ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

DIPARTIMENTO DEIS

TESI DI LAUREA

in
RETI LOGICHE L-A

Sviluppo e sperimentazione di un modulo USB 2.0 per una telecamera 3D basata su visione stereo

CANDIDATO

Tampellini Mattia

RELATORE:

Ing. Stefano Mattoccia

CORRELATORE:

Prof. Giovanni Neri

Ing. Michele Borgatti

Ing. Davide Nanni

Anno Accademico 2010/2011
Sessione II

INDICE

CAPITOLO I

Introduzione

La realizzazione di sensori in grado di fornire immagini in tre dimensioni ha avuto grande interesse in ambito scientifico ed industriale per via delle innumerevoli applicazioni in cui tali sensori possano essere utilizzati. Gli strumenti teorici utilizzati a questo scopo sono parzialmente acquisiti e in continua evoluzione così come i componenti pratici che ne permettono la realizzazione fisica. Il fulcro fondamentale su cui si basano questi dispositivi è la visione stereoscopica, finalizzata alla ricostruzione della struttura tridimensionale di una scena osservata da più telecamere. In particolare il nostro interesse si è focalizzato sulla visione binoculare che utilizza solamente due sensori di immagine.

Il principio utilizzato dalla visione stereo è la triangolazione, con lo scopo di mettere in relazione la proiezione di uno specifico punto della scena su due (o più) piani immagine delle telecamere, i punti ottenuti con questa procedura vengono chiamati punti omologhi. L'individuazione dei punti omologhi (problema noto come problema delle corrispondenze o "matching") consente, attraverso la conoscenza di opportuni parametri (ottenibili mediante il processo di calibrazione), di risalire alla posizione nello spazio 3D del punto considerato.

Il matching può essere eseguito "on board" cioè direttamente sul dispositivo che si occupa dell'acquisizione immagini oppure da un Host tipicamente un PC.

Un dispositivo molto utilizzato ed efficiente per eseguire elaborazione di immagine on board è l'FPGA (Field Programmable Gate Array). In questi componenti elettronici digitali la funzionalità è programmabile via software direttamente dall'utente finale, consentendo così la diminuzione dei tempi di progettazione, di verifica mediante simulazioni e di prova sul campo dell'applicazione.

Di fondamentale importanza risulta essere anche la scelta del protocollo di trasmissione con l'Host PC, sia in termini di velocità di trasmissione sia in termini di diffusione sul mercato.

L'evoluzione tecnologica nel campo elettronico e informatico ha portato alla nascita nel 1995 dell'architettura USB. Tramite l'interfaccia USB è possibile connettere facilmente ad un'unica unità di elaborazione un elevato numero di dispositivi e di avere a disposizione un canale con un'ampia banda di trasmissione.

La presente tesi si colloca all'interno di un progetto complesso finalizzato allo sviluppo di una telecamera stereo con elaborazione dei dati on board e dotata di interfaccia USB 2.0.

L'obbiettivo specifico del lavoro svolto è stato lo studio con il conseguente sviluppo hardware e software di un particolare controller USB 2.0, finalizzato alla successiva integrazione del medesimo nel progetto complessivo.

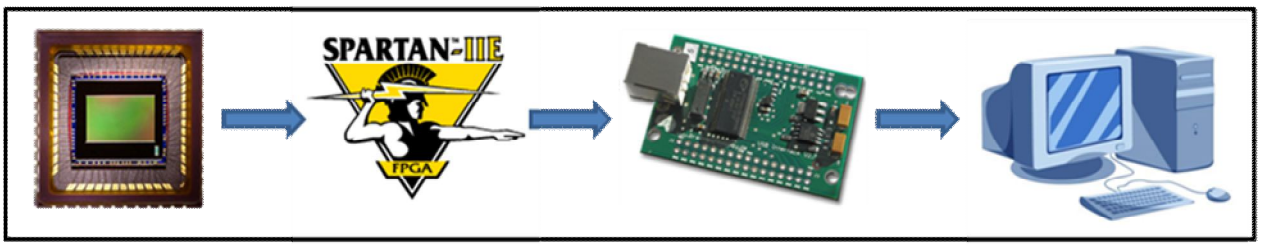

Figura 1.1 Flusso dati tra sensori ed Host

I.2 Organizzazione della tesi

Il lavoro svolto si articola in diverse fasi:

La prima parte della tesi è stata dedicata allo studio del materiale riguardante l'USB, di cui nel *CAPITOLO 2* è presente una panoramica riguardante le caratteristiche principali. Si rimanda alla lettura delle specifiche dell'USB 2.0 per un'analisi più approfondita [1].

Il passo successivo svolto nel *CAPITOLO 3* è stato quello di analizzare in maniera dettagliata il componente Cypress FX2 capace di implementare il protocollo USB per PC e di interfacciarsi ai dispositivi esterni.

La spiegazione dell'architettura complessiva della telecamera stereo e la descrizione dei singoli dispositivi che la compongono è riportata nel *CAPITOLO 4*.

Il CAPITOLO 5 riporta in dettaglio quelli che sono stati gli aspetti realizzativi del progetto di una telecamera stereo sviluppata presso il DEIS e nell'ambito della quale il mio lavoro si inquadra.

Nel *CAPITOLO 5* sono illustrate le problematiche e le soluzioni sviluppate per l'implementazione del Firmware del controller FX 2.

Il *CAPITOLO 6* descrive i risultati e le prove pratiche effettuate durante la fase di sperimentazione.

Le conclusioni e gli sviluppi futuri sono riportati nel *CAPITOLO 7*.

Infine, nell'Appendice A, è presente un chiarimento sul bus I2C, mentre nelle appendici B e C viene riportato una parte del codice sviluppato.

CAPITOLO II

Il protocollo USB 2.0

USB è l'acronimo di Universal Serial Bus, è un protocollo di trasmissione seriale asincrono bidirezionale ad alta velocità in grado di garantire il passaggio di dati e della alimentazione.

L'USB nasce nel 1995 dalla volontà di numerosi ed importanti costruttori di hardware, di trovare e sviluppare nuove tecnologie in grado di rendere più agevole ed efficace la connessione delle varie periferiche con il PC. L'USB interessa quindi dispositivi di input come tastiere, mouse, scanner ecc.. ma anche di output come stampanti, dischi fissi esterni ecc..

Allo stato attuale l'Universal Serial Bus è di fatto il protocollo maggiormente utilizzato nei sistemi a media bassa velocità per i seguenti principali mativi:

- E' economico
- Non necessita di particolari configurazioni da parte dell'utente
- Consente il collegamento di più dispositivi in un'unica porta
- Il sistema "HOT PLUG" permette la connessione e disconnessione delle periferiche in qualsiasi momento
- Il sistema "PLUG AND PLAY" permette il riconoscimento dinamico della periferica grazie a un complicato sistema software presente nell'Host.

II.2 Funzionamento generale

L'USB non è semplicemente un bus ma è un complesso sistema di interconnessioni tra periferiche collegate a stella avente centro in un PC (detto H*ost* del sistema).

L' Host ha il compito di interrogare la periferica e gestire l'intera transazione; questo permette di creare periferiche abbastanza semplici mantenendo così un prezzo competitivo.

Esistono tuttavia dispositivi USB PEAR TO PEAR che implementano il protocollo "ON THE GO", sono utilizzati per avere una connessioni tra due periferiche USB senza l'intervento di un Host; in questo caso la complessità delle singole periferiche aumenta e con essa anche il loro costo.

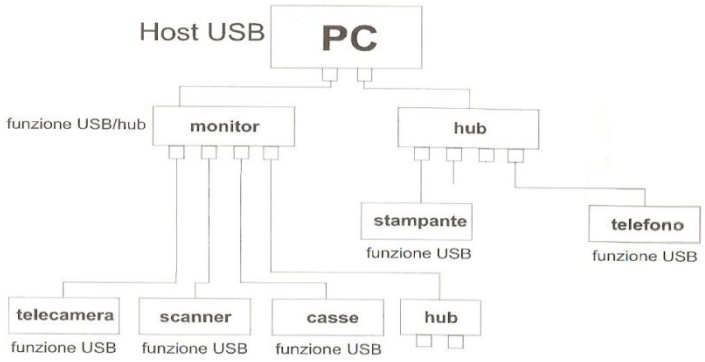

Figura 2.1 Schema ad albero

Il protocollo USB è in grado di gestire fino a 127 periferiche connesse ad albero, occorre però tenere presente che la banda viene suddivisa tra i vari dispositivi e che la priorità dipende dalla posizione nella catena. Una periferica può essere connessa all'Host o in modo diretto oppure attraverso un H*ub* che ha il semplice compito di moltiplicare le porte disponibili.

Come in ogni rete è presente un limite sulla massima distanza tra l'Hub e la

periferica connessa (fissato a 5m). Questo in realtà, non costituisce un limite effettivo poiché l' USB nasce come protocollo di trasmissione punto a punto per trasmissioni di breve distanza.

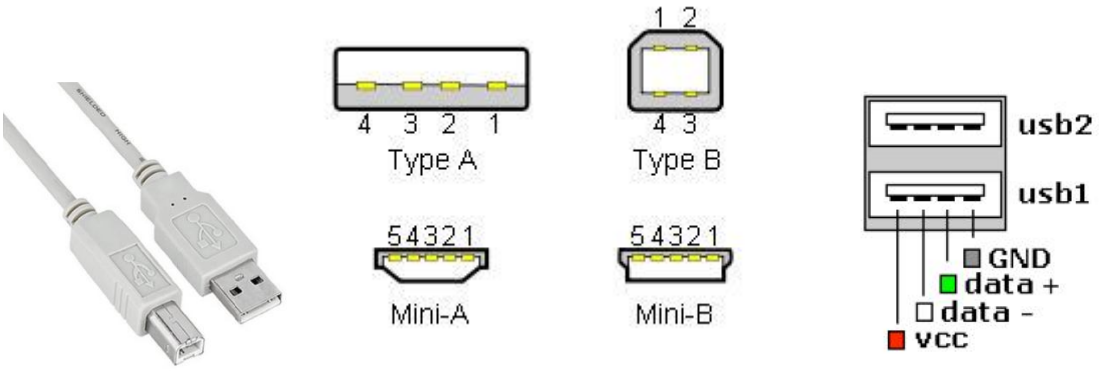

Figura 2.2 Connettori USB

Il cavo utilizzato nello standard USB è l'insieme di quattro fili, due dei quali costituiscono la vera e propria linea di trasmissione differenziale (D+, D-), mentre gli altri due sono posti a potenziale costante: uno fissato a GND e l'altro a +5V.

L'Host è perciò in grado di alimentare le periferiche (a bassa potenza, circa 500mA) attraverso lo stesso cavo utilizzato per la trasmissione dati.

Il software per la gestione della comunicazione presente nell'Host ha un ruolo essenziale nel protocollo USB, ha infatti il compito di interporsi tra il controller USB e i programmi applicativi. Detiene il controllo assoluto delle informazioni scambiate tra le applicazioni e le periferiche connesse e monitora le attività sul bus provvedendo a instaurare i flussi di comunicazione.

Di seguito viene spiegato brevemente come l'Host riesce a identificare e configurare la periferica al momento della connessione. Questa procedura viene chiamata enumerazione ed è suddivisa nei seguenti passi:

1. La periferica altera la tensione nelle sue linee differenziali: se la linea positiva (D+) viene portata sopra la tensione di soglia per più di 2,5μs è connessa una periferica veloce (2.0), viceversa se è la linea differenziale negativa (D-) a essere portata sopra soglia vuol dire che la periferica è a bassa velocità (1.0). Notare che nel funzionamento a regime una delle due linee D è sempre superiore a una tensione di soglia mentre l'altra è vicina ad una tensione di riferimento, l'informazione viene propagata attraverso la variazione della tensione differenziale.
2. L'Host riconosce una nuova connessione ed interroga il dispositivo che risponderà con alcune informazioni utili per la sua programmazione.
3. L'Host assegna un indirizzo alla periferica che passa quindi nello stato di periferica indirizzata.
4. L'Host può quindi cominciare la programmazione della periferica che inizialmente si trova in uno stato di default.
5. La periferica è infine pronta all'uso.

II.3 Lo standard USB

Il protocollo USB è molto flessibile e prevede tre diverse velocità di trasferimento in base all'esigenza della periferica:

Low Speed (1.5 Mbits/sec) tipicamente utilizzata per mouse e tastiere.

Full Speed (12 Mbits/sec) tipicamente utilizzata per modem scanner.

High Speed (480 Mbits/sec solo USB 2.0) tipicamente utilizzata hard disk cd-rom, flussi video.

La comunicazione logica tra client software (Host) e periferica è effettuata tramite

pipe. Un pipe è un'associazione tra uno specifico *endpoint* sulla periferica e l'appropriato software sull'Host. Un endpoint è la sorgente o la destinazione dei dati trasmessi sul cavo USB. Un'interfaccia è composta da un insieme di endpoints raggruppati. Il client software trasmette dati tra il buffer dell'Host e l'endpoint nella periferica, gestendo la specifica interfaccia.

L'intera comunicazione è divisa in finestre temporali di 1 ms chiamate Frame. Le transizioni USB avvengono tramite l'invio seriale di un insieme di bit chiamati pacchetti. Ogni trasferimento è composta da tre fasi:

- *Token phase*: l'Host inizia il pacchetto indicando il tipo di transizione
- *Data phase*: i dati attuali sono trasmessi tramite pacchetti. La direzione dei dati coincide con la direzione indicata nella fase di token trasmessa precedentemente
- *Handshake phase* (opzionale): è il pacchetto inviato indicante il successo o il fallimento della transizione

Quando l'Host vuole ricevere dati da una periferica le invia un token, se la periferica ha dati da inviare li invia e l'Host risponde con un pacchetto di handshake, se invece la periferica non ha dati da inviare, l'Host emetterà un token verso la periferica successiva.

Nel caso invece che sia l'Host a volere inviare dati ad una periferica, le invia prima un token appropriato e successivamente il pacchetto di dati. La periferica risponde con un pacchetto di handshake.

II.3.1 Il package identifier (PID)

Ogni pacchetto è diviso in campi di dimensione 8 bit (o multipli di 8) ma ne esistono anche di diversa misura.

Tutti i pacchetti iniziano con un campo di sincronizzazione (SYNC); viene poi trasmesso un campo a 8 bit identificativo del tipo di pacchetto chiamato package identifier (PID).

Esistono quattro tipi di PID che a loro volta sono differenziati in altri sottotipi in base alla loro funzionalità:

Token: I pacchetti di questo tipo possono essere inviati solo dall'Host, si suddividono in: OUT indica che i dati successivi saranno trasmessi dall'Host alla periferica; *IN* indica che i dati successivi saranno trasmessi dalla periferica all' Host; *SOF* segnala lo start del frame; *SETUP* significa che i dati successivi saranno inviati dall'Host alla periferica e contengono comandi di setup usati per la configurazione.

Data: DATA0, DATA1 sono identificativi del pacchetto dati corrente; in un una transizione vanno alternati per verificare che non si perda un pacchetto.

Handshake: Sono pacchetti che riportano solo il campo PID e sono utilizzati per riportare lo stato di una transazione, si dividono a loro volta in quattro tipi: *ACK* significa che il ricevitore ha accettato il pacchetto senza errori; *NAK* il pacchetto non è stato accettato; *STALL* significa che

l'endpoint è bloccato e non è possibile ricevere il comando; *NYET* viene inviato quando il pacchetto è stato ricevuto con successo ma l'endpoint non è pronto per un nuovo trasferimento.

Special: PRE, ERR e SPLIT servono per abilitare il traffico con periferiche lente; PING è un piccolo pacchetto che viene inviato per verificare la disponibilità della periferica che, se pronta, risponderà con *acknowledge*.

II.3.2 Formato dei pacchetti

Il formato dei pacchetti cambia a seconda del tipo:

TOKEN PACKET: Ogni transizione inizia con l'emissione di un token da parte dell'Host. I campi ADDR e ENDP definiscono univocamente l'endpoint che deve ricevere i dati di SETUP o OUT o l'endpoint che deve trasmettere dati negli spostamenti IN.

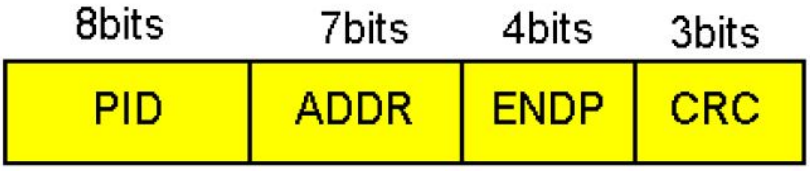

Figura 2.3 Token packet

START OF FRAME PACKET: l'Host emette un SOF ogni millisecondo. Il pacchetto contiene il campo del numero del frame. SOF può essere usato come trigger per processi di OUT isocrono

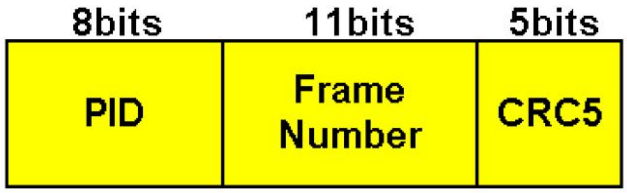

Figura 2.4 Start of Frame Packet

DATA PACKETS: sono composti da PID (che indica che il pacchetto contiene dati), campo dei dati, e il codice CRC16 per proteggere i dati

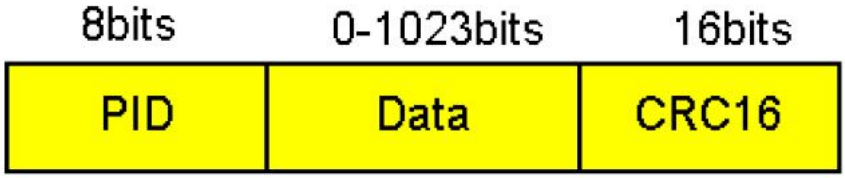

Figura 2.5 Data Packets

HANDSHAKE PACKETS: composto solo da PID indicante il risultato dello operazioni precedenti

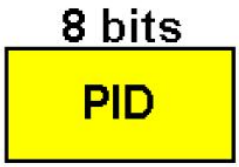

Figura 2.6 Handshake Packets

Di seguito è riportata un esempio di trasmissione dati, come si può vedere per ogni blocco dati inviato (DATA0, DATA1 vanno alternati) è necessario un token packet e un handshake packet.

I campi in rosso e quelli in giallo rappresentano il sottotipo del PID del pacchetto.

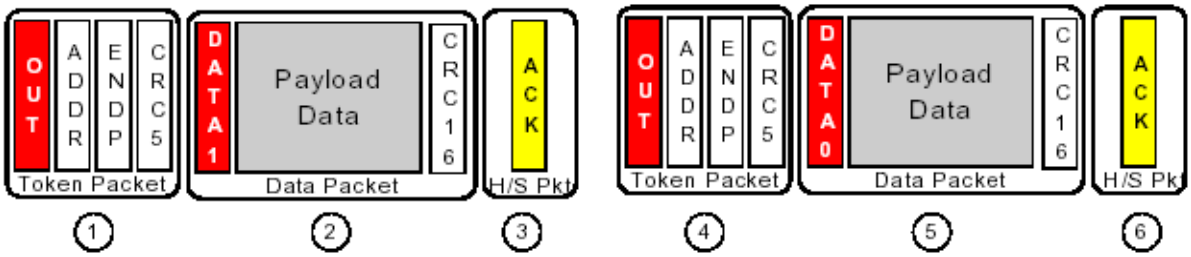

Figura 2.7 Struttura dei pacchetti

II.3.3 Modalità di trasferimento

Lo standard USB prevede quattro diverse modalità di trasferimento dati:

BULK: Utilizzata per trasmissioni discontinue in cui è tollerata una latenza e non ci sono vincoli restrittivi sulla temporizzazione, per esempio per stampante e scanner. Prevede pacchetti dati da 8, 16, 32 o 64 bytes (full speed) o 512 bytes (high speed), è possibile la ritrasmissione in caso venga rilevato un errore ed è implementato il protocollo di Handshaking (il trasmettitore prima dell''invio di nuovi dati aspetta l'Handshake packet).

INTERRUPT: In questo caso viene fatto un polling da parte dell'Host sull'interrupt endpoints che viene attivato dalla periferica quando è pronta per l'invio di nuovi dati. Questa modalità di trasmissione permette una reazione dell'Host molto veloce infatti il tempo di latenza è sempre garantito. Viene utilizzata una stream pipe unidirezionale e viene implementata una ritrasmissione in caso di verifica di errore. La modalità di trasferimento interrupt viene utilizzata per tastiera e mouse. La dimensione dei pacchetti di dati varia da 1 a 46 bytes (full speed) oppure 1024 bytes (high speed).

ISOCRONOUS: E' utilizzata quando i dati devono arrivare a una velocità costante per potere mantenere una temporizzazione con il tempo reale, esempi tipici di dati isocroni sono musica e voce. Non è presente un sistema di ritrasmissione automatica in caso di errore altrimenti si perderebbe la sincronizzazione, tuttavia sono tollerati alcuni errori occasionali. Nel trasferimento isocrono, a differenza di quello bulk, viene garantita una banda sul bus in termini di numero di byte ogni frame ma non è implementato l'Handshaking. I pacchetti possono arrivare ad una dimensione di 1024 bytes (high speed).

CONTROL: L'unica transazione che prevede un flusso di dati bidirezionale, questi dati hanno il compito di comandare e configurare il dispositivo, data la sua estrema importanza si implementa un sistema di controllo errori.
Questo tipo di trasferimento è suddiviso nell'invio di 3 pacchetti all'interno dei quali si realizza la richiesta. Vengono inviati un Setup Token con indirizzo e numero dell'endpoint, un pacchetto data che include un pacchetto Setup con i dettagli della richiesta. Infine un pacchetto handshake con indicazione di errore o successo.

Come già anticipato alcune connessioni utilizzano un sistema di controllo degli errori, in particolare si implementa il CRC (Cycling Redundancy Check). Il CRC è un particolare algoritmo che il trasmettitore utilizza per creare alcuni bit di controllo in base ai dati da inviare; il ricevitore costruisce i suoi bit di controllo utilizzando i

dati ricevuti poi li confronterà con quelli creati dal trasmettitore se sono uguali è probabile che i dati ricevuti siano corretti.

II.4 Driver

Il driver è un particolare software che fa da appendice al sistema operativo, rendendolo capace di usare e gestire efficientemente un particolare dispositivo Hardware periferico, senza conoscere come esso funzioni ma dialogandoci con una interfaccia standard.

Il concetto di driver è relativamente recente. Nei primi sistemi informatici, inclusi i primi personal computer, erano infatti le applicazioni stesse a gestire i dispositivi che esse utilizzavano. Ma questa filosofia era svantaggiosa per scrivere le applicazioni era molto più complesso e richiedeva programmatori con conoscenze approfondite circa gli standard dei dispositivi e dei meccanismi dei sistemi operativi, in secondo luogo, nel momento in cui si doveva sostituire il dispositivo, era molto difficile dover cambiare per ogni applicazione tutte le impostazioni riguardanti l'uso del dispositivo.

