector....21 e seg.
L
LaurTec PIC bootloader....27
LDO....8
linker....22
Low Speed....20
LTB....15
M
Mass Storage Device....5
memoria 0x1000....23
Microchip....5
Microchip Libraries for Applications....16
miniCOM Extension Board....15
miniCOM Relay....15
miniCOM RS232....15
MLA....16
modalità Interrupt....20
modalità Polling....20
MSP430F5xx....5
O
OSC1_PLL2....20
OUT....4
P
Phase Lock Loop....6
PHY....5, 10
PIC18F14K50....8, 11, 15, 24
PIC18F4550....12 e seg., 23
PIC18F46K22....11
PICDEM FS USB....17
picdem_fs_usb_k50....17
picdem_fs_usb.x....17
PICDEMTM FS USB....13 e seg.
PICK18F4550....8
PID....25
PLL....6, 8
PLLDIV....6 e seg., 20
PLLEN....8
PORTB....23
POWERED_STATE....19
Product ID....25
pull-up....20, 24
putrsUSBUSART....29
putsUSBUSART....30
putUSBUSART....29
R
RB4....23
Real Time Clock Calendar....8
Reset....23
Reset Vector....21 e seg.
resistori di pull-up....7
ROM Ranges....22
RS232....4, 28

RS232 Terminal....28
RTC....8
S
Serial Interface Engine....8
SIE....8
ST....5
ST32Fxx....5
stack USB....16
state machine....18, 36
sublicensing....25
Suspend....39
T
Texas Instruments....4 e seg.
TM4C12x....5
U
UCFG....7 e seg.
UCON....8
Universal Serial Bus....4
UPUEN....7
USB Implementers Forum....41
usb_config.h....26
usb_device_cdc....28
usb_device_cdc.c....28
usb_device_cdc.h....28
USB_INTERRUPT....18
USB_POLLING....18
USB-UART....11 e seg.
USB.org....25
USBDeviceState....19
USBDeviceTasks....18
USBDeviceTasks()....18
USBDIV....8, 20
USBEN....8
USBGetDeviceState()....19
USBUSARTIsTxTrfReady....28
UTRDIS....8
V
Vendor ID....25
VID....25
Visual Studio .NET....28
VREGEN....7
.
.inf....4

Bibliografia

[1] www.LaurTec.it: sito ufficiale delle schede di sviluppo presentate nel Tutorial, dove poter scaricare i relativi manuali utente ed esempi presentati.

History

Data	Versione	Autore	Revisione	Descrizione Cambiamento
05.11.17	1.1	Mauro Laurenti	Mauro Laurenti	Apportate correzioni varie nel testo.
11.06.17	1.0	Mauro Laurenti	Mauro Laurenti	Versione Originale.