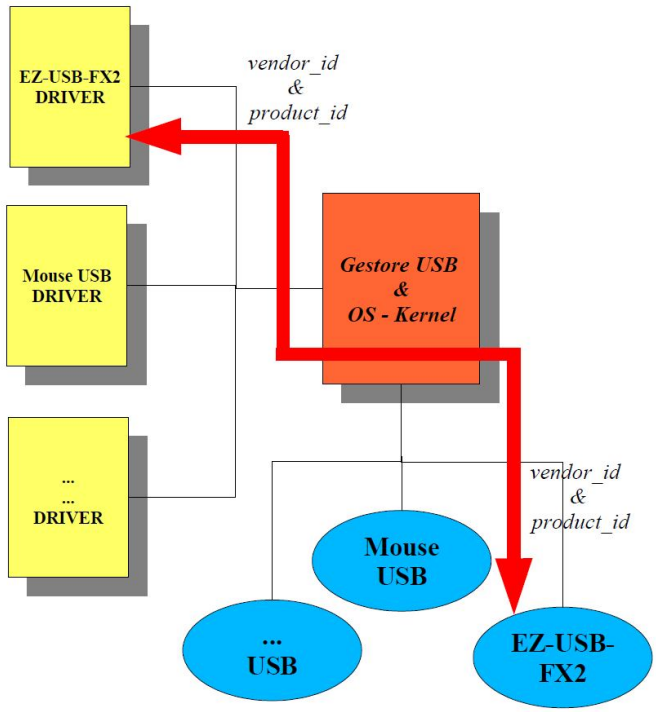

Figura 2.8 Driver

La filosofia odierna prevede, invece, che le applicazioni ignorino quali siano i dispositivi effettivamente utilizzati e che parlino tutte un linguaggio standard. E'

compito invece del sistema operativo tradurre i comandi inviati al dispositivo virtuale nei giusti comandi comprensibili al particolare dispositivo reale usato. Questo è appunto il compito principale del driver che costituisce effettivamente l'unica parte di software che è a conoscenza sia del sistema operativo adottato che delle caratteristiche tecniche del dispositivo reale.

Nel protocollo USB il driver viene caricato durante la fase di enumerazione, questa procedura cambia in base al sistema operativo che si sta utilizzando.

Il sistema operativo Windows cerca all'interno di una particolare archivio di file di testo (file .INF) che contengono informazioni sulle periferiche e i driver ad esse associati. Nel caso in cui il Vendor Identifier (Vendor Id) ed il Product Identifier (Product Id) della periferica coincidano con quelli memorizzati in un file INF viene caricato il driver associato (file .sys). Nel caso in cui non venga riscontrato nessun match, Windows, cerca di utilizzare un driver della stessa classe o sottoclasse.

Il driver, se correttamente istallato, mette a disposizione un set di librerie esterne composto da un insieme di funzioni API (in inglese *Application Programming Interface*) che consentono al programmatore un maggiore livello di astrazione dalle specifiche caratteristiche del dispositivo utilizzato.

II.5 Applicazioni utente

Nel lavoro svolto durante questa tesi sono stati presi in considerazione due diversi tipi di driver: quello sviluppato da Cypress e quello messo a disposizione da LibUsb. Di seguito vengono riportate le principali caratteristiche di entrambi, verrà posta maggiore enfasi sul driver che verrà effettivamente utilizzato cioè su Libusb.

II.5.1 SuiteUSB

Cypress fornisce un pacchetto comprensivo di tutti gli elementi necessari per lo sviluppo delle applicazioni utente chiamato SuiteUSB [8].

All'interno di questo kit di sviluppo troviamo il file .INF, il driver CyUSB.sys, le API messe a disposizione e alcuni programmi eseguibili (Cyconsole e Control center) per eseguire una prima fase di test.

E' compito del programmatore modificare il file .INF in modo tale che il sistema operativo utilizzi il driver Cypress. In pratica è necessario andare a modificare il file (di testo) .INF inserendo i campi Vendor Id e Product Id specifici della nostra periferica (nel nostro caso 04B4 Hex e 8613 Hex).

All'interno della Suite Usb è presente la Control Center che risulta essere, soprattutto nelle fasi iniziali, un utile tool di sviluppo per il programmatore. Questa utility permette la comunicazione con tutti i dispositivi che utilizzano il driver CyUSB.sys e consente inoltre di caricare in RAM o memoria EEPROM (se presente) il firmware del controller FX2.

Il firmware può anche essere caricato automaticamente da uno script (generato con un particolare procedimento dalla Cyconsole) ogni qualvolta viene collegata la periferica.

Per questo metodo risulta di fondamentale importanza che il nuovo firmware imposti un nuovo Product Id diverso dal precedente altrimenti la periferica continuerà ad essere rinumerata all'infinito entrando in un deadlock.

La finestra della Control Center è suddivisa in due parti: a sinistra vediamo le periferiche connesse con i relativi settaggi di interfaccia e degli endpoint, mentre a

destra troviamo il risultato della trasmissione.

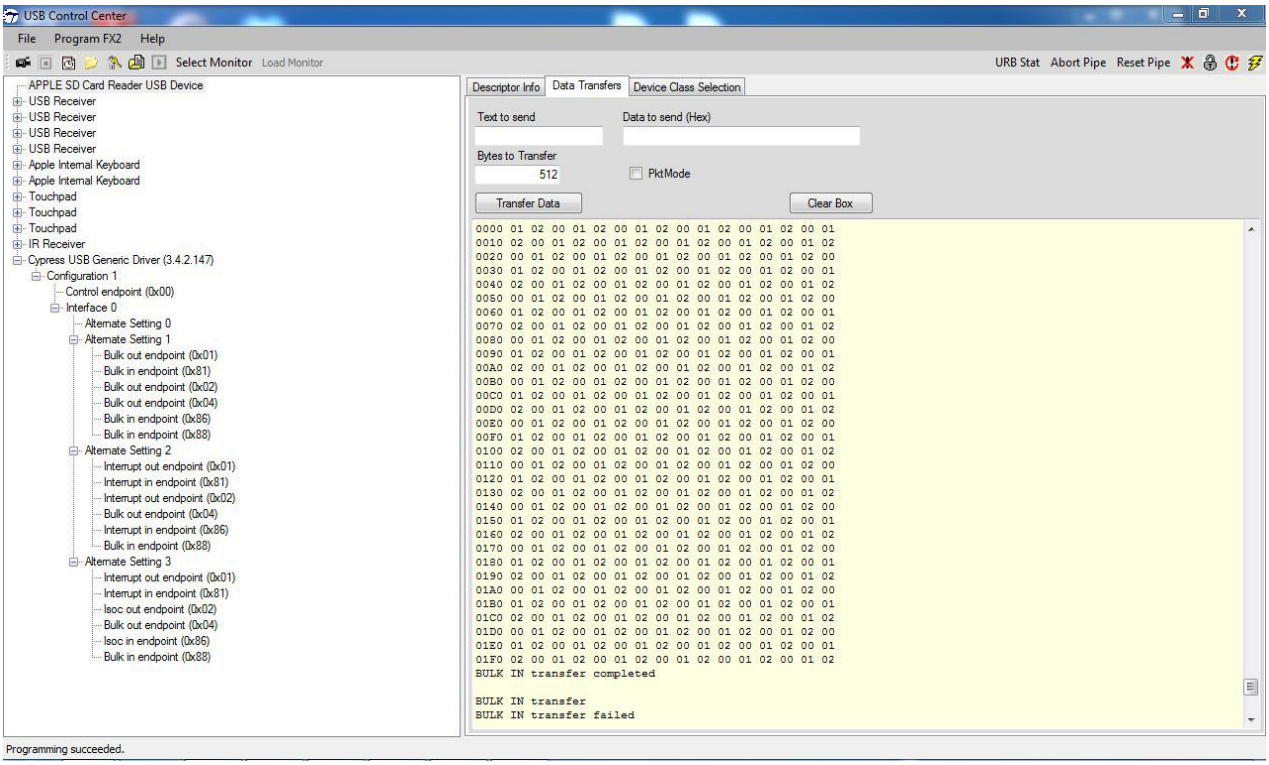

Figura 2.9 Control Center

Per effettuare un trasferimento è sufficiente selezionare un endpoint in ingresso, impostare i byte da ricevere e cliccare su tasto transfer data.

E' inoltre possibile avere ulteriori informazioni sulla configurazione corrente degli endpoint cliccando su descriptor info.

Le librerie messe a disposizione da Cypress nella SuiteUSB sono di due tipi: CyAPI.lib è statica mentre CYUSB.dll è dinamica. Nelle librerie statiche il codice viene inglobato nell'eseguibile in fase di compilazione, invece nelle librerie a link dinamico (DLL), il codice viene caricato in fase di esecuzione nella memoria RAM ed è condiviso con tutte le applicazioni che lo utilizzano con un vantaggio, sia in termini di flessibilità, perché se viene cambiato qualcosa nella libreria i programmi

che la utilizzano non vanno ricompilati ogni volta, sia in termini di velocità perché vengono condivise le risorse comuni.

Nella presente tesi è stata implementata un applicazione utente che sfrutta la CyAPI.lib. Questa libreria è scritta in C++ e fa ampio uso delle classi e dei metodi, che sono appunto caratteristici di tale linguaggio.

Per utilizzare la libreria in questione è necessario includere il file Header CYAPI.h, inoltre bisogna lincare la libreria stessa al progetto.

I test effettuati su Windows con il driver Cypress e con la libreria CyAPI.lib non sono stati soddisfacenti poiché l'elevata complessità del driver, si è in pratica tradotta in una lentezza nella trasmissione dati che ci ha costretti a scartare questa ipotesi.

II.5.2 LibUsb

LibUsb è un progetto open source (con licenza GNU "Lesser General Public License") [2] che mette a disposizione delle librerie applicative che danno accesso, da diversi sistemi operativi, alla totalità delle periferiche USB. Esistono diverse versioni delle librerie e dei driver, sia per ambiente Windows, sia per Linux.

LibUsb, a differenza del driver Cypress, cerca di astrarre il programmatore dalla conoscenza del complesso protocollo USB offrendo una serie di funzioni di alto livello.

Per prima è stata analizzata la libreria WIN32 versione 0.1 per sistemi operativi Windows (è garantita la compatibilità con: Windows98, Windows 2000, Windows XP, Windows Vista and Windows 7); successivamente si è passato alla versione 1.0

su Linux per valutare le differenti performance tra i due sistemi operativi e tra le due versioni.

LibUsb in ambiente Windows mette a disposizione il driver e un utility (Wizard inf) per generare il file .INF specifico della periferica che si sta utilizzando.

Di seguito viene brevemente descritto il codice implementato (in linguaggio C++) per Windows; per una comprensione più approfondita si rimanda all' Appendice B.

Per prima cosa viene allocato un buffer in RAM su cui andranno salvati i dati temporanei:

```
char * buffer = (char*)malloc(sizeof(char)*n_byte*2);
```

Vengono poi dichiarati ed aperi i file su cui andranno salvati i dati definitivi:

```
FILE *out;
out = fopen("example.txt","w");
```

Si inizializzata la periferica e vengono cercati tutti i dispositivi connessi:

```
usb_init();
usb_find_busses();
usb_find_devices();
```

La periferica è gestita tramite un *handler* che è una istanza della classe usb_dev_handle:

```
usb_dev_handle *current_handle;
```

La particolare periferica viene associata con i suo handler attraverso la funzione usb_open:

```
current_handle=usb_open(current_device);
```

E' necessario selezionare la configurazione specifica del dispositivo USB:

```
set_configuration=usb_set_configuration(current_handle, 1);
```

Viene poi selezionata l'interfaccia e la alternative interface:

```
usb_claim_interface(current_handle, 0);
```

```
usb_set_altinterface(current_handle, 0);
```

E’ ora possibile eseguire trasferimenti bulk dati da un endpoint configurato in ingresso:

```
usb_bulk_read(current_handle,ENDPOINT_BULK_IN,buffer+index*PACKET_
            BULK_LEN, PACKET_BULK_LEN, 2000);
```

Segue un successivo controllo sui dati ricevuti (nell'esempio considerato veniva trasmesso un pattern incrementale predefinito) finalizzato alla stampa a video del numero e della posizione degli errori.

Infine chiudiamo i file, liberiamo la ram e rilasciamo la periferica USB:

```
fclose(out);
fclose(tempi);
free(buffer);

usb_release_interface(current_handle, 0);
usb_close(current_handle);
```

Attraverso delle funzioni messe a disposizione dalla libreria OpenCv è stato possibile misurare con estrema precisione i tempi impiegati per i trasferimenti consentendo il calcolo delle velocità medie.

Le API che sono state menzionate appartengono alla versione 0.1 e non sono compatibili con quelle della versione 1.0 per Linux, tuttavia le differenze concettuali sono minime, variano invece le velocità medie misurate dei trasferimenti, che in ambiente Linux arrivano a 41 MB/s mentre in ambiente Windows si attestano a 37 MB/s.

Un ulteriore incremento di prestazioni si è ottenuto inserendo le nuove versioni 1.0 delle API per Windows che consentono velocità di trasferimento misurate di 40 MB/s

L’approccio che è stato utilizzato per lo sviluppo dell’applicazione utente è il multithreading. Il programma risulta essere diviso in tre parti: un thread che si

occupa di leggere da USB (utilizzando le API libusb) e salva i dati ricevuti in RAM, un'altro thread scrive il contenuto della RAM in un file, ed infine il programma principale che ha il compito di creare e sincronizzare i due thread.

Un possibile modo per risolvere l'evidente conflitto di accesso alla RAM che si viene a creare tra i due thread, è l'utilizzo di diversi buffer su cui si può alternativamente scrivere.

Nel momento in cui il thread dedito all'acquisizione da USB riempie un buffer verrà chiamato l'altro thread per svuotarlo mentre i nuovi dati vengono inseriti in un'altro buffer.

Questo tipo di architettura consente l'acquisizione ed il salvataggio dei dati in contemporanea permettendo all'applicazione di lavorare in streaming senza mai doversi fermare.

CAPITOLO III

Il controller USB CY7C68013

Uno degli obbiettivi di questo lavoro di tesi è stato lo studio del controller CYC68013 [3], utilizzato dalla telecamera stereo sviluppata presso il DEIS per inviare immagini all'host e per agire sui parametri della stessa telecamera (per impostare il *frame rate*, le caratteristiche dei sensori di immagine come l'*autogain*, etc). Questo dispositivo implementa un'interfaccia USB 2.0 in grado di sfruttare la banda offerta dall' USB 2.0 in high speed.

Le funzionalità di cui dispone sono molto superiori a quelle necessarie per svolgere il progetto, ma è comunque utile avere una visione più generale di tale dispositivo; cercheremo quindi di studiarlo più approfonditamente dando una descrizione anche dei blocchi che non prenderanno direttamente parte al progetto finale della telecamera stereo.

III.1 Descrizione a blocchi

Nella pagina seguente viene riportata la schematizzazione a blocchi del dispositivo Cypress CYC68013.

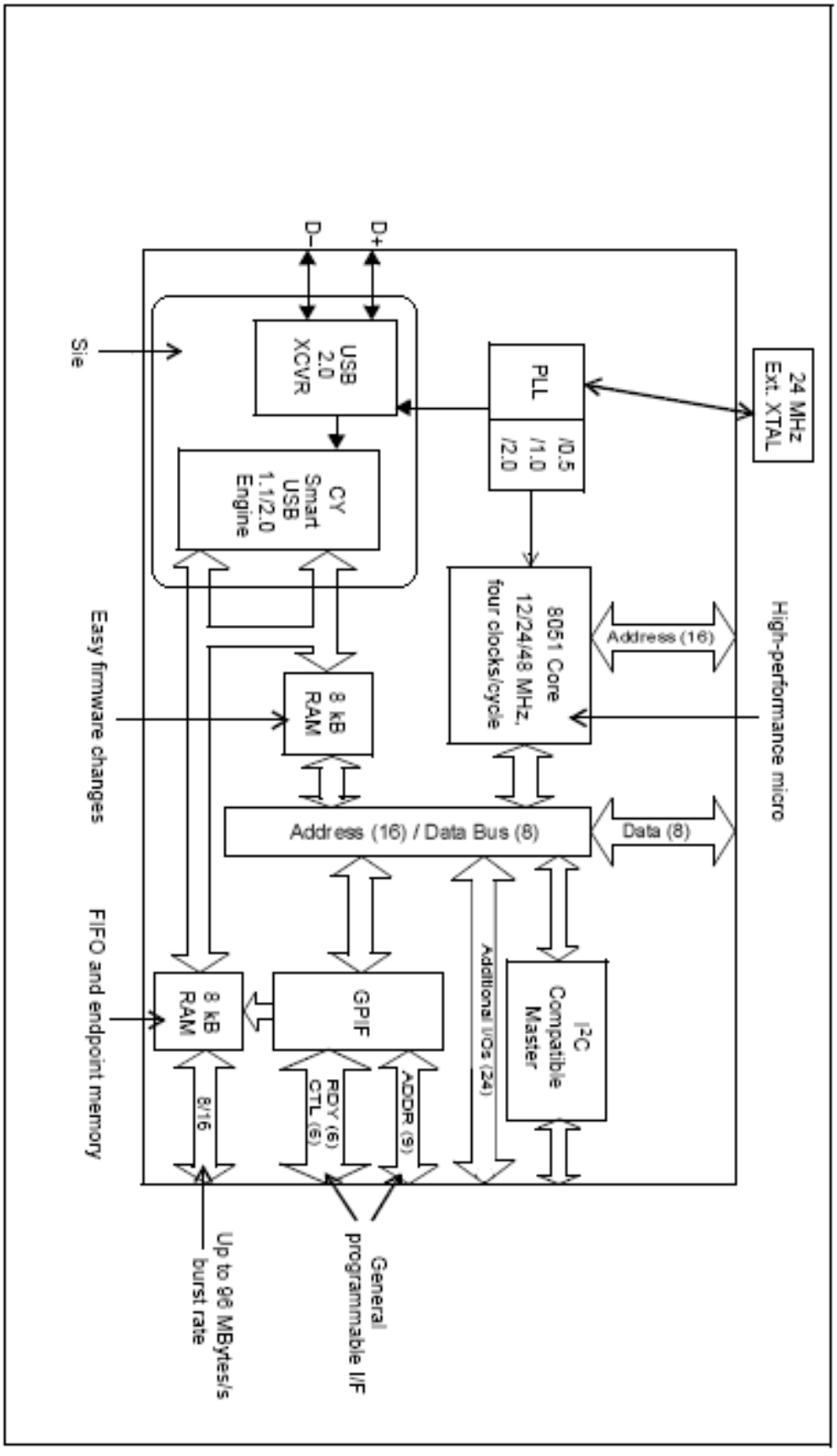

Figura 3.1 Schema a blocchi del CYC68013

SIE (Serial Interface Engine): Rappresenta l'interfaccia fisica che connette la memoria all'USB bus, è perciò direttamente collegata con le linee differenziali D+ e D- attraverso un transceiver (che ha il compito di gestire la direzione dei dati e l'isolamento dal bus).

Questa interfaccia si occupa di: Codificare e decodificare i pacchetti secondo la logica NRZ, gestire il protocollo di handshaking con l'USB Host e della verifica degli errori (error checcking), cioè crea e verifica il CRC e chiede la ritrasmissione in caso di errore (nelle comunicazioni che la supportano). E' abbastanza potente per implementare al suo interno parte del protocollo USB, in questo modo può funzionare in full speed senza *firmware* caricato in ram.

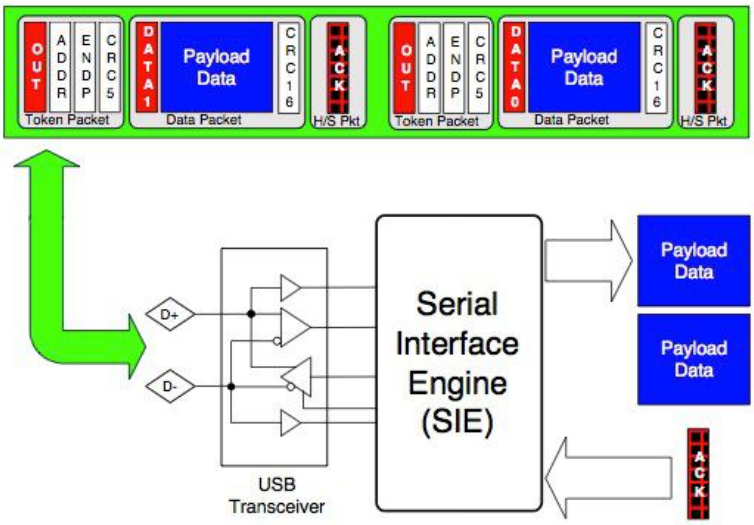

Figura 3.2 Schema della SIE

8051: E' un microcontrolore sviluppato dalla INTEL nel 1980, è diventato ormai di largo consumo grazie al suo basso costo, viene utilizzato in sistemi che non devono avere prestazioni particolarmente elevate. La Cypress ha deciso di affiancarlo al controller USB per creare un sistema più potente e più versatile. Viene utilizzato per gestire gli interrupt, controllare l'attività sul bus, programmare le interfacce di I/O e configurare la SIE per l'inizio della trasmissione. E' dotato di una memoria interna da 256 Byte dove è possibile allocare il programma e una piccola quantità di dati.

L'8051 è in grado di eseguire controlli sui dati in arrivo e in partenza dall'USB ma, data la sua lentezza, non prende parte alle comunicazioni in cui è richiesta velocità elevata (high speed).

GPIF (General Programmable Interface): E' un interfaccia programmabile molto potente, mette a disposizione alcune linee in uscita per il controllo di dispositivi esterni, ha diretto accesso alla memoria e si avvale del supporto dell'8051 per la gestione dei protocolli di comunicazione più complessi.

PLL (Phase Locked Loop): è un circuito elettronico progettato per generare un'onda ad una data frequenza la cui fase ha una relazione specifica con un segnale di riferimento. Una PLL è generalmente composta da tre blocchi:

Oscillatore controllato in tensione

Divisore di frequenza

Comparatore di fase.

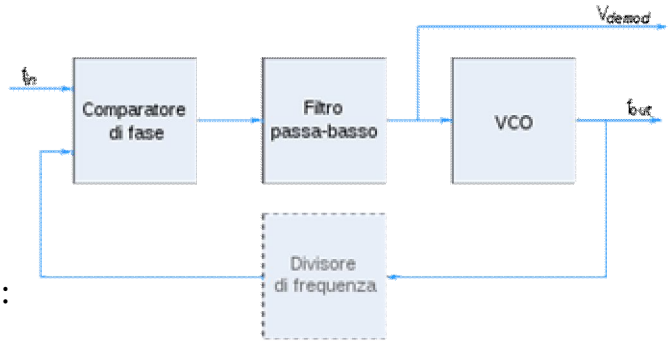

Figura 3.3 Schema PLL

Il funzionamento è il seguente: il segnale in ingresso, cioè quello che condizionerà il funzionamento del VCO, viene inserito nel comparatore di fase, che ne identifica il fronte di salita dell'onda; il VCO viene posto in oscillazione libera ad un valore prossimo a quello che sarà il suo valore di lavoro. Un divisore di frequenza programmabile ricava un segnale sottomultiplo di quello generato dal VCO e lo applica ad un secondo ingresso del comparatore. L'uscita del comparatore sarà una tensione continua che controllerà il

VCO, mantenendone la frequenza rigorosamente agganciata in fase con il segnale entrante.

MEMORIA INTERNA: E' costituita da due banchi di ram ciascuno di dimensione 8 Kbyte, in un blocco vi sono allocati gli endpoint ed il FIFO, nell'altro il firmware e altri dati.

FIFO: E' un buffer (First In First Out) allocato in una specifica parte della RAM, è direttamente comandato dalla logica esterna se programmato in modalità SLAVE, mentre è controllato dalla GPIF quando configurato come MASTER. In quest'ultimo caso la GPIF dovrà essere programmata anche per la gestione dei segnali di controllo per il dispositivo secondario esterno (slave).
Le principali caratteristiche di questo terminatore sono:

1) E' direttamente collegato con il bus USB (grazie alla SIE)
2) La sua architettura "Quantum FIFO" muove quasi istantaneamente i dati dalla SIE all'endpoint FIFO indirizzato
3) E' un Interfaccia versatile: Slave FIFO (external master) or Master FIFO (GPIF internal master)

I^2C (Inter Integrated Circuit): E' un bus seriale creato dalla Philips per collegare periferiche a media-bassa velocità, oggi è in grado di trasferire dati ad una frequenza massima di 400kHz. Questa interfaccia utilizza due linee seriali: Serial Data (SDA)

e Serial Clock (SCL) entrambe open drain, necessitano quindi di due resistori per il pull up.

E' presente un solo master nel bus che ha il compito di generare il clock di linea (*SCL)* per tutti i dispositivi slave collegati al bus ed è l'unico che può selezionare con quale periferica dialogare e in quale direzione.

SDA rappresenta la linea di scambio dati, come verrà spiegato in seguito deve rispettare una certa sincronizzazione con SCL.

III.1.1 Caratteristiche principali

Lo studio del manuale tecnico fornito dalla Cypress comprende le caratteristiche di tre diversi modelli: 56, 100, 128 pin.

Si è preferito l'utilizzo del dispositivo a 56 pin perché più semplice ed idoneo ai compiti che gli andranno affidati. Questo capitolo prende comunque in considerazione anche le funzionalità presenti negli altri due modelli.

Per maggior chiarezza riportiamo ora le caratteristiche del 56 pin package:

- Tre porte di I/O A,B,D da 8 bit multiplexate per diversi utilizzi.
- FIFO configurabile con 8 o 16 bit di bus dati (porta B e D), 5 bit di controllo non multiplexate più 4 o 5 bit multiplexate con la porta A.
- GPIF per l'implementazione di diversi protocolli di trasmissione.
- Un bus I^2C compatibile.
- Un sistema di suspend mode per il risparmio energetico.
- Una complessa gestione degli interrupt interni ed esterni.

Per sottolineare le diverse potenzialità delle tre versioni del CYC68013 viene riportato di seguito uno schema che mette a confronto le tre interfacce fornite dai diversi modelli di questo controller.

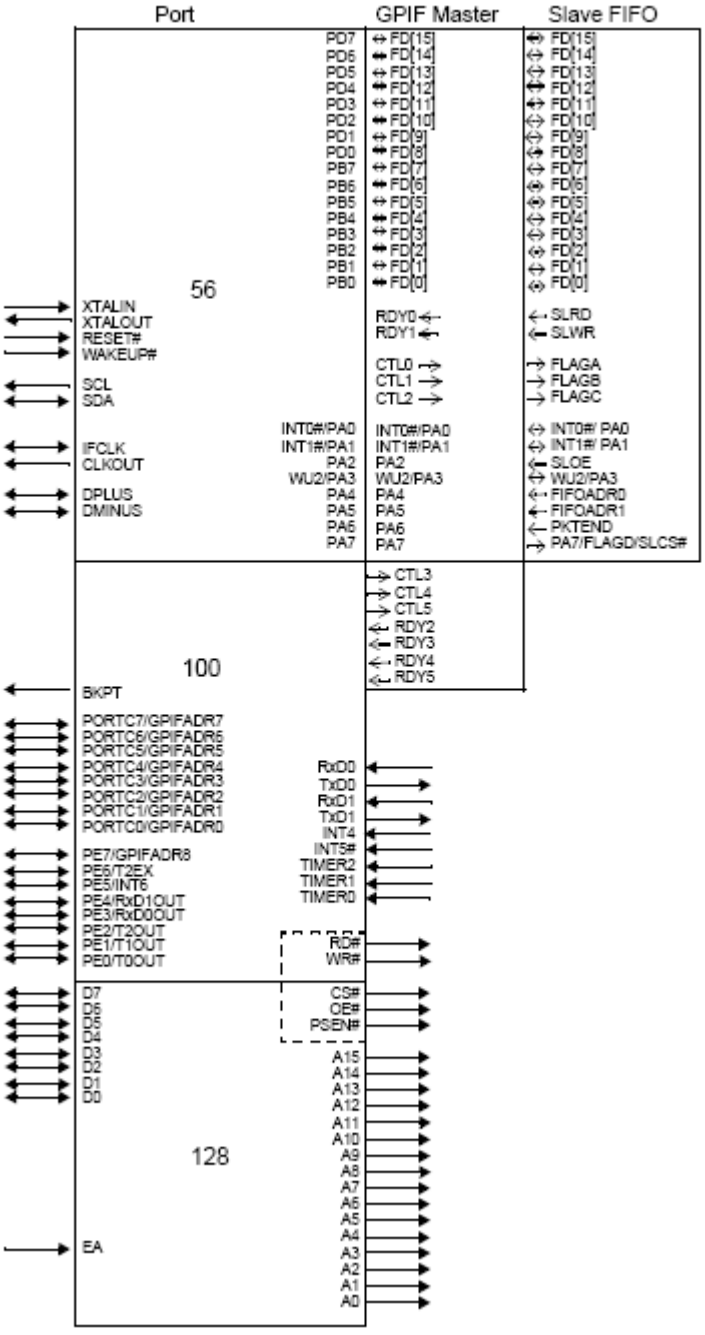

Figura 3.4 Schema di confronto delle le versioni del CYC68013

III.2 Numerazione e rienumerazione

Una periferica USB 2.0 per essere rilevata pone la linea differenziale D+ a 3,3V con una resistenza di valore 1,5k, nell' Host invece questo pin è pull down a massa con una R di 15K, in questo modo l' Host avverte la variazione di tensione e la interpreta come nuova connessione.

Per creare una disconnessione software la periferica può lasciare D+ fluttuante, l'Host non vede così l'alterazione della tensione.

Vediamo ora cosa succede nel momento in cui colleghiamo il controller al computer remoto (Host) attraverso l'USB (plug in del dispositivo):

1) Viene dato un reset generale attraverso il pin di RESET (chiamato power on reset) per mettere il controller USB in uno stato noto, normalmente questo pin viene controllato da una rete RC opportunamente dimensionata che provvede a fornire il reset non appena viene fornita l'alimentazione.
2) L'Host manda all'endpoint zero del dispositivo appena collegato una richiesta di identificazione, questa parola di comando viene chiamata Get Descriptor-Device.
3) La periferica risponde mandando delle stringhe chiamate ID (identifier data), tra questi dati troviamo le tre informazioni di cui si era già discusso in precedenza:

   Vid (Vendor Identifier) nel nostro caso la Cypress

   Pid (Productid Identifier) è il tipo di modello, nel nostro caso "ez USB2 fx2"

   Did (Device Releasid) è un parametro che dipende dal singolo dispositivo.

   Quando l'Host viene in possesso dei dati, controlla nel database (che ha in

memoria) per capire di che tipo di dispositivo si tratta e di quali altre informazioni avrà bisogno per trovare il driver più adatto.

4) L'Host invia all'endpoint zero un particolare comando (Set Address) per assegnare un unico indirizzo alla periferica.
5) Vengono inviati dei successivi Get Descriptor per avere maggiori informazioni sul numero di endpoint, tipo di alimentazione, classe, velocità, tipo di driver più adatto ecc...
6) L'Host manda un comando all'endpoint zero finalizzato a fermare l'attività della CPU.
7) Viene poi caricato il firmware sulla ram del controller USB in due possibili modi:
    - Nel caso in cui non sia presente memoria fisicamente esterna eprom o I^2C, il CYC68013 manda all'Host i vid/pid/did che ha allocati nella memoria interna seguiti dalla richiesta di invio firmware. L'Host invia il firmware all'endpoint zero della periferica, provvederà poi il controller a spostarlo nella specifica parte della ram.
    - Nel caso in cui venga riconosciuta una memoria esterna collegata tramite il bus I^2C oppure tramite il bus di I/O standard (presente solo nella versione 128 pin) il programma interno al processore controlla il primo byte di tale memoria ed in base al suo valore, o carica direttamente dalla memoria esterna il firmware, oppure carica solamente i vid/pid/did e li invia all'Host insieme alla richiesta di invio firmware.

8) L’ultimo comando inviato dall’Host pone il registro cpucs a zero per permettere alla CPU di eseguire il codice scaricato.

   La procedura che è appena stata descritta provoca una disconnessione e una riconnessione software (processo di rienumerazione) in questo modo il PC vede una periferica già programmata e pronta alla comunicazione.

III.2.1 Reset

Il reset ha una funzione fondamentale nei circuiti elettronici, ha infatti il compito di mettere tutti i blocchi che lo compongono in uno stato logico conosciuto per rendere non alleatoria l’esecuzione di una certa funzione. Il CYC68013 prevede diversi tipi di reset:

Power on Reset: Controllato dal reset pin viene dato subito dopo l’alimentazione per mettere tutti i registri in stati conosciuti, normalmente è attivato da logica esterna (rete RC) ma può essere utilizzato anche da un altro dispositivo che in qualsiasi momento può abbassarlo.

Cpu Reset: Il registro cpucs permette lo spegnimento dell’8051 interno.

Bus Reset: Attivato abbassando entrambe le linee D+ e D- per almeno 10ms, genera un USB interrupt (INT2).

La seguente tabella riassume gli effetti dei tre tipi di reset e di una disconnessione fisica della periferica dall’Host.

	RESET Pin	CPU Reset	USB Bus Reset	Disconnect
CPU Reset	Reset	n/a	—	—
IN Endpoints	Unarm	—	—	—
OUT Endpoints	Unarm	—	—	—
Breakpoint	0	0	—	—
Stall Bits	0	—	—	0
Interrupt Enables	0	0	—	—
Interrupt Requests	0	—	—	—
CLKOUT	Active	—	—	—
CPU Clock Speed	12 MHz	—	—	—
Data Toggles	0	—	0	0
Function Address	0	—	0	0
Default USB Device Configuration	0	—	0	0
Default USB Device Alternate Setting	0	—	0	0
RENUM Bit	0	—	—	—

** Il trattino indica che non vengono apportati cambiamenti*

Figura 3.5 Tabella dei reset

III.3Architettura della memoria interna

La memoria è suddivisa in tre distinte aree: Internal Data Memory, External Data Memory e External Program Memory (le memorie esterne non sono necessariamente fisicamente fuori dal controller).

- *Internal Data Memory* (256 byte) fisicamente interna all'8051 divisa in tre parti:

  - The lower (128 byte) contiene: uno spazio utilizzabile come general purposes, uno indirizzabile bit a bit e otto registri.

  - The upper (128 byte) è di uso generale contiene lo stack ed è indirizzabile solo in modo indiretto.

  - SFR indirizzabile solo in modo diretto, ha lo stesso indirizzo della upper 128 ma si utilizzano due modi differenti per accedervi, contiene i registri di stato e di controllo alcuni dei quali sono indirizzabili bit a bit.

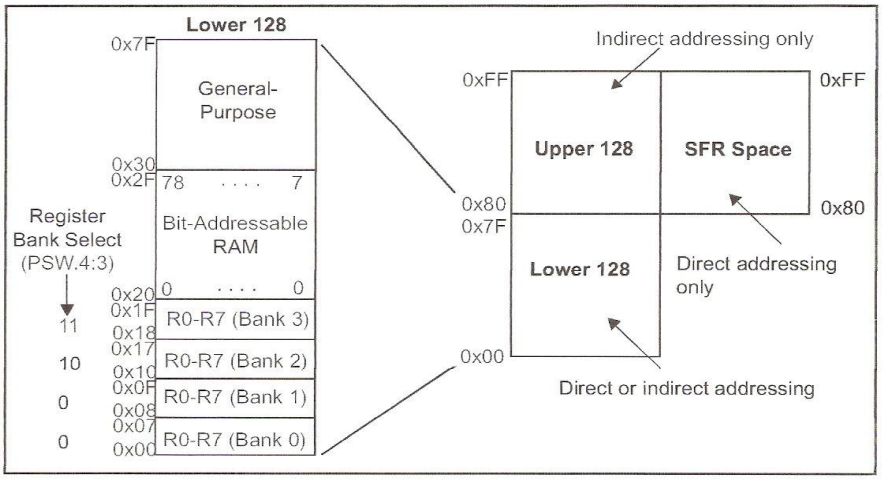

Figura 3.6 Schema memoria interna all'8051

- *External Data Memory e External Program Memory*

  Il CYC68013 possiede 8 kbyte di main RAM utilizzata per dati e programma e 512 byte di RAM utilizzabile solo per dati, entrambe sono indirizzabili dal firmware; possiede inoltre una memoria da 7,5 kbyte riconosciuta come data memory, contiene gli endpoint buffer, GPIF waveforms e alcuni registri di stato e di controllo. Queste tre parti di memoria sono fisicamente interne alla scheda ma sono viste dall' 8051 come esterne.

Il dispositivo 128 package prevede anche l'interfacciamento verso memoria fisicamente esterna attraverso un bus dati da 8 bit e un bus indirizzi a 16 bit; la memoria esterna di programma e quella dati possono avere una dimensione massima di 64 Kbyte. La memoria dati e quella di programma restano fisicamente divise perché l'8051 le indirizza usando differenti segnali di controllo (PSEN attivo per selezionare la memoria di programma).

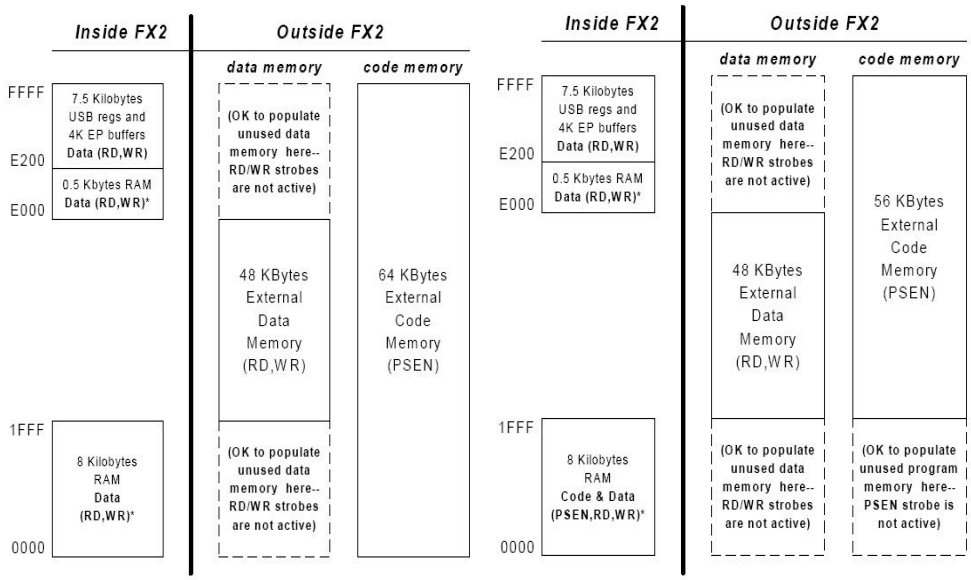

Figura 3.7 Spazi di indirizzamento memoria fisicamente esterna

Ci sono due possibili modi di organizzare la memoria esterna in base al segnale EA se è a uno il programma è interamente allocato esternamente (nella *code memory*) e la main RAM è dedicata solamente ai dati, notare che la memoria dati esterna va mappata tra gli indirizzi 2000 e DFFF perché gli altri indirizzi sono occupati dalla memoria dati interna.

Mentre se EA è a zero la main RAM contiene sia il programma sia dati, in questo caso c'è un duplice conflitto sia con la data memory sia con la code memory esterne come mostrato in figura 3.7.

III.4 Il sistema di Interrupt

Il sistema di interrupt è interamente gestito dall'8051, i timer interni e le interfacce seriali generano interrupt settando gli appositi interrupt flag che vengono campionati dal processore una volta per ciclo di istruzione (cioè una volta ogni quattro clock),

gli interrupt esterni INT0 e INT1 possono essere configurati a livello (attivi bassi) o sul fronte di discesa. La latenza di servizio agli interrupt varia dallo stato in cui si trova il controller, può variare da un minimo di cinque ad un massimo di tredici cicli di bus.

Ci sono 27 interrupt dedicati all'USB (INT2), 17 al GPIF/FIFO (INT4), 1 al wakeup, alcuni dedicati al bus I^2C (INT3) e altri dedicati all'interfaccia seriale standard (solo 100 e 128 package).

Questa tabella mette a confronto gli interrupt forniti dal CYC68013 (Fx2) e quelli forniti dall'8051 standard mostrando in particolare cosa viene aggiunto nel controller.

Standard 8051 Interrupts	Additional FX2 Interrupts	Source
INT0		Pin PA0 / $\overline{\text{INT0}}$
INT1		Pin PA1 / $\overline{\text{INT1}}$
Timer 0		Internal, Timer 0
Timer 1		Internal, Timer 1
Tx0 & Rx0		Internal, USART0
	INT2	Internal, USB
	INT3	Internal, I²C-Compatible Bus Controller
	INT4	Pin INT4 (100- and 128-pin only) **OR** Internal, GPIF/FIFOs
	INT5	Pin $\overline{\text{INT5}}$ (100- and 128-pin only)
	INT6	Pin INT6 (100- and 128-pin only)
	WAKEUP	Pin WAKEUP or Pin RA3/WU2
	Tx1 & Rx1	Internal, USART1
	Timer 2	Internal, Timer 2

Figura 3.8 Tabella degli interrupt

Il CYC68013 implementa un secondo livello di decodifica degli interrupt chiamato *AUTOVECTOR*.

Per spigare meglio questo meccanismo verrà utilizzato come esempio un USB interrupt (INT2). Normalmente al momento di questo tipo di chiamata viene messo

sullo stack il program counter, si salta poi all'indirizzo 0x0043, dove ci si aspetta di trovare la jump instruction verso la routin di servizio all'interrupt.

Se l'Autovectoring è attivo, l'FX2 carica in INT2VEC il tipo di USB interrupt questo valore viene automaticamente copiato all' indirizzo 0x0045, si salta poi all'indirizzo 0x0043, dove però ci sarà un'altra jump istruction verso la jump table, l'indirizzo a cui saltare è calcolato concatenando 0x04 con il valore presente all'indirizzo 0x0045, il contenuto di tale indirizzo è USB Interrupt source ISR.

La figura 3.9 riassume questo meccanismo.

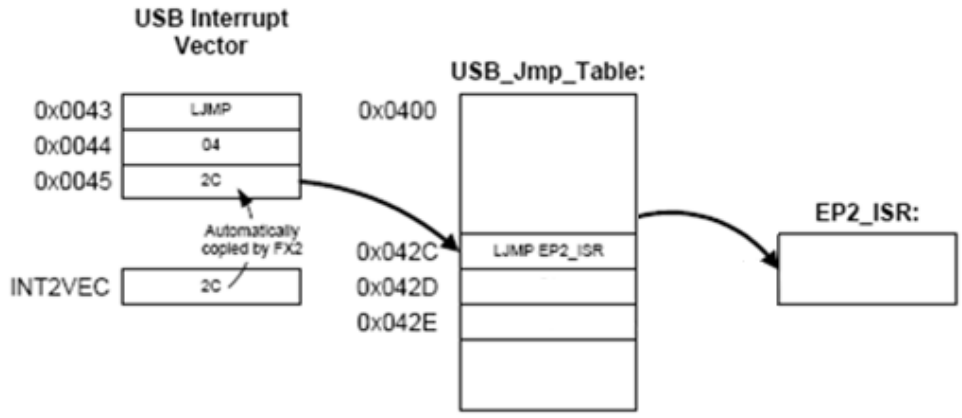

Figura 3.9 Funzionamento dell'interrupt autovector

Questo complesso sistema di interruzioni permette di cambiare in modo dinamico (grazie al registro USB interrupt vector) l'indirizzo che punta alla jump table, ottenendo cosi un secondo livello di decodifica.

III.5 Slave e master FIFO

Il termine FIFO è acronimo inglese di "First In First Out" (il primo ad entrare è il

primo ad uscire), esprime quindi la modalità con cui vengono immagazzinati i dati. Praticamente un FIFO viene creato utilizzando un endpoint presente in RAM opportunamente configurato.

Esistono due modalià di funzionamento quando il buffer è configurato come slave:

- Modalità SINCRONA in questo caso i segnali di controllo sono sincronizzati con un clock che viene generato dal master esterno.
- Modalità ASINCRONA non cè nessun sincronismo, viene solamente passato il segnale di strobe (slwr,slrd).

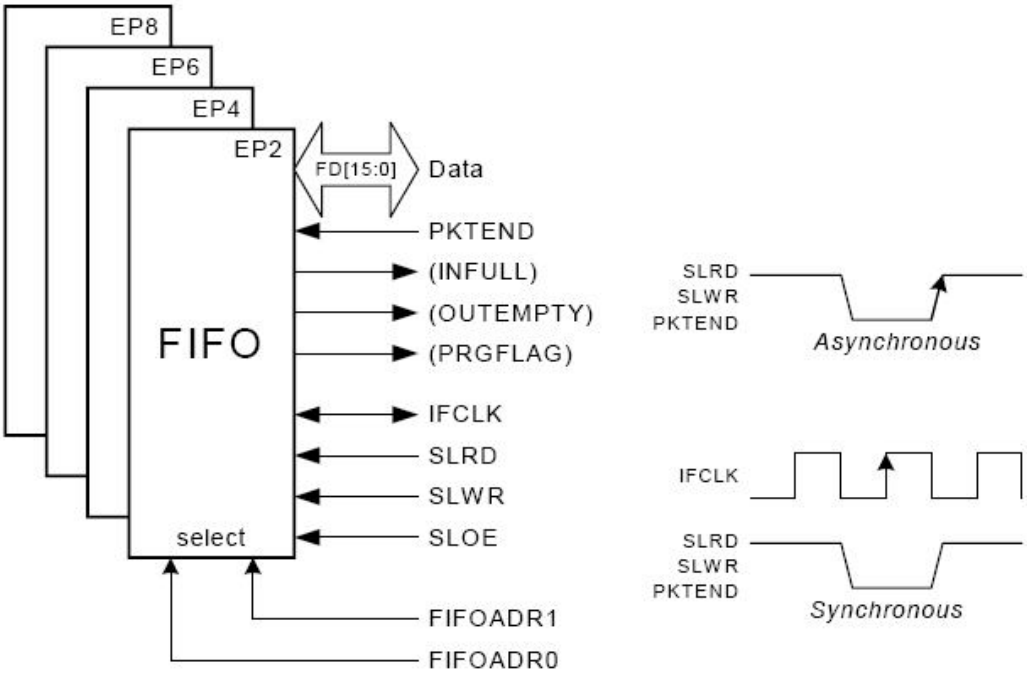

Figura 3.10 Interfaccia verso l'esterno dello slave FIFO

Per controllare questo buffer il dispositivo pilota esterno (master) agisce sui seguenti segnali (multiplexati con i pin di I/O generali): IFCLK, SLCS#, SLRD, SLWR, SLOE, PKTEND mentre il FIFO controlla alcune linee (chiamate flag) per indicare

il suo stato al master.

Vediamo le funzionalità di queste linee in dettaglio:

FD[15,0] E' il bus dati a 16 bit viene utilizzata la parte bassa della porta D e la parte alta della porta B, tale bus è configurabile anche a 8 bit, in questo caso, si utilizza solo la porta B mentre la porta D rimane disponibile come I/O generale. La dimensione del bus dati viene modificata agendo su un particolare registro chiamato WORDWIDE, se è a zero il bus è a 8 bit altrimenti è a 16 bit.

FLAG Sono delle linee che rappresentano lo stato dello slave FIFO, in figura sono rappresentate dai segnali infull, outempty e prgflag; i primi due segnalano rispettivamente buffer pieno e buffer vuoto mentre la terza linea è in realtà composta da più linee chiamate flag programmabili (flaga, flagb, flagc, flagd).

A questi flag possiamo attribuire diversi significati in base alle esigenze del master esterno.

I flag programmabili possono essere configurati in due modi indexed o inndex; nel primo caso viene selezionato un particolare endpoint e questi flag rappresentano il suo stato :

- flagA riporta il "programmable-level" status
- flagB full status
- flagC empty status

Quando sono invece settati come fixed i flag sono configurabili dal programmatore autonomamente e posso rappresentare diversi stati di diversi endpoint, una possibile configurazione può essere la seguente:

- flagA empty FIFO 6
- flagB empty FIFO 4
- flagC FIFO 4 programmable level
- flagD FIFO 6 full

Tutti i flag sono attivi bassi ma la polarità può essere invertita settando un registro dedicato.

SLOE Ha il compito di abilitare le uscite del bus dati del FIFO.

SLRD, SLWR Sono i segnali di controllo di lettura e scrittura:

i dati vengono scritti sul FIFO sul fronte di salita di ifclk con slwr attivo se la comunicazione è sincrona, mentre se è asincrona, vengono scritti in seguito alla transizione di slwr (fronte di salita). Una cosa analoga accade per la lettura con slrd.

E' presente uno speciale contatore chiamato FIFO pointer che ha il compito di puntare all'indirizzo dell'ultimo dato scritto o letto.

PKTEND Come abbiamo già detto il FIFO è direttamente collegato all'USB grazie alla SIE, i pacchetti scritti sul endpoint FIFO vengono automaticamente inviati alla SIE appena raggiungono la dimensione giusta (che dipende dal tipo di trasmissione). Se vogliamo inviare dati più piccoli del formato USB standard dobbiamo riempire i pacchetti con dati fasulli fino alla dimensione corretta oppure dare il comando di packet end.

SLCS Permette al master di rimuovere il controller dal bus; in questa situazione vengono ignorati i comandi slrd, slwr, ifclk ecc...

FIFOADR[1,0] Sono due bit utilizzati per selezionare uno, dei quattro endpoint, con cui comunicare.

CLOCK Può essere configurato come interno a 30 o 48 Mhz oppure come esterno, in quest'ultimo caso il clock viene passato dall'esterno attraverso il pin IFCLK con l'unico vincolo che sia compreso tra 5 e 48 Mhz.

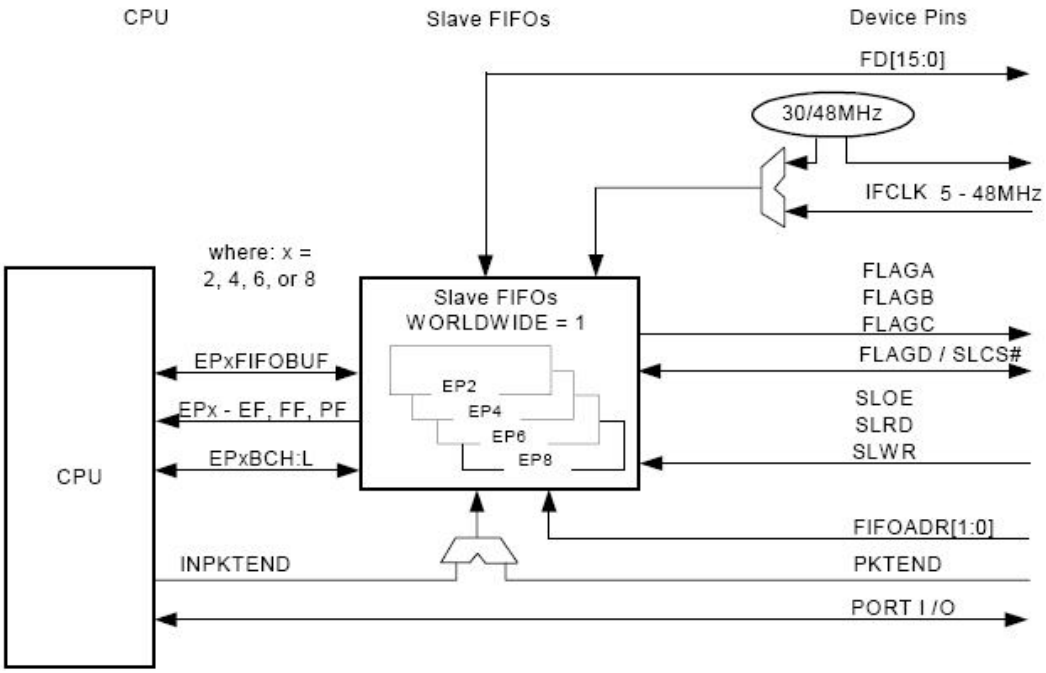

Figura 3.11 Schema logico slave FIFO

Il firmware accede fisicamente ai vari FIFO utilizzando i seguenti registri a 16 bit: EP2FIFOBUF, EP4FIFOBUF, EP6FIFOBUF e EP8FIFOBUF che possono essere letti o scritti usando un istruzione di tipo MOVX, sono presenti inoltre altri registri per lo stato ed il controllo.

Nel progetto è stata testata sia la modalità sincrona che quella asincrona per confrontarne le prestazioni. Per chiarire ulteriormente le idee vediamo riepilogati i segnali di interfaccia necessari per una scrittura e una lettura da slave FIFO in entrambi i casi.

Figura 3.12 Scrittura e lettura da slave FIFO asincrona

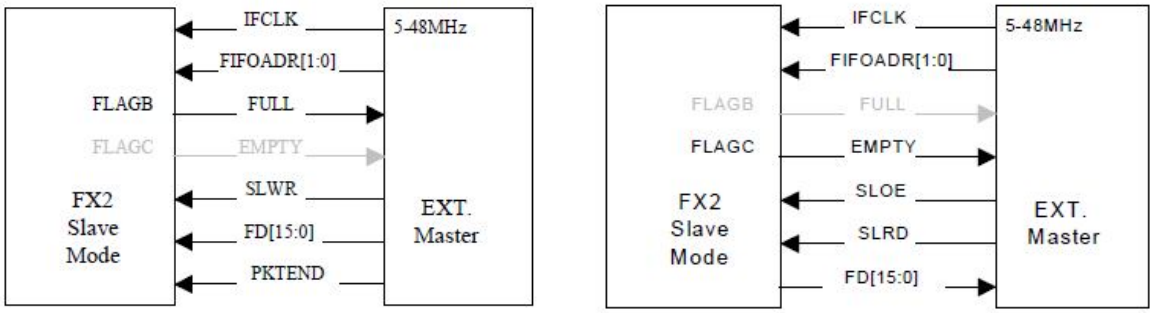

Figura 3.13 Scrittura e lettura da slave FIFO sincrona

Di seguito è riportata invece la modalità di master FIFO, in questo caso la GPIF ha il compito di gestire i comandi per il FIFO e generare i segnali di controllo (CTL) del dispositivo esterno slave.

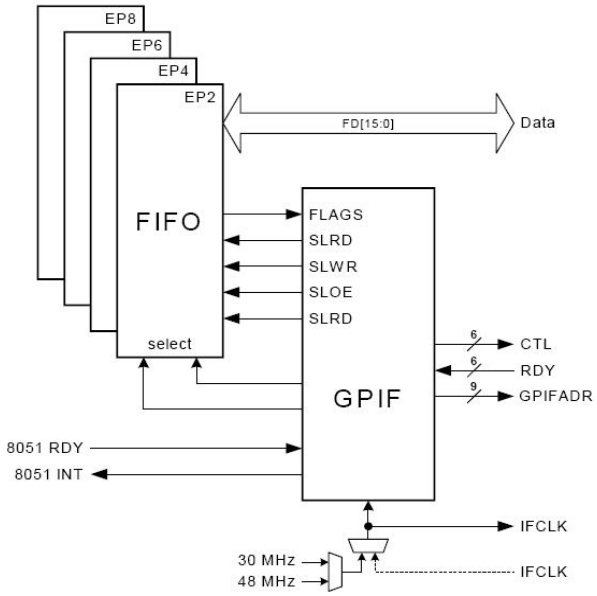

Figura 3.14 Master FIFO

La GPIF è una macchina programmabile a stati (FSM) che viene utilizzata per creare un interfaccia tra il CYC68013 ed il mondo esterno. Nella versione 56 pin è in grado di controllare:

- 3 bit di controllo programmabili (CTL) normalmente utilizzati come rd, wr, en da inviare al dispositivo slave sterno
- 2 external ready sono segnali utilizzati dallo slave per inserire stati di attesa nella GPIF
- 2 internal ready inviati dall'8051 alla GPIF

Le versioni 100 e 128 pin implementano anche:

- 9 bit di indirizzamento esterni (GPIFADR); queste linee vengono ricavate multiplexando la porta C e parte della porta E
- 3 bit per segnalare in quale degli otto stati si trova la GPIF (GSTATE porta E[2,0]) utilizzati per la verifica del corretto funzionamento
- Altri 3 bit di controllo
- Altri 4 bit di external ready

La GPIF viene normalmente utilizzata per generare i segnali di controllo (in modalità master FIFO) da inviare al dispositivo slave esterno. Non si limita però all'implementazione di un semplice handshaking, ma è abbastanza potente da supportare protocolli complessi come atap, ieee1284, utopia .

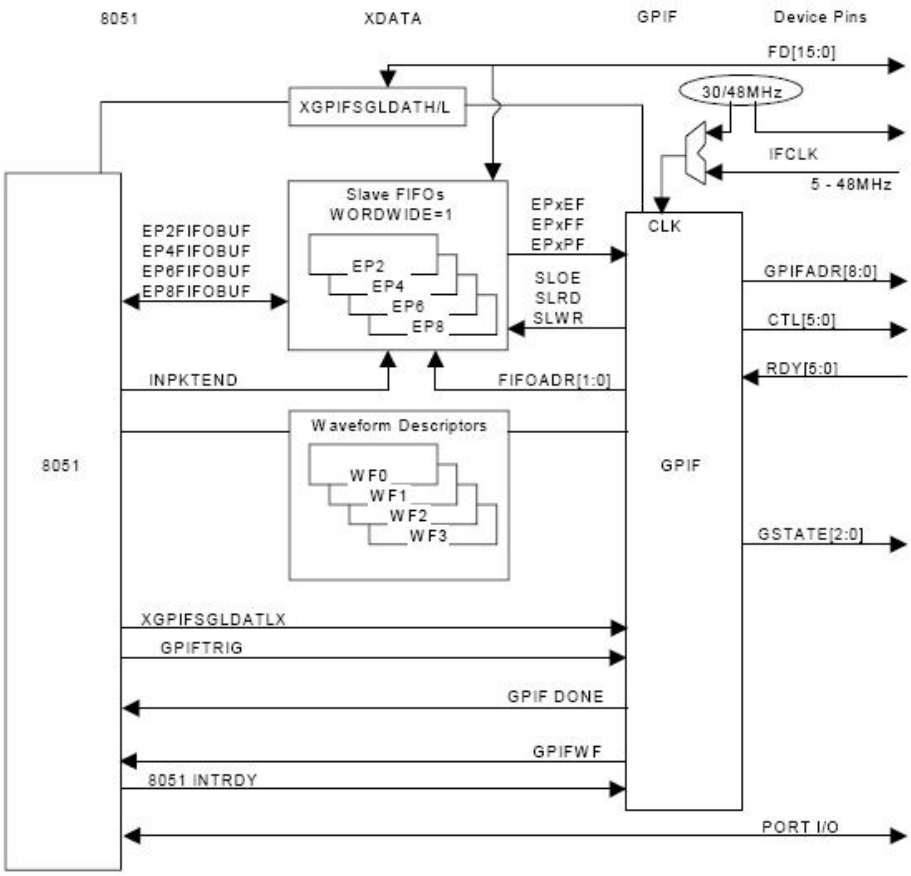

Figura 3.15 Schema della GPIF

Per implementare la macchina a stati finiti la GPIF utilizza quattro registri allocati in ram chiamati *WAVEFORM DESCRIPTOR; a*l momento dell'inizializzazione il firmware li copia automaticamente nella memoria interna alla GPIF per un utilizzo più veloce.

Questi registri hanno il compito di descrivere l'andamento che ogni segnale deve assumere nelle varie fasi della trasmissione; di default sono in grado di implementare single write, single read, FIFO write, FIFO read, ma le varie funzionalità possono essere cambiate e rese più complesse.

Un singolo WAVEFORM DESCRIPTOR è composto da sette blocchi chiamati *State Instructions* rappresentate da 4 byte ciascuna.

Ogni istruzione di stato definisce quali devono essere i valori dei segnali di

controllo, e quali ready campionare per decidere lo stato futuro. La transizione tra uno stato all'altro avviene normalmente sul fronte di salita di ifclk ma si può restare in uno stato per più clock.

Ci sono sette stati (in corrispondenza con le sette istruzioni di stato) da s0 a s6 più uno stato particolare s7 chiamato idle state che viene utilizzato per terminare la connessione.

III.5.1 Modalità auto out

Il controller CYC68013 può essere utilizzato sia come elaboratore dati ma anche come una semplice interfaccia USB, in questo ultimo caso la commissione dei pacchetti tra FIFO e SIE è automatica e interamente gestita via hardware.

In alcune applicazioni è però opportuno inserire la CPU nel flusso dati, in questi casi si parla di modalità di commissione manuale.

Tale modalità di trasferimento prevede che sia generato un interrupt nel momento in cui sia presente (in un endpoint) un pacchetto pronto per essere spedito. La CPU può quindi decidere tramite lo SKIP bit se commissionare oppure no questo pacchetto alla SIE.

La modalità automatica permette comunicazioni più veloci perché esclude la lenta CPU interna, ma perde di flessibilità, poiché non è possibile controllare o modificare i dati in transito dall' USB.

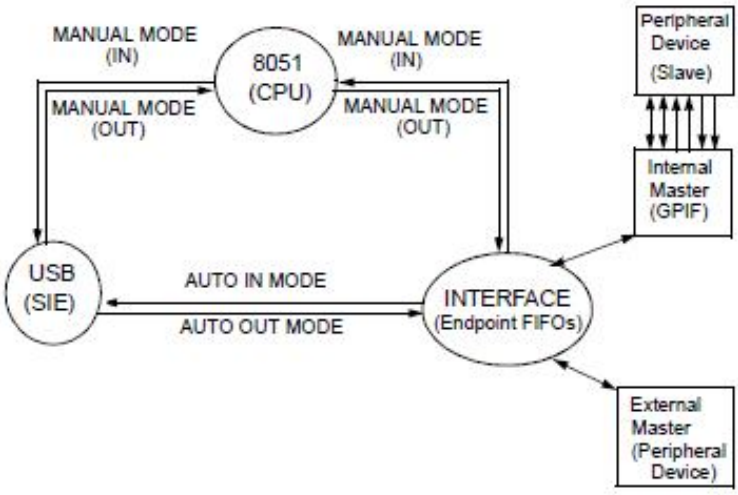

Figura 3.16 Modalità auto out/in

Il nostro obbiettivo è cercare di sfruttare al meglio la banda messa a disposizione dall'USB 2.0, per questa ragione è stata utilizzata la modalità auto in.

III.5.2 Accesso agli endpoint buffer

Un endpoint, come già precedentemente è stato definito, è una terminazione di un flusso di comunicazione tra Host e periferica. In pratica si tratta di un buffer dove vengono messi i dati in arrivo e in uscita. Di default sono attivi tre endpoint accessibili dalla CPU ma non dalla logica esterna:

- *END0*= E' l'unico endpoint che supporta un flusso dati bidirezionale, ha una dimensione di 64 byte. In pratica è il buffer in cui l'Host manda i segnali di controllo e dove viene inviato il firmware,
- *END1IN/OUT*= Sono due distinti buffer: uno in ingresso e uno in uscita, sono utilizzati per il trasferimento dati a bassa velocità, vanno configurati in base al tipo di trasmissione (supportano solo bulk o interrupt).

Esistono inoltre altri endpoint (END2,END4, END6, END8) allocati nella ram da 4kbyte ma devono essere specificatamente abilitati dal firmware.

Questi endpoint hanno dimensione variabile, supportano connessioni bulk, interrupt, o isochronous e possono essere configurati come doppio triplo o quadruplo buffer per velocizzare la connessione (vanno configurati in base al tipo di trasmissione cioè alla grandezza dei pacchetti altrimenti c'è uno spreco di risorse).

I campi da impostare durante la configurazione sono i seguenti:

- *direzione:* ingresso o uscita dal punto di vista dell'Host
- *tipo di trasmissione:* bulk, interrupt o isocronous
- *dimensione:* configurabili come 64, 513 oppure 1024 Byte
- *buffering:* doppio triplo o quadruplo; permette il caricamento in parallelo di più pacchetti
- *enable*: abilitazione dell'endpoint

A differenza di END0, ed END1 gli END2,END4, END6, END8 sono controllati senza l'intervento diretto del firmware.

La logica esterna al controller scrive sull'endpoint FIFO attraverso una connessione diretta senza l'utilizzo della CPU; il firmware può accedervi o come speciale blocco di ram o attraverso uno speciale auto-increment pointer.

Come già detto l'endpoint 0 è un buffer di controllo dove l'Host manda i pacchetti di controllo (sutup token).

Esistono due differenti modi per accedere a EP0 in scrittura:

- Tramite il Setup Data Pointer dove viene caricato l'indirizzo di partenza,

vengono poi configurati i registri EP0BCH:L con il numero totale di byte da trasferire e l'Fx2 trasferisce sequenzialmente tutti i byte spezzettandoli nella misura di pacchetto standard (per un control transfer 64 byte).

- In modo diretto caricando il pacchetto in EP0BUF e settando il numero di byte da trasferire in EP0BCH:L.

  In una transazione OUTtransfert in cui EP0 deve essere letto, i due registri EP0BCH:L contengono al loro interno il numero di byte che l'Host ha inviato su EP0BUF.

III.7 Il microcontrollore 8051

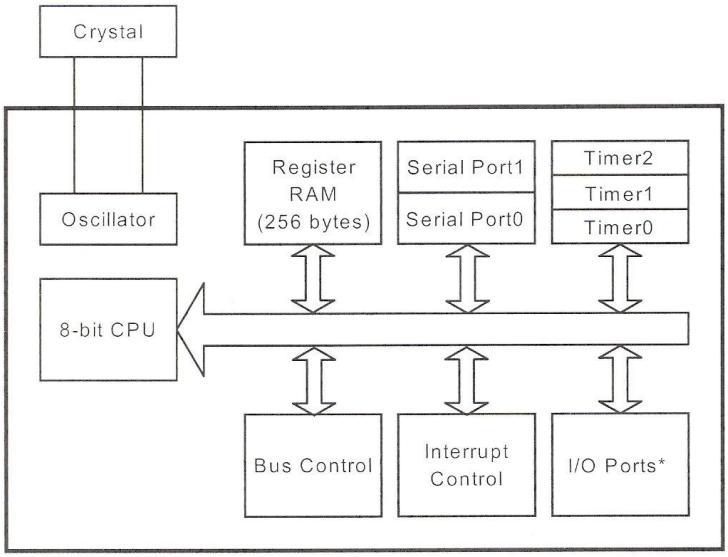

Figura 3.17 Schema interno 8051

Il microcontrollore 8051 presente all'interno del controller FX2 presenta caratteristiche leggermente diverse da quelle dell'8051 standard pur utilizzando lo stesso instruction set (per essere compatibile con gli stessi compilatori).

La differenza più rilevante è l'aumento della velocità di esecuzione delle istruzioni dovuto alle seguenti modifiche:

- Un ciclo di bus impiega 4 clock contro i 12 dello standard 8051.
- Per eseguire alcune istruzioni nell'8051 del controller possono essere necessari più cicli di bus ma data la sua velocità di accesso al bus rimane notevolmente più veloce dell' 8051 standard (di tre volte circa).
- la cpu ha una frequenza di clock che varia tra 12-24-48 Mhz cioè fino a 4 volte più veloce dell'8051 standard.

Oltre alla velocità aumentata vengono aggiunti anche:

- un altro data pointer
- una seconda usart (indirizzabili con due sfr)
- un terzo timer da 16 bit
- una interfaccia per memoria esterna con 16 bit di indirizzamento non multiplexati
- due autopointer (si autoincrementano a ogni accesso)
- vector USB and FIFO/GPIF interrupt
- sleep mode con tre sorgenti di wakeup
- I^2C compatibile bus serial controller da 100 a 400kHz
- alcuni SFR
- due porte seriali

I compiti a cui è dedito l'8051 interno al controller sono i seguenti:

- Configurare gli endpoint.

- Rispondere ai segnali di controllo inviati all'endpoint0.
- Monitorare l'attività della GPIF
- Gestire la memoria fisicamente esterna
- Mette a disposizione USARTs, counter-timers, interrupts e I/O pins.
- Non prende parte alle comunicazione in cui è richiesta velocità di trasmissione elevata (high-speed 480 megabit/second).

I contatori/timers implementati nel CYC68013 sono tre e sono interamente gestiti dall'8051 interno. Questi timer possono funzionare con il clock interno oppure come contatori di eventi dei pin t0, t1, t2 rispettivamente nei timer0, timer1, timer3 (funzionalità non presente nel 56 pin). Ogni timer è a 16 bit (registro tlx+thx) accessibile dal software come SFR.

Le versioni 100 e 128 pin dispongono anche di due *serial port. La* serialport0 opera in modo analogo a quella dell'8051 standard, come generatore di baud rate può utilizzare sia clk (diviso 4 o 12) sia timer1 o timer2, la serialport1 è identica alla precedente ma può utilizzare solo il timer2.

Ogni porta può essere gestita in modo autonomo ed implementare un sistema in full duplex.

III.8 Input output

Il modello a 56 pin prevede due sistemi di I/O:

- Un set di pin I/O programmabili porte A, B, D a 8 bit bidirezionali; ad ogni porta è associata una coppia di SFR (Special Function Register) per il controllo:

- OEx (con x=A, B, D) configura la direzione dei singoli bit
- IOx rappresenta o il valore che scriviamo sui pin di uscita oppure il valore letto sui pin di ingresso

- Un bus I^2C compatibile per l'interfacciamento con periferica compatibile e il caricamento da serial eprom del firmware o dei pid/pid/did.

Le versioni 100 e 128 pin aggiungono 2 usart (programmabili) per le interfacce seriali e due porte bidirezionali (porte C, E).

Molti pin di I/O sono multiplexati e possono essere configurati (attraverso il configuration register) come generali oppure per implementare altre funzioni (es. GPIF adress/data, FIFO data, usart, interrupt signal); al momento dell'accensione tutti i pin sono configurati di default come I/O generali in ingresso.

Per quanto riguarda invece la gestione della comunicazione I^2C sono utilizzati tre registri:

- i2dat registro dati
- i2cs registro stato (done, ack, berr, start, stop)
- i2ctl registro configurazione bus

Il CYC68013 è sempre il bus master e genera una frequenza di bus variabile tra 100 e 400kHz.

CAPITOLO IV

Architettura e componenti del sistema

Il progetto di una telecamera stereo risulta essere piuttosto complesso, è perciò conveniente dividerlo in macro blocchi che andranno esaminati singolarmente.

L' elemento fondamentale del sistema è il sensore per l'acquisizione delle immagini (MT9V032). Le diverse modalità di funzionamento di questo sensore ne fanno un dispositivo particolarmente efficiente e versatile; può infatti essere utilizzato sia singolarmente nelle telecamere monoculari, sia in coppia con un'altro sensore per implementare una telecamera stereo.

Per collegare fisicamente il sistema stereo all'Host utilizzeremo il controller USB 2.0 (FX2) le cui caratteristiche sono già state prese in esame nel *CAPITOLO 3*.

Il compito del controller FX2 consiste solo nell'inviare, attraverso l'USB, i dati in arrivo dai due sensori, verrà perciò utilizzato in modalità auto in/out auto.

La scelta di non utilizzare l'8051 interno all' FX2 porta a una semplificazione del già complesso firmware, ma rende necessario l'inserimento di un'altro microcontrollore (PIC18F4690) che ricoprirà il ruolo di master dell'intero sistema.

I due sensori di immagine, nel complesso, trasferiscono attraverso due linee seriali differenziali (LVDS); per convertire i dati da seriali a paralleli è stato inserito un deserializzatore (DS92LV16).

Di seguito è riportato lo schema a blocchi complessivo della telecamera stereo progettata al DEIS nell'ambito della quale è stata svolta la mia tesi.

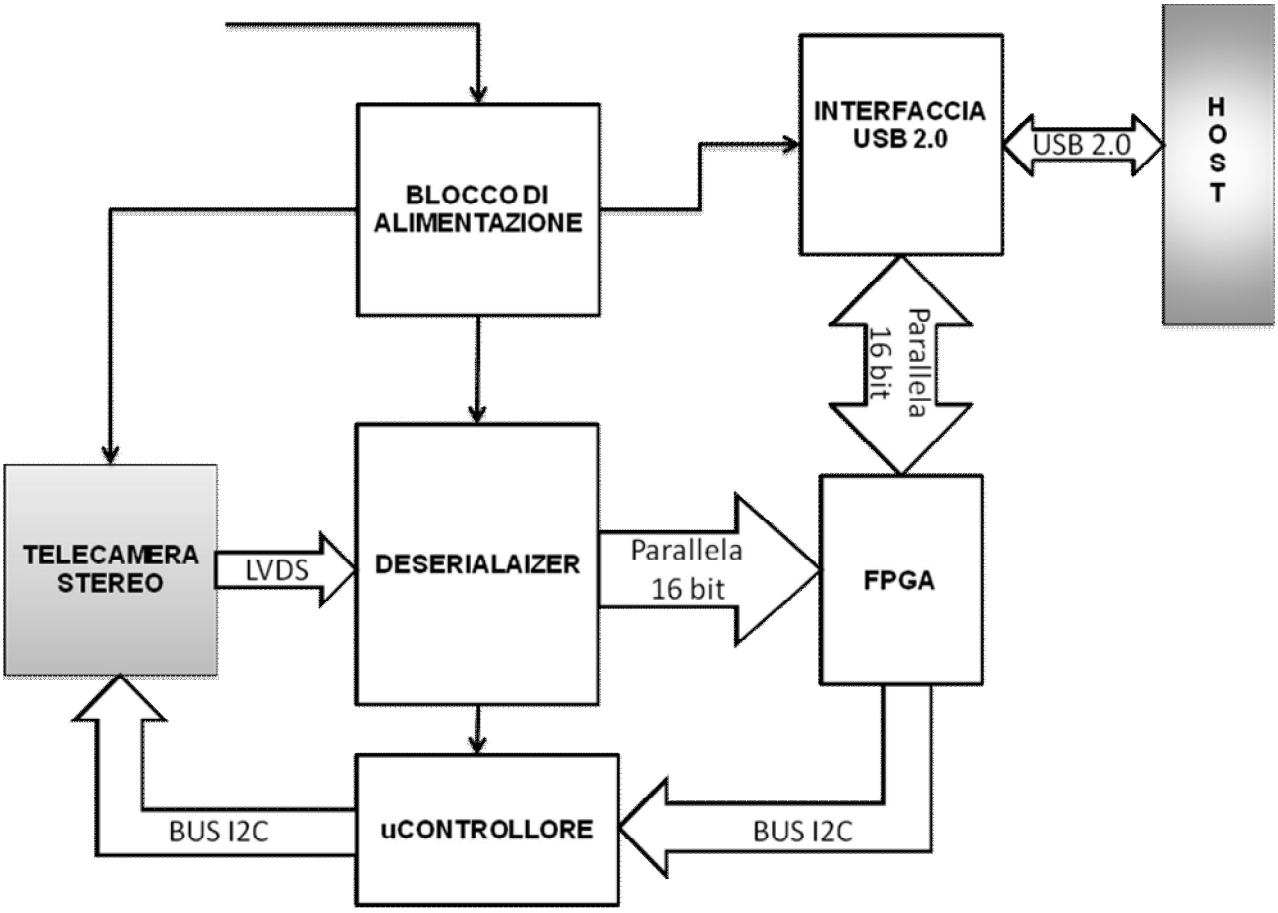

Figura 4.1 Architettura logica di sistema

IV.1 Sensori di immagine

Le diverse modalità di funzionamento del sensore MT9V032 [11] ne fanno un dispositivo efficace e alquanto versatile; può infatti essere utilizzato sia nelle telecamere tradizionali monoculari sia nelle telecamere che sfruttano la tecnologia stereo, è proprio in quest'ultimo impiego che focalizzeremo la nostra attenzione.

Non ci soffermeremo alla descrizione dettagliata del funzionamento del sensore di immagine, ma si concentrerà l'attenzione sul funzionamento del sistema stereoscopico e come esso dialoga verso l'esterno. Come già accennato in

precedenza, il sistema per l'acquisizione di immagini stereo è composto da due sensori aventi differenti funzionalità.

Il sensore secondario funzionante in stereoscopic slave ha il compito di trasmettere all'altro sensore, attraverso due linee seriali differenziali, le informazioni di uno specifico pixel letto e alcuni bit di handshaking.

Il secondo sensore funzionante in modalità stereoscopic master ha il compito di gestire la sincronizzazione con lo slave sensor e di trasmettere in uscita, attraverso due linee seriali differenziali, le informazioni ricevute allegando anche i propri dati letti (non ci è dato sapere come avviene l'operazione di sincronizzazione tra i due sensori). I due sensori di immagine sono quindi in grado di implementare un sistema stereo perfettamente sincronizzato senza nessun intervento dall'esterno a patto che siano configurati con gli stessi parametri (tempo di esposizione, risoluzione, ampiezza della finestra, tipo di scansione, frame rate ecc..).

Cominciamo con l'esaminare meglio il singolo sensore di immagine, questo è lo schema a blocchi.

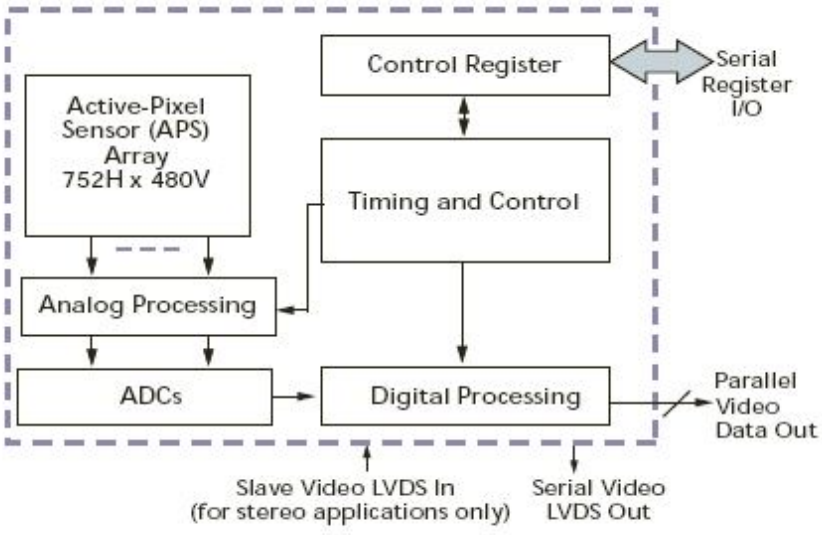

Figura 4.2 Schema a blocchi del sensore MT9V032

La matrice di pixel ha una risoluzione massima di 640*480, l'informazione in tensione proveniente da ogni elemento fotosensibile viene convertita in formato digitale a 8/10 bit e processata al fine di essere trasmessa verso l'esterno nel formato più adeguato.

- In modalità STAND ALONE, in cui viene utilizzato un singolo sensore, possiamo scegliere tre diverse tipi di funzionamento:
  - Simultaneous mode: In questa modalità il readout e l'esposizione avvengono in parallelo, rendendo l'operazione più veloce, ma più difficile da sincronizzare.
  - Sequential mode: A contrario della modalità precedente, qui l'esposizione avviene successivamente al readout.
  - Snapshot mode: In questa modalità il sensore accetta un segnale di trigger esterno che fa cominciare immediatamente l'esposizione e il successivo readout dell'immagine catturata.

Lo stream di uscita può esser in formato parallelo a 8 bit oppure seriale a 12 bit per pixel. Nel formato seriale possiamo selezionare due diverse modalità di impacchettamento: la prima prevede 10 bit di dato più lo start e lo stop bit, mentre la seconda prevede 8 bit di dati, lo start bit , lo stop bit e sono presenti inoltre line valid e frame valid.

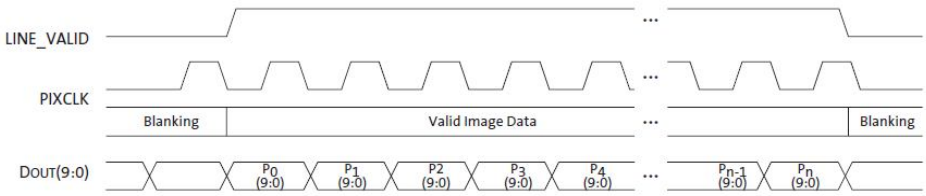

Figura 4.3 Modalità di funzionamento seriale

- In modalità SERIAL STEREOSCOPIC i dati possono essere inviati solo per via seriale in pacchetti da 18 bit contenenti: il bit di start, 8 bit di dati su uno specifico pixel provenienti dallo slave sensor, 8 bit di dati riguardanti lo stesso pixel provenienti dal master sensor ed infine il bit di stop. I segnali di frame e line valid sono ricavati con l'utilizzo di parole riservate.

  I bit vengono trasmessi in uscita ad una frequenza di 520 Mhz supponendo che il tempo di esposizione della matrice di pixel sia il minimo possibile e che i sensori funzionino alla massima risoluzione.

  I sensori dovranno essere configurati uno come SLAVE STEREOSCOPIC, e l'altro come MASTER STEREOSCOPIC, come già detto in precedenza.

  Le linee seriali di ingresso del master sensor SER_DAT_IN andranno collegate con le linee seriali di uscita dello slave sensor per permettere la comunicazione tra i due sensori.

Per quanto riguarda la programmazione dei sensori dobbiamo settare il valore di 256 registri (da 16 bit) per ciascun dispositivo, ricordando anche che alcuni registri vanno programmati in simultanea (broadcast) per mantenere una corretta temporizzazione.

Da notare inoltre che la programmazione avviene attraverso un bus I^2C; il nostro controller USB implementa questo tipo di I/O ma, per mantenere il software di controllo più semplice possibile, si è deciso di non utilizzarlo per la programmazione ed inserire un altro processore esterno dedito ad eseguire questo lavoro.

La parte bassa del select code del bus I^2C di ogni sensore è assegnata attraverso i due pin CTRL_ADR0 e CTRL_ADR1, come raffigurato nella seguente tabella.

{S_CTRL_ADR1, S_CTRL_ADR0}	Slave Address	Write/Read Mode
00	0x90	Write
	0x91	Read
01	0x98	Write
	0x99	Read
10	0xB0	Write
	0xB1	Read
11	0xB8	Write
	0xB9	Read

Figura 4.4 Tabella degli indirizzi del sensore MT9V032

Per ridurre il frame rate in uscita dai sensori è possibile aggiungere dei pixel *dummy* a lato ed in fondo alla finestra di immagine, queste righe e colonne vengono chiamate rispettivamente horizontal e vertical blanking.

L'ammontare del banking orizzontale e verticale è configurabile tramite due registri R0x05, R0x05. Il segnale IMAGE VALID, quando basso, avverte che in uscita si sta trasmettendo un immagine non valida, cioè il banking.

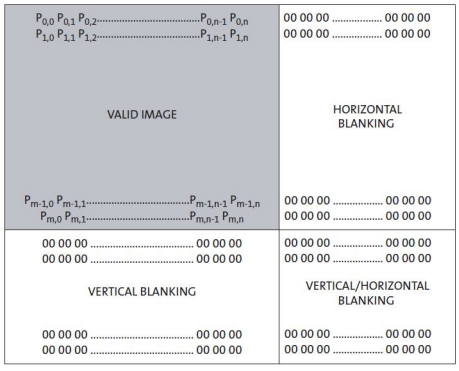

Figura 4.5 Schema a blocchi del sensore MT9V032

IV.1.1 Schema di collegamento

I due sensori sono connessi in modalità stereoscopica: le linee seriali di uscita dello slave sono connesse a quelle di entrata del master, mentre i pin di uscita del sensore master vanno direttamente al deserializer tramite due cavi SMA.

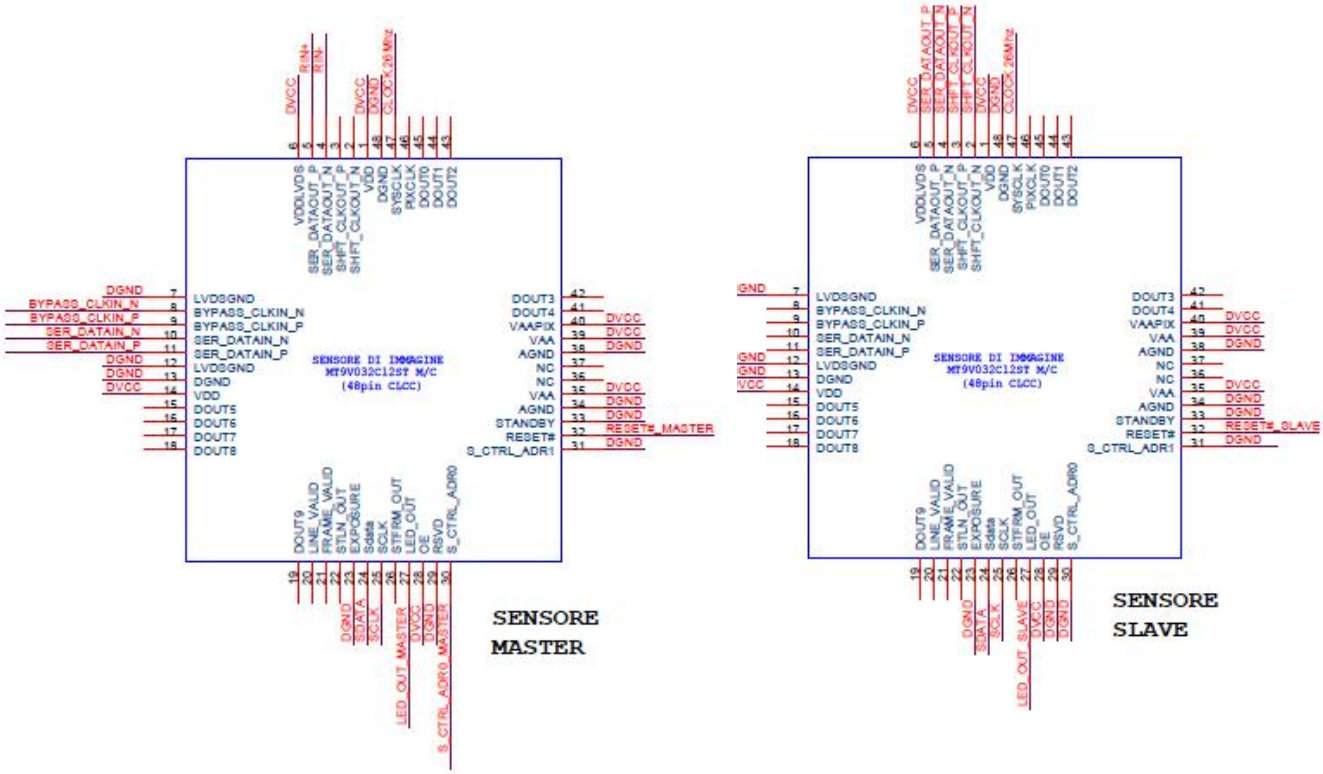

Figura 4.6 Schema di collegamento dei due sensori

IV.2 Deserializer

Questo dispositivo combina un convertitore parallelo/seriale (serializer) ed un convertitore seriale/parallelo (deserializer) in un singolo chip [10].

I due blocchi vengono mantenuti separati l'uno dall'altro sia per l'alimentazione, sia per quanto riguarda la generazione del clock. I due convertitori possono quindi lavorare anche in simultanea per avere una connessione in full duplex mode.

Di nostro interesse risulta solo il blocco deserializer, per trasformare dati seriali,

provenienti dai sensori, in dati paralleli a 16 bit da inviare al controller USB.

IV.2.1 Descrizione delle funzionalità

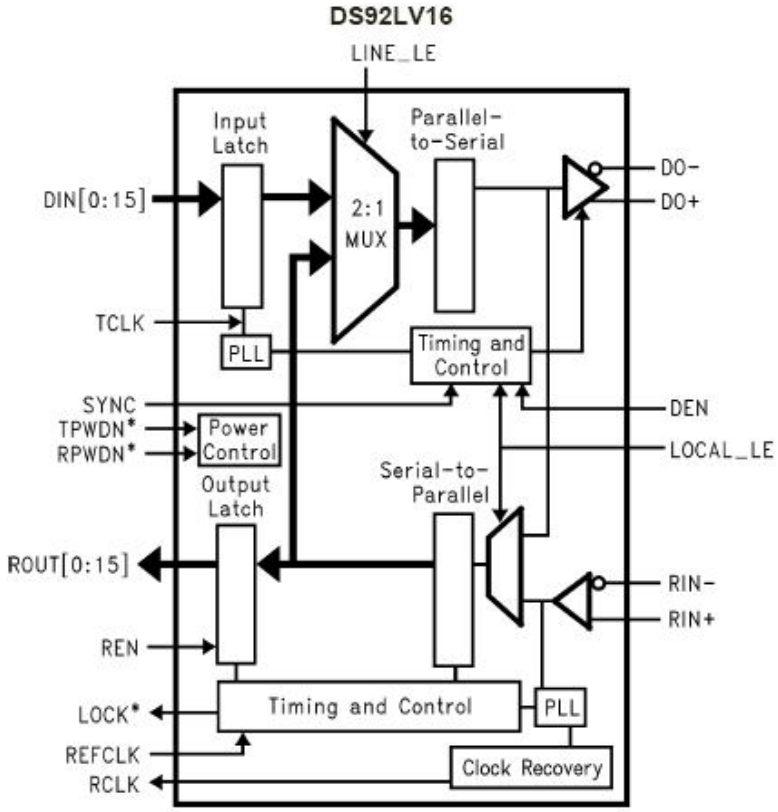

Figura 4.7 Schema a blocchi del DS92LV16

Verranno prese ora in esame le caratteristiche e le funzionalità presenti nel blocco deserializer:

Inizialization: Quando viene fornita l'alimentazione tutte le uscite sono in tristate. Appena la tensione supera i 2,2V la PLL interna cerca un clock locale (fornito sul pin REFCLK) a cui agganciarsi; le uscite rimangono ancora in trisate e il pin LOCK rimane alto.

Per completare la sincronizzazione la PLL del blocco deserializer deve sincronizzarsi con i dati ricevuti sui pin: RIN+ e RIN-. Per agevolare questa fase

può essere inviato un pattern di sincronizzazione di bit non ripetitivi, ma è altresì possibile sincronizzare la PLL con dati random, in tal caso però non è stimabile il tempo necessario per l'aggancio.

Lo stato logico basso del pin LOCK segnala la corretta sincronizzazione e la validità dei dati in uscita sulle linee parallele.

Data transfer: Il circuito interno di clock recover viene utilizzato per generare un segnale di strobe. Il fronte di salita di questa linea segnala la campionabilità dei dati presenti nella porta parallela.

Resynchronization: Quando si perde l'aggancio (il pin LOCK va alto) il deserializer cercherà automaticamente di risincronizzarsi in modo random estraendo informazioni dai dati in arrivo.

Powerdown: E' uno stato a bassa potenza in cui si può entrare abbassando TPWDN, RPWDN (notare che il serializer e il deserializer possono essere spenti autonomamente).

In questo stato si spegne la PLL (si perde l'aggancio) e il dispositivo si mette quindi in attesa per una nuova inizializzazione.

Tristate: Quando il pin REN viene portato basso le uscite del deserializer sono poste in alta impedenza.

Loopback Test Operation: E' possibile creare una sorta di feedback collegando RIN+, RIN- con DO+, DO- e ROUT[0:15] con DIN[0:15] in questo modo tutti i dati ricevuti dal dispositivo vengono rinvianti al mittente, che potrà quindi controllare se siano stati ricevuti dal deserializzatore correttamente.

In figura 4.3 vengono mostrate le forme d'onda dei segnali coinvolti nella deserializzazione in fase di aggancio.

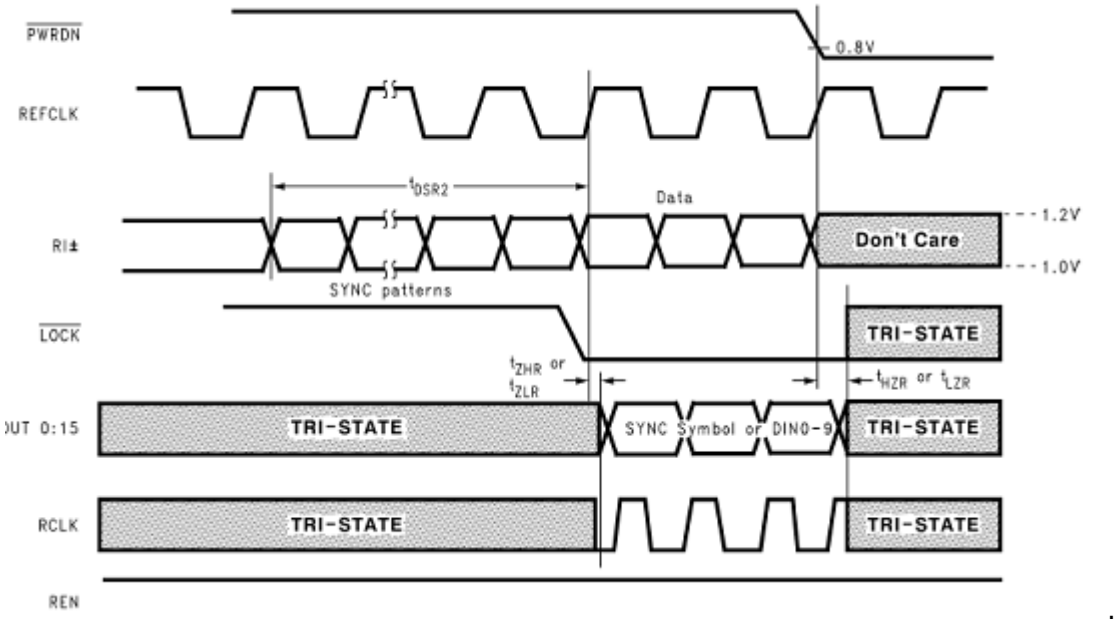

Figura 4.8 Forme d'onda dei segnali coinvolti nella deserializzazione

Da notare che il tempo (indicato in figura da tdsr2) necessario alla sincronizzazione è molto breve (circa un micro secondo) nel caso venga inviato un sync pattern; ha invece durata non predicibile se si utilizza la funzione random lock.

IV.2.2 Schema di collegamento

Di seguito viene descritto come il deserilizer sarà inserito nel progetto della telecamera:

I datti in uscita dai due sensori stereo vengono mandati attraverso due connettori SMA al deserializer (RIN+, RIN-).

Il pin di LOCK viene mandato all' FPGA, che potrà quindi vedere gli istanti in cui la PLL non è agganciata.

Il segnale in grado di spegnere il deserializer (RPWDN#) è controllato dal master del sistema (il ucontrollore).

I dati paralleli in uscita (Rout [0:15]) vengono mandati assieme al pin di strobe (RCLCK) all'FPGA che provvederà a reindirizzarli al controller USB.

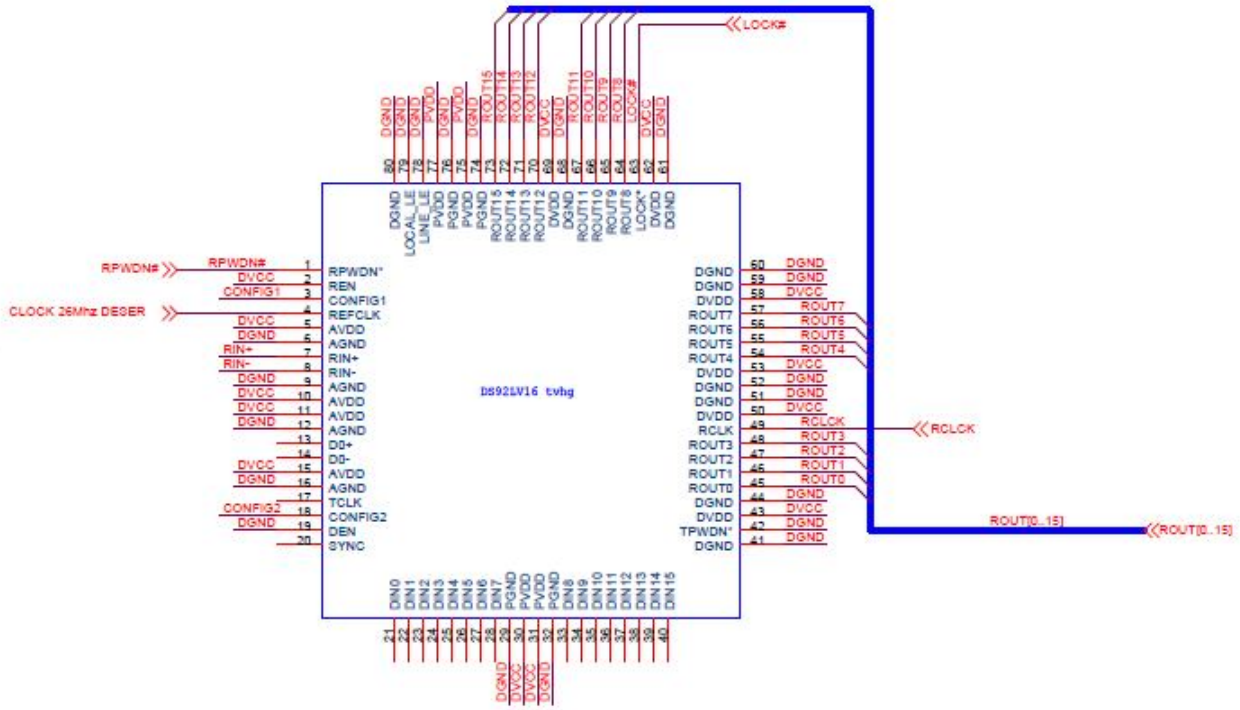

Figura 4.9 Schema di collegamento deserializer

IV.3 Interfaccia USB 2.0

Il controller USB (FX2) verrà utilizzato in modalità slave FIFO sincrono a 16 bit con modalità auto in, auto out attiva.

L'FPGA provvederà a fornire sia il clock esterno (sul pin IFCLK) sia il segnale di scrittura dei dati (SLWR). All'interno dello schematico sono stati inseriti anche dei jumper per decidere manualmente se eseguire il pull up o pull down dei seguenti segnali SLRD, FIFOADR0, FIFOADR1, PKTEND, CLCS#, WAKEUP#, SLOE.

Il pin di reset hardware è connesso, oltre che a una rete RC (per eseguire il power on reset), anche a un pulsante.

E' stato collegato un oscillatore al quarzo su XTALIN, XTALOUT come da datasheet.

Sono state inserite le due resistenze di pull up con resistori da 3.3 KOhm sulle linee SDA, SCL come prevede il bus I^2C.

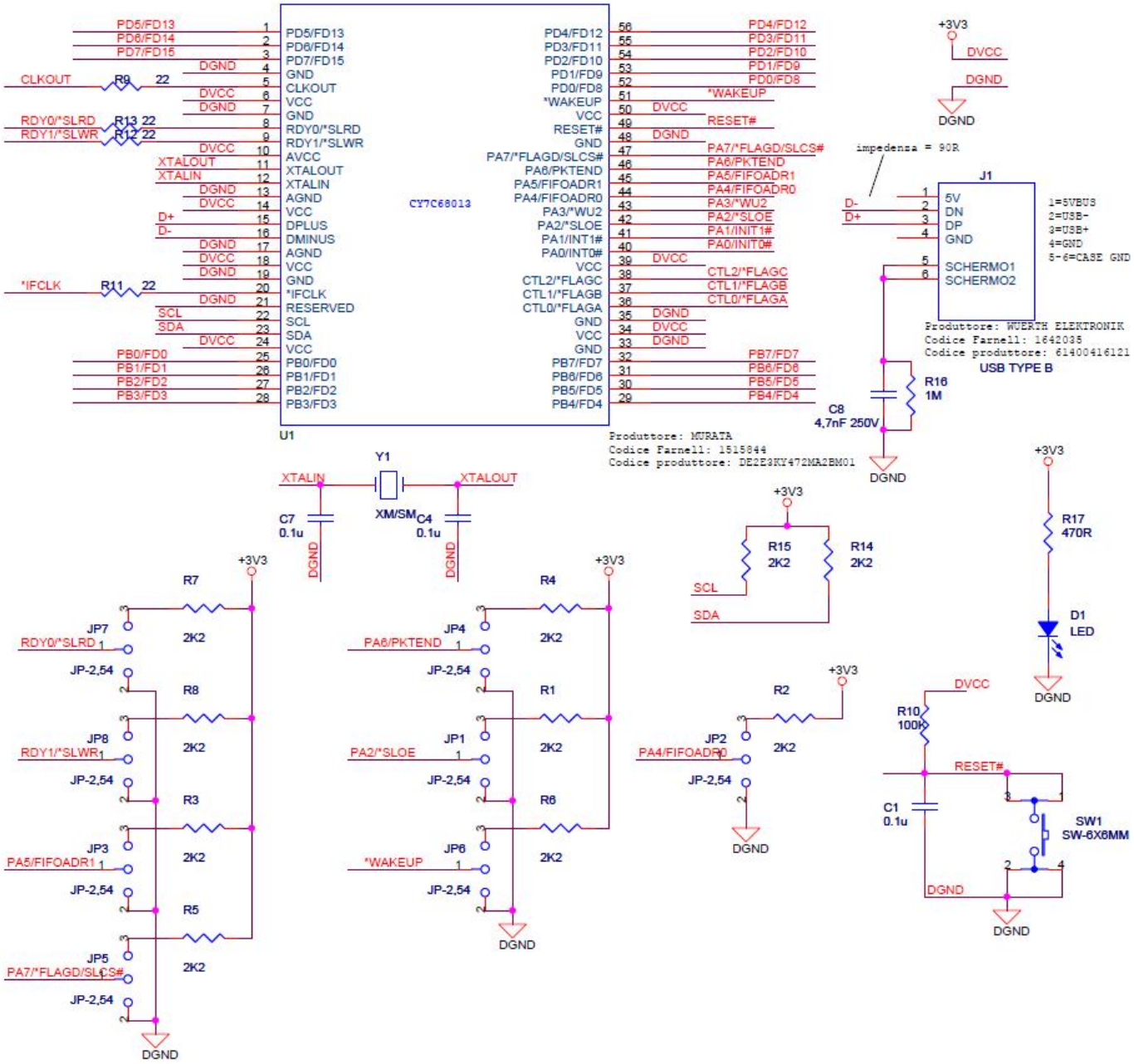

Figura 4.10 Schema di collegamento FX2

IV.4 Circuito di generazione del clock

La generazione del clock dell'intero sistema è affidata a un apposito blocco, il cui unico scopo è fornire il clock di riferimento ai sensori e al deserializer.

Il chip che è stato utilizzato è il CY22393 prodotto dalla Cypress [9], questo componente implementa:

- Tre PLL autonome
- Una serie di divisori/moltiplicatori programmabili via I^2C
- Un sistema di suspend mode

IV.4.1 Schema di collegamento

Le uscite A,B,C delle tre PLL vengono rispettivamente connesse al deserializer, ai sensori e a un clock ausiliario (momentaneamente inutilizzato).

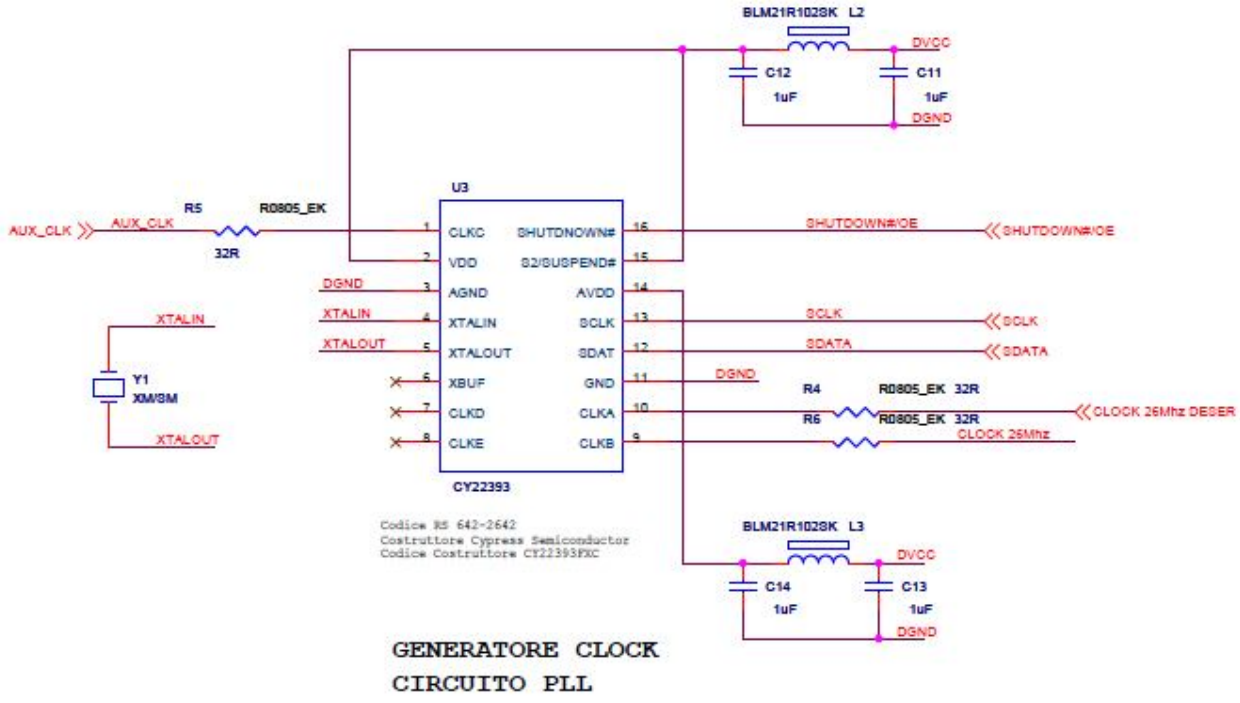

Figura 4.11 Schema di collegamento PLL

IV.5 Microcontrollore

Come già chiarito in precedenza, per mantenere il software di controllo del controller USB il più semplice possibile, non viene utilizzato il processore 8051 messo a disposizione dalla Cypress [12]. Si è reso quindi necessario l'inserimento di un processore esterno dedito al controllo dei reset e all'inizializzazione dei due sensori di immagine (ricordiamo che la programmazione di tali dispositivi avviene tramite un bus I^2C [*Appendice C*]). Un valido candidato a svolgere questi compiti è il PIC18F2585 prodotto dalla Microchip.

Questo microcontrollore ha prestazioni modeste, compatibilmente con le mansioni che dovrà svolgere, ed è disponibile sul mercato a basso costo.

Di seguito vengono brevemente elencate le sue principali caratteristiche:

- Il parallelismo della ALU (Aritmetical Logical Unit) è 8 bit
- Diverse versioni 28/40/44 package (noi utilizzeremo la 44)
- 64Kbyte di Flash, 3Kbyte di SRAM, 1Kbyte di EEPROM
- Programmabile e testabile via JTAG
- 32 linee di Input Output (I/O) multiplexate
- Watch Dog Timer (WDT)
- Modulo Pulse width Modulation (PWM)
- 4 Phase Lock Loop (PLL)
- 3 Interrupts esterni con priorità programmabile
- E' munito di interfacce: I^2C, SPI e RS232
- 3 timer/counter a 8/16 bit programmabili

- ADC a 10 bit con 11 canali con velocità fino a 100 Ksps
- 2 comparatori analogici
- Modalità di stop e Idle

IV.5.1 Schema di collegamento

Il Pic mantiene in memoria flash i valori dei registri dei sensori e delle PLL che dovranno essere configurati e li programma via I^2C [*Appendice C*].

E' in grado di cambiare dinamicamente l'indirizzo I^2C del sensore master (attraverso il pin S_CTRL_ADR0_MASTER) per eseguire una programmazione broadcast dei registri comuni.

Ha inoltre il compito di controllare gli stati di suspend dell'Fx2, del deserializer e delle PLL.

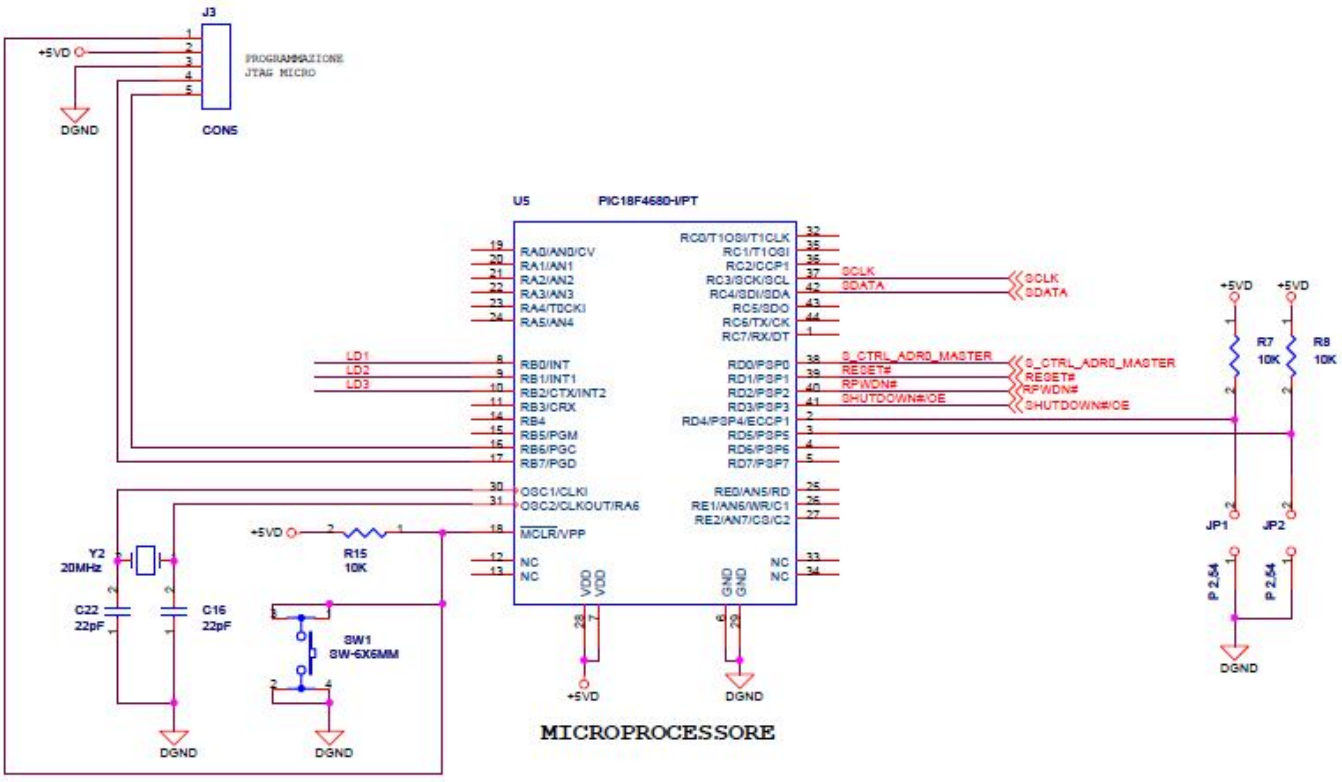

Figura 4.12 Schema di collegamento PIC

CAPITOLO V

Dispositivi utilizzati

Nel corso della Tesi sono stati utilizzati diversi dispositivi, il presente capitolo ha lo scopo di descrivere sommariamente le caratteristiche di ciascuno di essi.

V.1 Development board Spartan 3

La sigla FPGA rappresenta l'acronimo di Field Programmable Gate Array. L'architettura di una FPGA e' rappresentata da un insieme di celle logiche che comunicano tra loro e con i segnali di ingresso-uscita mediante un insieme di collegamenti programmabili disposti orizzontalmente e verticalmente. Le funzioni logiche sono ottenute combinando diverse celle logiche in differenti modi.

La struttura delle celle logiche e lo sviluppo dell'insieme dei collegamenti differiscono da costruttore a costruttore.

Attualmente le tecnologie più utilizzate con cui realizzare FPGA sono : SRAM (Static RAM) e ANTIFUSE. Ognuna di queste permette di realizzare FPGA che soddisfano a particolari esigenze di mercato.

L'architettura di una cella logica e' fortemente influenzata dalle caratteristiche delle interconnessioni. Le FPGA che hanno una struttura di interconnessioni caratterizzata da molti fili e molti elementi di connessione programmabili tendono ad avere celle logiche semplici con un minor numero di ingressi; in questi casi viene perciò utilizzata la tecnologia antifuse. Invece FPGA che presentano strutture di

interconnessione con un minor numero di fili e di interconnessioni programmabili tendono ad avere celle logiche più complesse con un maggior numero di ingressi. In quest'ultimo caso si utilizza tipicamente la tecnologia a SRAM.

L' FPGA della serie Spartan 3 utilizza blocchi di interconnessione programmabili di tipo SRAM con blocchi logici complessi (Look Up Table).

La programmazione di tale dispositivo non è quindi permanente, ma va effettuata ogniqualvolta viene fornita alimentazione.

L'azienda costruttrice Xilinx, al fine di agevolare al massimo il compito del progettista, mette a disposizione una Development board che sfrutta al meglio le caratteristiche della spartan3 [5,6]. Questa complessa scheda contiene già al suo interno:

- Una FPGA Spartan 3 dotata delle seguenti caratteristiche:

  in un package 320 solder ball grid array(BGA)
  10.000 gate logici equivalenti
  18K byte di block ram
  18 moltiplicatori hardware a 18 bit
  4 Clock managers (DCM)
  232 user-defined I/O signals
- 4 Mbit di memoria PROM configurabile mediante jumpers
- 64 MByte di memoria sincrona DDR SDRAM bus a 16 bit
- 16 Mbyte di NOR flash
- 3 bit, 8 color VGA display port
- 9 pin, RS232 serial port
- PS2 port per mouse e tastiera
- 10/100 Ethernet inteface
- 4 DAC e 2 ADC

- Encoder rotativo
- Display a lcd con due righe
- 4 interruttori a slitta, 4 pulsanti
- 8 led indirizzabili singolarmente
- Un oscillatore a 50 MHZ ed un socie libero per un oscillatore ausiliario
- Porta JTAG
- Un controller USB 2.0
- Sulla scheda sono presenti diversi linearizzatori per fornire tensioni a: 5V, 2.5V e 1.2V
- Un connettore HIROSE 100 pin verso l'esterno

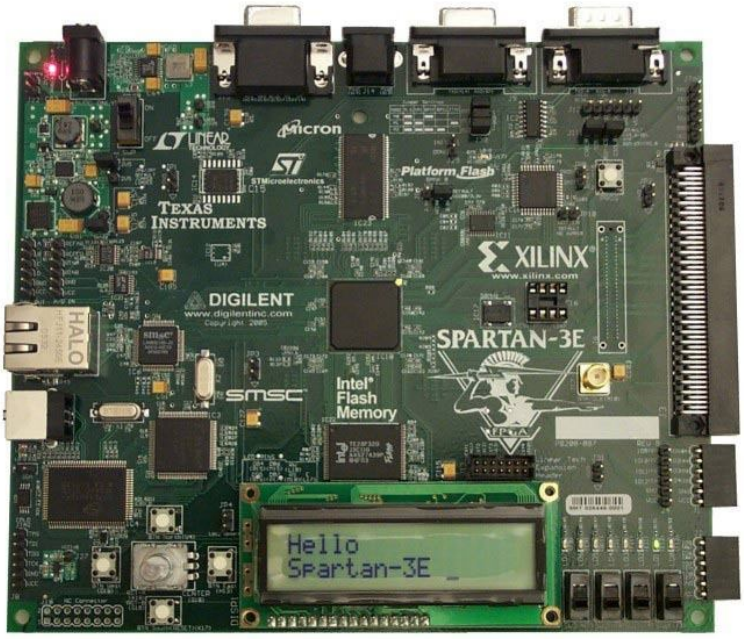

Figura 5.1 Development board Spartan3

Viene inoltre fornito un IDE (Integrated development environment) comprensivo di un tool software (Xilinx Design Suite) che consente la programmazione dell'FPGA via USB. In questo modo è possibile programmare sia la memoria volatile (RAM), sia la memoria persistente (Flash).

V.1.1 Il codice VHDL

Il VHDL (Hardware Description Language) è un linguaggio di descrizione dell'hardware che consente la progettazione di circuiti integrati.

È un derivato del programma Very High Speed Integrated Circuit (VHSIC) ideato dal dipartimento della difesa Statunitense tra la fine degli anni 70 e l'inizio degli anni 80. Gli obbiettivi di questo linguaggio sono duplici: stabilire uno standard di interfacciamento tra i vari progettisti ed avere un linguaggio le cui istruzioni siano orientate alla rappresentazione circuitale.

L'insieme delle istruzioni che descrivono il progetto hanno lo scopo di modellare la realtà a cui il progettista fa riferimento.

Per ritornare ai gate reali è necessario un passaggio intermedio che ha lo scopo di tradurre le istruzioni in logica combinatoria e logica sequenziale. Questo processo è chiamato sintesi ed normalmente affidato a dei tool di sintesi automatici.

L'utilizzo di tool automatici conferisce la possibilità di creare circuiti sempre più complessi in un tempo molto più contenuto rispetto alla progettazione gate level.

Il rischio che si corre è quello di implementare una funzionalità logica su un area molto maggiore del necessario con prestazioni di timing deludenti o addirittura implementare delle funzionalità che non sono effettivamente sintetizzabili.

Una volta descritto il circuito in un linguaggio con una struttura formale si può procedere alla verifica della correttezza del progetto, tale fase viene chiamata simulazione e può essere solo in termini logici o può includere anche le tempistiche.

Dopo questa piccola introduzione sul diffuso linguaggio VHDL verrà ora presa in esame la struttura di una parte del codice che è stato utilizzato inizialmente per la sperimentazione.

Il codice VHDL, in questa fase, ha lo scopo di simulare il comportamento dei sensori mascherando la loro assenza al controller USB. L'FPGA dovrà controllare il clock (IFCLK), il comando di scrittura (SLWR), verificare lo stato dell'endpoint campionando il flag full dell'EP6, inoltre invierà al controller USB un pattern incrementale noto fermandosi non appena l'endpoint si riempie.

Questa gestione dei dati inviati, in verità, non rispecchia a pieno il caso in cui siano presenti effettivamente i sensori.

I dati provenienti dai sensori infatti, non possono essere bloccati se l'endpoint è pieno ma dovranno essere memorizzati in una memoria.

Questo test risulta comunque utile perché fornisce indicazioni sulle prestazioni del controller USB in un caso ideale.

L'invio di un pattern incrementale noto permette all'Host PC di sapere se i dati ricevuti sono corretti oppure, in caso contrario, di conoscere quanti byte sono andati persi nella trasmissione.

Il codice risulta essere diviso in tre parti:

- L'UCF è un file di testo che ha lo scopo di legare i segnali di uscita con i pin fisici di uscita dell'FPGA, un esempio di dichiarazione è il seguente:

  ```
  NET "slw"       LOC = "D10" | IOSTANDARD = LVCMOS33 | SLEW = slow | DRIVE = 2 | KEEPER = TRUE;
  ```

- Il Digital Clock Manager (DCM) è un generatore digitale di clock programmabile la cui struttura interna viene fornita da Xilinx. Questo componente ha diverse

funzionalità ma in questo progetto è stato semplicemente utilizzato come generatore di onda quadra a frequenza variabile.

- Il programma principale contiene le istanze di tutti i componenti e la definizione delle varie architetture.

  La Top entità è denominata TestClk ed è definita in questo modo:

```
entity TestClk is
   Port ( clk_50MHz_in : in  STD_LOGIC;
                  flag : in  STD_LOGIC;
                  clk_out  : out STD_LOGIC := '0';
                  data_out : out STD_LOGIC_VECTOR (7 downto 0):= (others => '0');
                  line_out : out STD_LOGIC_VECTOR (7 downto 0):= (others => '0');
                  slw  : out STD_LOGIC := '0';
                  enable_SLW  : in  STD_LOGIC;
                  enable_clk_out : in  STD_LOGIC);
end TestClk;
```

  La sua architettura è così composta:

  Vengono inizialmente stanziati due componenti per la generazione di un clock a 50MHZ e di uno a frequenza variabile (grazie all'ausilio del DCM)

```
CLKIN_IBUFG_INST : IBUFG port map (
                                          I=>clk_50Mhz_in,
                                          O=>signal_clk_50Mhz_in );

b2 : entity work.dcm_input(BEHAVIORAL) port map (
                                CLKIN_IN => signal_clk_50Mhz_in,
                                CLKFX_OUT => clk_50MHz);

clk_out <=  clk_50MHz;
```

  Per la generazione dei dati viene utilizzato un contatore incrementale che utilizza il clock a 50 MHz, mentre il clock in uscita (IFCLK) è collegato al DCM.

  Il processo per la generazione dei dati test da mandare all'FX2 blocca la scrittura (cioè alza il pin di SLW) e ferma il conteggio nel caso in cui si verifichi un evento di full, in questo modo l'Host deve necessariamente leggere un pattern incrementale ripetitivo, altrimenti si è incorsi in un errore.

```
data : process (clk_50MHz,flag,enable_SLW)
variable count_data : integer := 51;
variable count_data2 : integer := 51;
variable data_out_T : std_logic_vector(7 downto 0) := (others => '0');
variable data_out_T2 : std_logic_vector(7 downto 0) := (others => '0');
begin
if(flag = '1' and enable_SLW='1') then
                    slw <= '0';
                    if (clk_50MHz = '0' and clk_50MHz'event )then
                              data_out_T := data_out_T + 1;
                              data_out_T2 := data_out_T2 + 1;
                              if (count_data > 0) then
                                        count_data := count_data - 1;
                              end if;
                              if (count_data2 > 0) then
                                        count_data2 := count_data2 - 1;
                              end if;
                               if (count_data = 0) then
                                        data_out_T :=(others => '0');
                                        count_data :=51;
                              end if;
                              if (count_data2 = 0) then
                                        data_out_T2 :=(others => '0');
                                        count_data2 :=51;
                              end if;

                              line_out <= data_out_T2;
                              data_out <=  data_out_T;

                    end if;
                     else
                              slw <= '1';
                     end if;
end process data;
```

V.2 Development board Brain technology

L'azienda costruttrice Brain technology produce una development board basata sul controller CY7C68013 [13].

Questa board contiene al suo interno tutto quello che è necessario per fare funzionare il controller USB:

- L'alimentazione viene prelevata dal bus USB (a 5V) e viene linearizzata ad un valore di 3,3V
- E' presente un oscillatore a 24 MHZ

- Una memoria EEPROM da 8 Kbyte I^2C
- Viene portato verso l'esterno tutto l'I/O del chip CY7C68013 con due strip da 30 pin ciascuna.

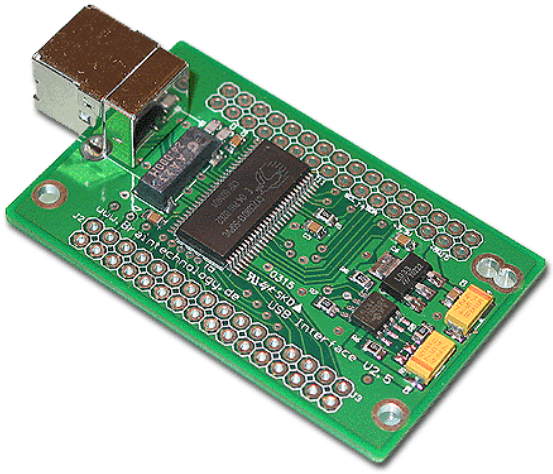

Figura 5.2 Development board Brain technology

In una prima fase della Tesi è stato sviluppato un prototipo filare su board mille fori per interfacciare la scheda development board della spartan3 con la scheda Brain technology.

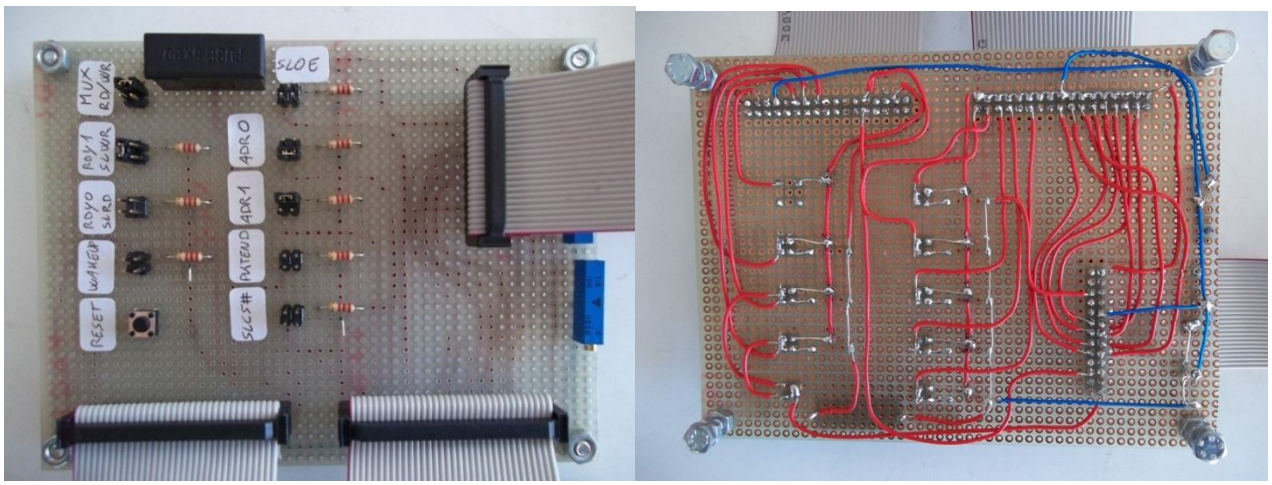

Figura 5.3 Scheda mille fori

Poiché nel connettore Hirose della spartan3 viene portato solo una piccola parte dell'I/O complessivo della FPGA è stato necessario inserire dei jumper in grado di eseguire il pull up o il pull down di alcuni pin a seconda dell'esigenza.

Sono presenti inoltre un fusibile da 0,5 A e un insieme di due trimmer e un diodo shotty per eseguire un adattamento di impedenza sulla linea di clock IFCLK.

La connessione tra la spartan 3 e la scheda forata, all'atto pratico, è risultata problematica poiché la scheda stampata che converte il connettore hirose in due connettori flat utilizza strip di passo più piccolo rispetto alle strip montate sulla mille fori. Si è quindi reso necessario un collegamento pin a pin con alcuni fili volanti come mostrato in figura 5.2. Questo ha portato non pochi problemi, sia in termini di robustezza meccanica, sia in veri e propri problemi elettrici i segnali risultavano infatti fortemente disturbati a causa dei vari salti di impedenza e delle lunghezze esagerate dei cavi di connessione.

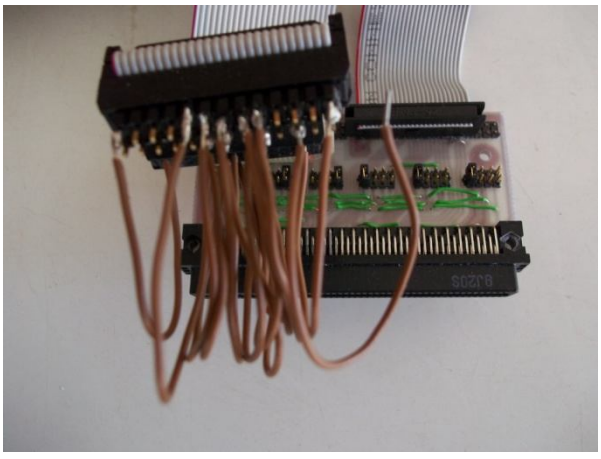

Figura 5.4 Collegamento tra mille fori e spartan 3

V.3 Lo stampato del prototipo

In una seconda fase di sperimentazione si è passati ad uno stampato del prototipo che inizialmente era stato implementato sulla scheda mille fori.

Questo prototipo stampato è stato suddiviso su due schede per mantenere distinte l'interfaccia USB 2.0 con la board di connessione verso la demo board dell'FPGA.

Questa modularità consente di cambiare il tipo di interfaccia di comunicazione senza dovere rifare completamente il layout.

Infatti è attualmente in studio da parte di altri studenti una interfaccia GIGABIT ETHERNET e si sta inoltre vagliando l'ipotesi di usare il recente protocollo USB 3.0 mediante un opportuno controller.

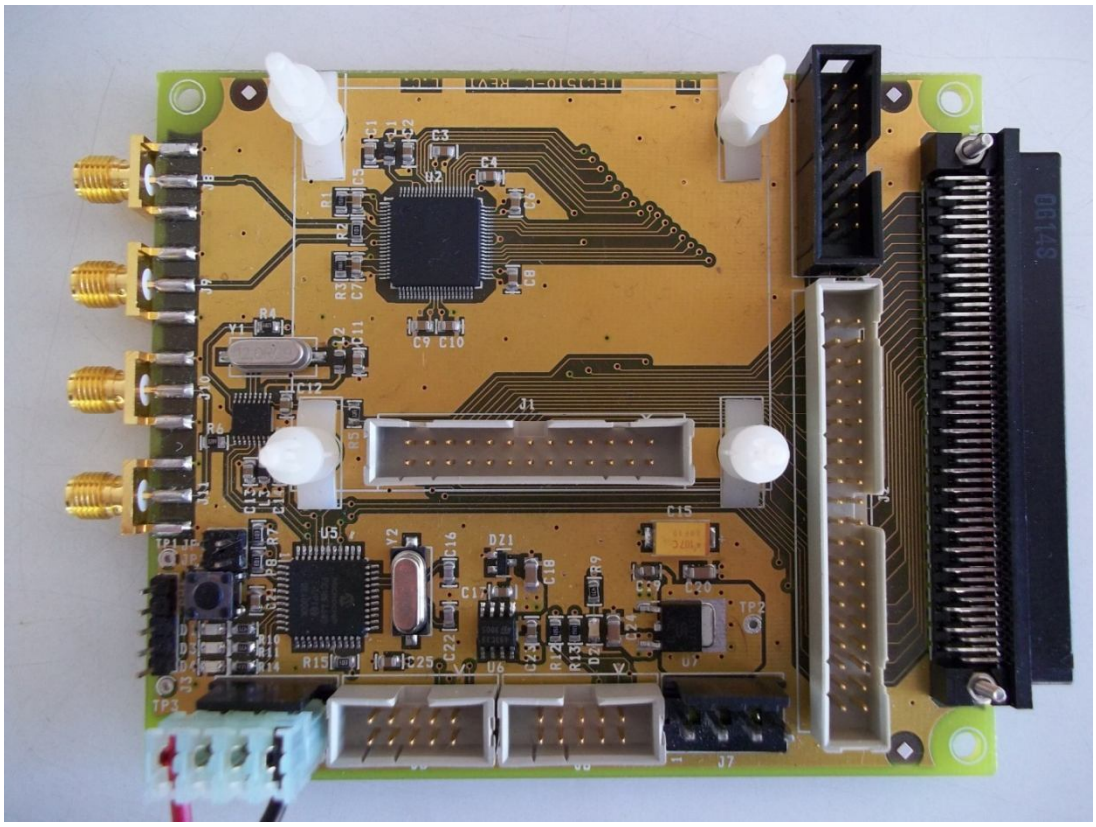

Figura 5.5 Stampato della board base

La prima scheda, mostrata in figura 5.2 contiene la logica di gestione dell'intero sistema. Al suo interno sono presenti: il deserializer, le PLL, il microcontrollore e un linearizzatore locale di tensione che trasforma la 8,8V nella 3,3 V necessaria per alimentare tutti i dispositivi.

Questa scheda, inoltre, si interfaccia sia con i sensori, attraverso due connettori SMA, sia con entrambe le FPGA spartan3 e spartan6.

Per connettersi alla Development board SPARTAN 3, le cui caratteristiche sono già state prese in esame nel paragrafo precedente, viene utilizzato il connettore HIROSE 100 pin.

La connessione con la demo board della SPARTAN 6 avviene invece attraverso due connettori FLAT da 18 e 40 pin.

Si è scelto di implementare le interfacce per entrambe le spartan poiché il progetto è stato prima realizzato su Spartan3 (le cui caratteristiche erano già note in precedenza) e solo successivamente sarà portato sulla più moderna e performante Spartan6.

La board di comunicazione, riportata in figura 5.3, contiene invece il controller USB 2.0. Preso atto che l'I/O messo a disposizione sui connettori esterni delle due demo board spartan non è sufficiente al totale controllo dell'FX2, vengono inseriti (come nel caso della mille fori) dei jumper per eseguire il pull'up o pull down delle linee che non necessitano il comando diretto dell'FPGA.

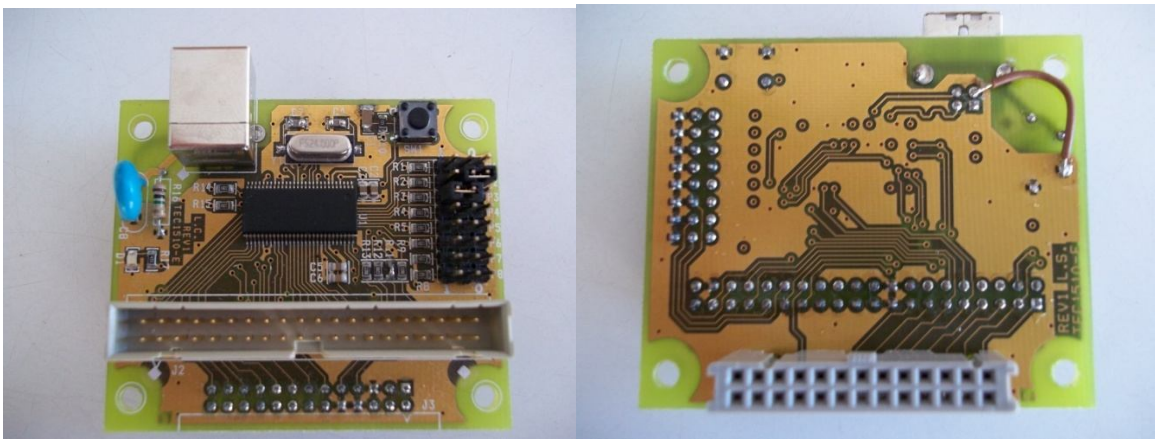

Figura 5.6 Stampato del prototipo

Le due schede sono connesse in pila grazie a quattro sostegni in plastica che garantiscono la solidità meccanica del tutto, come mostrato in figura 5.3.

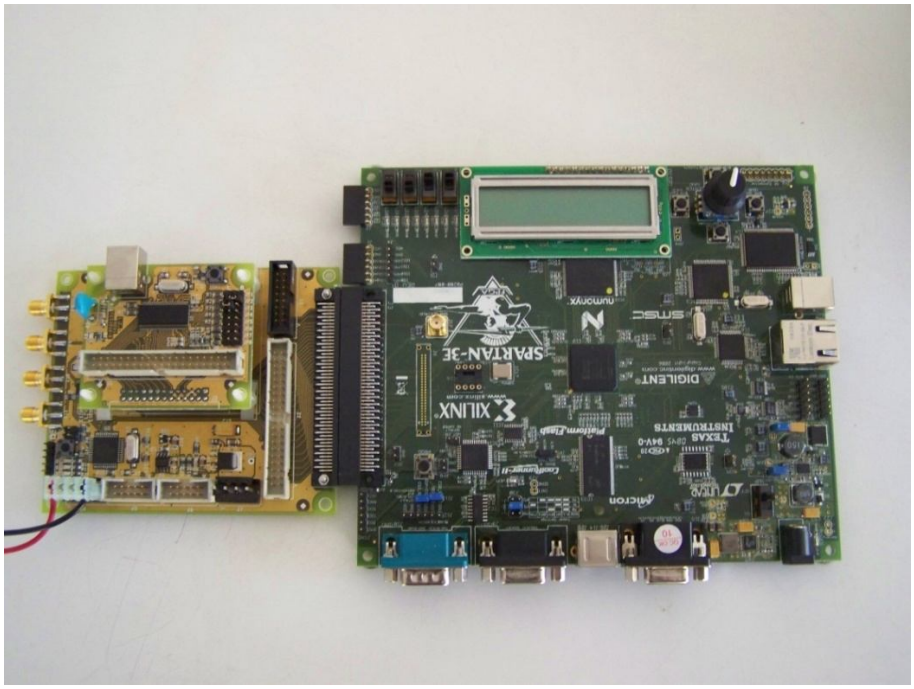

Figura 5.7 Il prototipo nel complesso

L'alimentazione viene portata al prototipo da un'altra scheda il cui unico compito è linearizzare la tensione proveniente da un trasformatore a muro e portarla ad un valore di 8V.

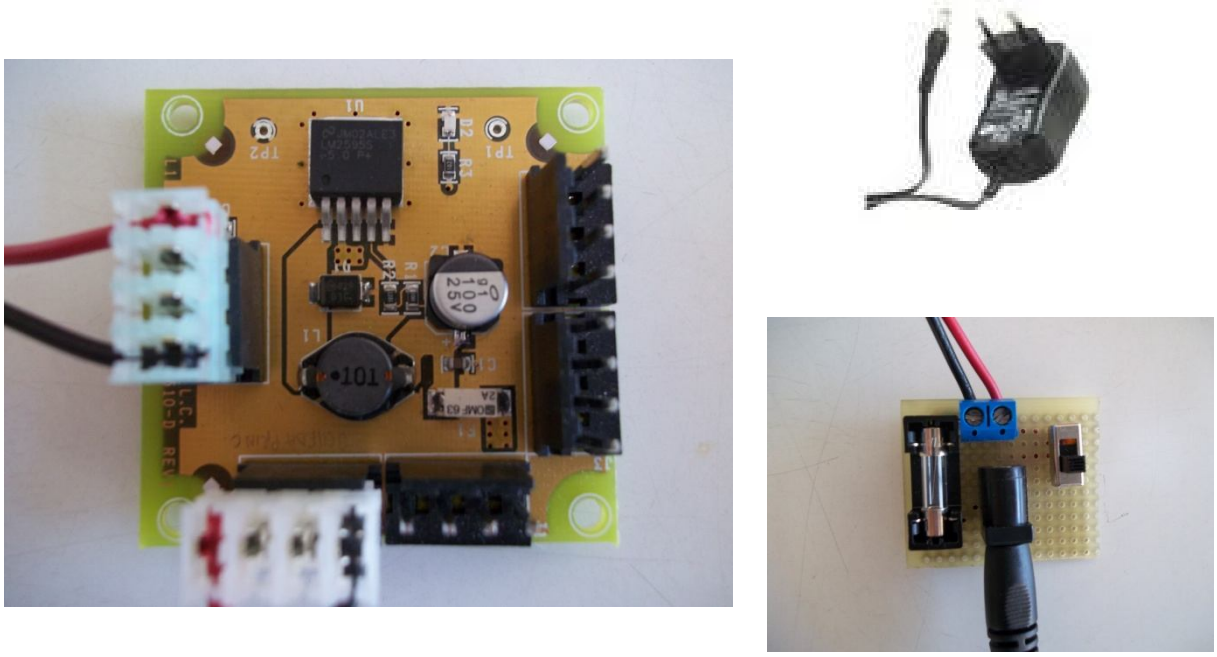

Figura 5.8 Blocco di alimentazione

Per testare separatamente la board di interfaccia e la board base del prototipo stampato, è stata realizzata un'altra scheda mille fori che realizza il collegamento tra la scheda della Brain Technology con la board base, in questo modo si è accertato il funzionamento corretto di quella parte di logica.

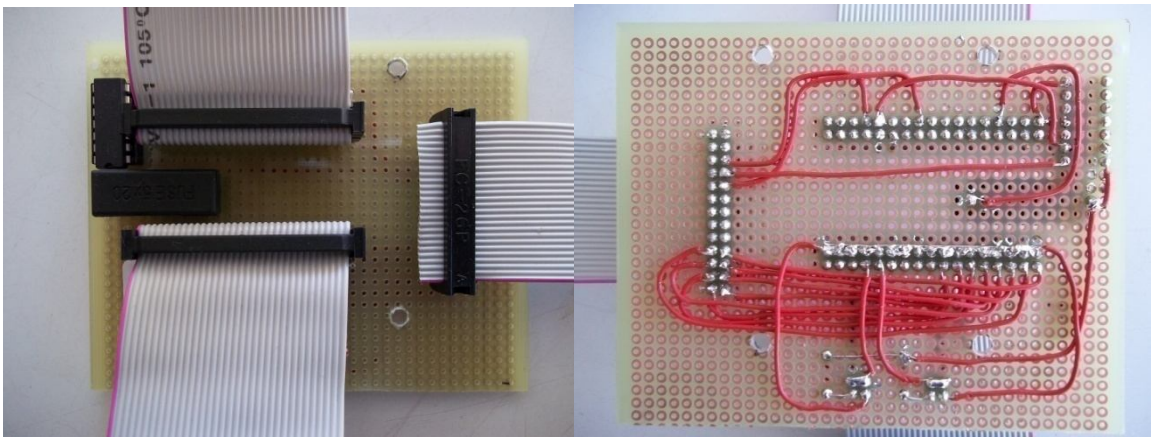

Figura 5.9 Adattatore per il prototipo

CAPITOLO VI

Sviluppo del Firmware

Ogni dispositivo elettronico complesso necessita di un particolare programma di inizializzazione il cui scopo è quello di avviare il componente stesso e consentirgli di interagire con altri componenti tramite l'implementazione di protocolli di comunicazione o interfacce di programmazione.

Questo programma viene chiamato firmware e non è altro che il punto di incontro fra componenti logiche e fisiche, ossia tra hardware e software.

Nel presente lavoro, è stato anche finalizzata alla preparazione del firmware per il controller FX2 da utilizzare per la telecamera stereo; il capitolo corrente ha appunto lo scopo di chiarire come esso sia stato sviluppato e quali sono state le scelte effettuate.

Cypress, con SuiteUSB, fornisce gli elementi necessari per la configurazione ed i test relativi al controller, questi tool risultano essere un valido ausilio al programmatore soprattutto nella fase iniziale di sperimentazione.

VI.1 Tool per lo sviluppo

Il firmware per l'FX2 è stato scritto utilizzando Keil µVision fornito dalla casa produttrice ARM, questo software è un compilatore C specifico per i micro controllori 8051. In figura 3.8 è mostrata l'interfaccia grafica di lavoro, la corretta inclusione di tutti i file di lavoro forma una struttura ad albero.

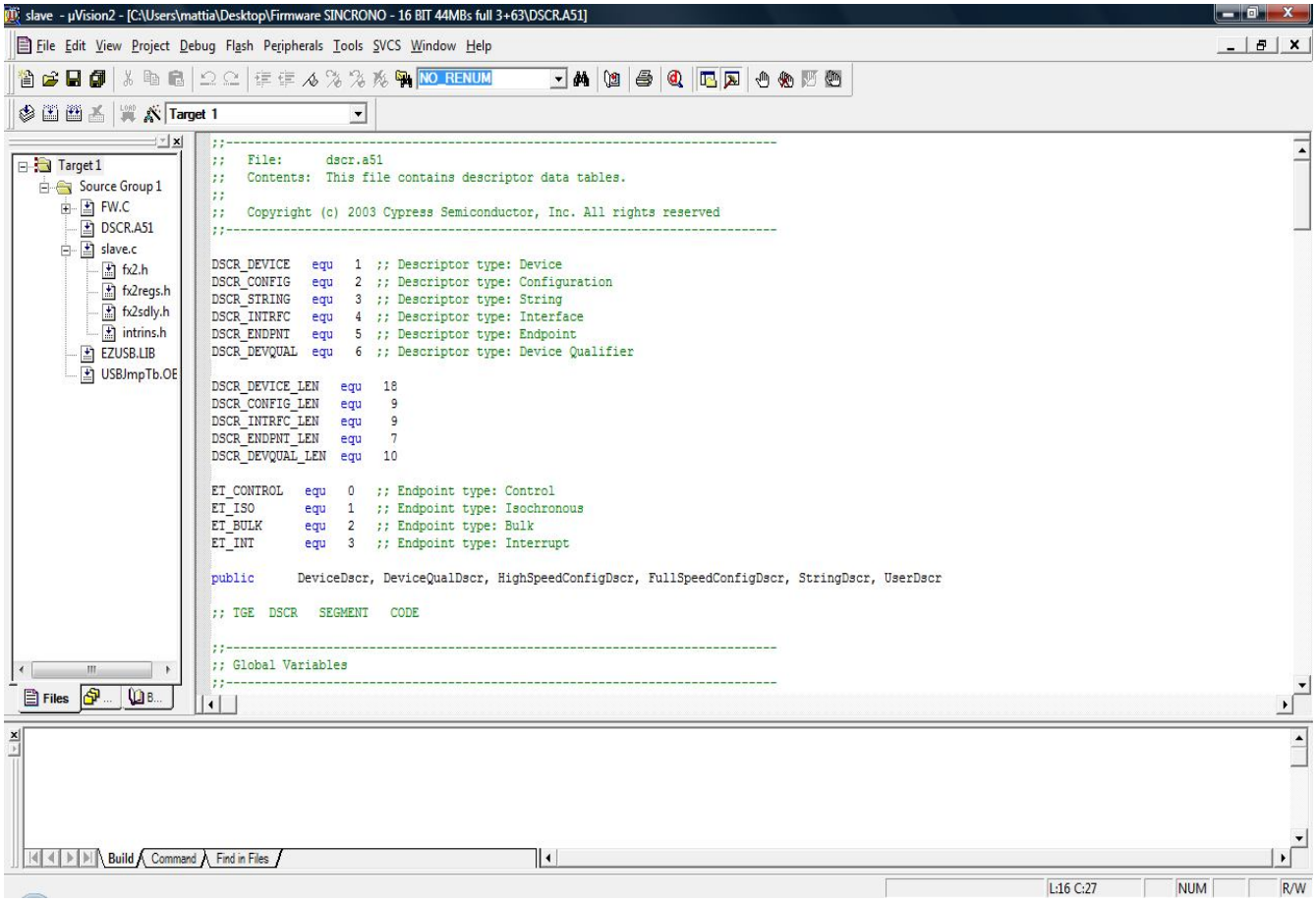

Figura 6.1 Keil micro vision

Keil µVision permette naturalmente di compilare il codice realizzato e di creare il relativo file esadecimale (.HEX) che verrà successivamente caricato, tramite il programma Control Center, nella RAM dell'FX2.

VI.2 Struttura del firmware

Il CYStream Reference Design Package fornito da Cypress è comprensivo di:

1. Frameworks file (fw.c), è la sezione del firmware che risponde alle richieste USB da parte dell'Host ed implementa il processo di enumerazione e rienumerazione.

2. Descriptor file (dscr.a51), questo file contiene i descrittori delle interfacce e degli endpoint. Esistono due differenti configurazioni indipendenti, una per la full speed e l'altra per l'high speed .
3. Slave.c, è il vero e proprio file di configurazione in cui vengono settati i registri dell' FX2 al fine di ottenere il funzionamento desiderato.

Sono inoltre presenti i file header fx2.h, fx2regs.h, fx2sldy.h, intrins.h oltre che la libreria EZUSB.lib, tutti questi file sono necessariamente da includere nel progetto.

E' di uso comune non riscrivere completamente un firmware, ma partire da un template di base (nel nostro caso quello fornito da Cypress) andando ad aggiungere e modificare la configurazione già presente.

Il codice utilizzato per il framework è un codice standard valido per tutti i prodotti CY7C68013 e, dato che non sono state apportate modifiche, verrà omesso il dettaglio implementativo.

Il file che tipicamente subisce le maggiori modifiche da parte del programmatore è il file della configurazione vera e propria (slave.c) e di conseguenza il descriptor file (dscr.a51).

Andremo ora a chiarire come questi due file siano stati sviluppati e quali modifiche sono state apportate, si rimanda all'*Appendice B* per la visione del codice nella sua interezza .

VI.2.1 Il file Descriptor (dscr.a51)

Nel file dscr.a51 sono presenti tre tipi di descrittori:

Il Device Descriptor: Necessario all'Host per capire quale dispositivo è stato connesso. E' in questa parte di codice che vengono settati i nuovi Product_id e Device_id che la periferica avrà dopo il caricamento del firmware.

```
DeviceDscr:
   db   DSCR_DEVICE_LEN      ;; Descriptor length
   db   DSCR_DEVICE          ;; Decriptor type
   dw   0002H                ;; Specification Version (BCD)
   db   00H                  ;; Device class
   db   00H                  ;; Device sub-class
   db   00H                  ;; Device sub-sub-class
   db   64                   ;; Maximum packet size
   dw   0B404H               ;; Vendor ID
   dw   0310H                ;; Product ID (Sample Device)
   dw   0000H                ;; Product version ID
   db   1                    ;; Manufacturer string index
   db   2                    ;; Product string index
   db   0                    ;; Serial number string index
   db   1                    ;; Number of configurations
```

L'Interface Descriptor: E' il descrittore delle varie interfacce (ogni dispositivo USB può averne diversi)

```
Interface Descriptor
   db   DSCR_INTRFC_LEN      ;; Descriptor length
   db   DSCR_INTRFC          ;; Descriptor type
   db   0                    ;; Zero-based index of this interface
   db   0                    ;; Alternate setting
   db   2                    ;; Number of end points
   db   0ffH                 ;; Interface class
   db   00H                  ;; Interface sub class
   db   00H                  ;; Interface sub sub class
   db   0                    ;; Interface descriptor string index
```

L'Endpoint Descriptor: Descrive nel dettaglio come è fatto un endpoint (indirizzo fisico, direzione, tipo ecc..)

```
Endpoint Descriptor
   db   DSCR_ENDPNT_LEN      ;; Descriptor length
   db   DSCR_ENDPNT          ;; Descriptor type
   db   86H                  ;; Endpoint number, and direction
   db   ET_BULK              ;; Endpoint type
```

```
db   00H                  ;; Maximum packet size (LSB)
db   02H                  ;; Maximum packet size (MSB)
db   00H                  ;; Polling interval
```

VI.2.2 Il file Framework (fw.c)

La stesura del Framework necessita della conoscenza approfondita del protocollo USB, l'utilizzo quindi di un codice standard agevola notevolmente il compito del programmatore.

La struttura logica del framework è rappresentata con uno schema a blocchi in figura 6.1 presente nella pagina successiva.

Per prima cosa viene chiamata la funzione TD_init() che ha il compito di modificare i registri dell'Fx2. Una volta che il dispositivo è stato rinumerato viene invocata ripetitivamente la funzione dedita al trasferimento (Td_Poll()). Nel caso in cui non sia rilevata attività sul bus oppure venga esplicitamente mandata una richiesta dall'Host viene chiamata la Td_Suspend() che mette il dispositivo in suspend mode.

Le funzione TD_init(),Td_Poll() e la Td_Suspend() vengono in realtà implementate nel file slave.c perché sono le uniche che andranno modificate, in base alle esigenze, dallo sviluppatore.

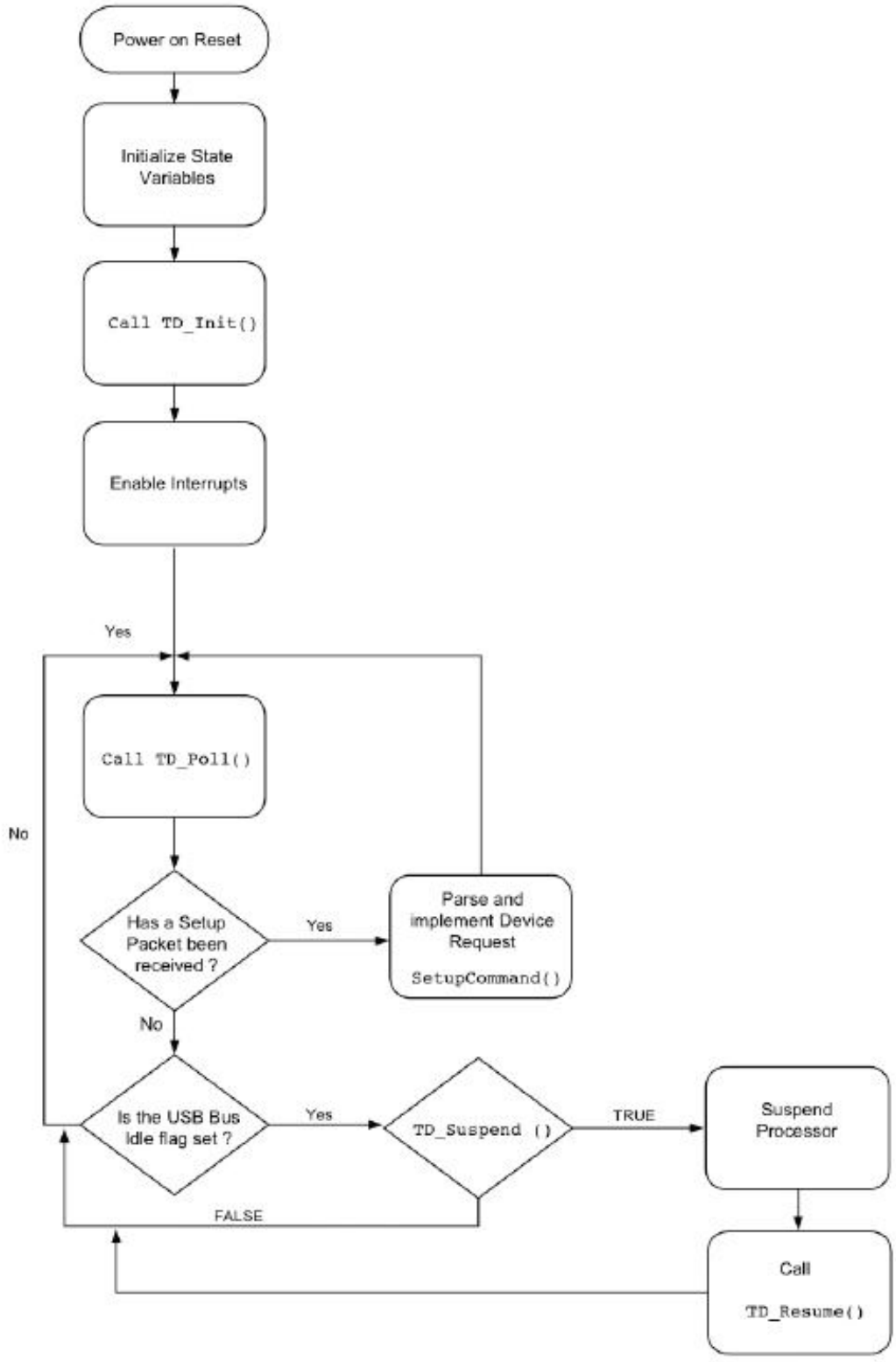

Figura 6.2 Struttura del Framework

VI.2.3 Il file di configurazione (slave.c)

```
void TD_Init( void )
{
  REVCTL=0x03;           // Necessario vedi TRM
  SYNCDELAY;

  CPUCS = 0x10;          // CPU 8051 a 48 MHz
```

Da notare che il flag standard ha evidenziato dei notevoli ritardi nei tempi di risposta, viene perciò utilizzato un flag di FULL programmabile.

```
  EP6FIFOPFH=0x19;       // FLAG programmabile attivo dopo 3 pacchetti
  SYNCDELAY;             // +510 byte
  EP6FIFOPFL=0xFC;
  SYNCDELAY;

  SYNCDELAY;
  PINFLAGSAB = 0x6A;      // FLAG B -> EP6 flag programmabile
  SYNCDELAY;             // FLAG A -> EP6 flag empty

  IFCONFIG = 0x43;       // Modalitaà sincrona con IFCLK esterno, non
                         // ivertito

  SYNCDELAY;
  FIFORESET = 0x80;      // Activate NAK-ALL
  SYNCDELAY;
  SYNCDELAY;
  FIFORESET = 0x06;      // reset, FIFO 6
  SYNCDELAY;
  FIFORESET = 0x00;      // deactivate NAK-ALL

  SYNCDELAY;
  EP2CFG = 0xA0;
  SYNCDELAY;
  EP6CFG = 0xE0;         // endpoint in ingresso, Bulk di dimensione
  SYNCDELAY;             // 512, buffer x4

  EP2FIFOCFG = 0x00;
  SYNCDELAY;
  EP2FIFOCFG = 0x10;

  SYNCDELAY;
  EP6FIFOCFG = 0x09;     // AUTOIN=1, fifo a 16 bit
  SYNCDELAY;

  EP6AUTOINLENH = 0x02   // Auto-commit 512-byte packets
  SYNCDELAY;
  EP6AUTOINLENL = 0x00;
  SYNCDELAY;

  REVCTL=0x00;           // Necessario vedi TRM
  SYNCDELAY;


}
void TD_Poll( void )
```

```
{

  // Niente da fare perchè l'Fx2 è in auto mode, quindi i dati vengono
  // automaticamente commissionati dalla FIFO alla SIE


}

BOOL TD_Suspend( void )
{
  // Per i nostri scopi non è necessario implementare questa funzione
   return( TRUE );
}

BOOL TD_Resume( void )
{
  // Questa funzione viene chiamata quando il controller ritorna alla
  // normale attività dopo una sospensione, anche essa resta vuota.
   return( TRUE );
}
```

Nell'ultima fase della tesi il firmware è stato modificato al fine di avere una comunicazione bidirezionale, in questo modo l'Host può programmare dinamicamente i sensori.

Le variazioni apportate sono le seguenti:

- E' stato configurato l'endpoint 2 come out, bulk, buffer x 4
- E' stata abilitata la modalità auto out
- E' stato configurato FLAG C come empty dell'endpoint 2.

CAPITOLO VII

Sperimentazione

Nel corso della fase di sperimentazione sono state svolte diverse prove finalizzate a massimizzare le prestazioni dell'FX2 in termini di banda in trasmissione.

Ricordiamo che l'ambito in cui il controller USB si andrà a collocare, è un ambiente che tratta una grande quantità di dati streaming in tempo reale in cui non è tollerata la presenza di errori.

Analizzando con l'oscilloscopio i frequenti eventi di full della FIFO dell'FX2 (che ricordiamo avere dimensione 2 KByte) si è evidenziata l'esigenza di una FIFO esterna di dimensioni superiori.

In una prima fase è stata implementata una FIFO software all'interno dell' FPGA Spartan 3 con l'ausilio del FIFO Core Generetor.

Questo tool utilizza le block RAM messe a disposizione dall'FPGA per creare l'architettura di un componente che assume il comportamento di una memoria FIFO.

Purtroppo la struttura interna della spartan 3 non permette di creare FIFO di dimensioni superiori a 16 Kbyte.

Dalle misure sperimentali effettuate successivamente tale valore non sembra sufficiente. Il sistema operativo presenta infatti dei lunghi periodi, di durata anche di alcuni milli secondi, in cui non serve il controller FX2. Questi lunghi periodi, che chiameremo d'ora in poi tempi morti, fanno si che sia la FIFO dell'FX2 sia la FIFO dell' FPGA vadano in overflow con conseguente perdita di dati.

Prima di entrare nel merito delle soluzioni che possono essere adottate per risolvere questo problema, è utile andare a eseguire delle prove sulle massime velocità supportabili dal controller FX2 in condizioni ideali.

Tali misure saranno poi le massime velocità teoriche raggiungibili quando verrà inserita la FIFO sull'FPGA, che indubbiamente peggiorerà le prestazioni globali del sistema.

VII.1.1 Misure sulle massime velocità di trasferimento

Di seguito vengono riportati alcuni grafici che mostrano l'andamento delle velocità di trasferimento (misurate in Mbyte/s) minime, massime e medie al variare della dimensione della singola Bulk read. Vengono presi in esame tre casi: Libusb 0.1 per Windows e Libusb 1.0 sia in ambiente Linux sia in ambiente Windows.

Le misure delle velocità sono state effettuate via software con l'ausilio di OpenCv misurando i tempi necessari al trasferimento e relazionandoli alla sua dimensione.

Un grande passo in avanti per quanto riguarda le velocità medie di trasferimento è stato fatto passando dalla FIFO dell'Fx2 da 8 bit a 16 bit, con cui si è ottenuto un incremento di velocità di 8-9MB/s.

Il controller USB funziona quindi in modalità sincrona a parallelismo 16 bit, con l'FPGA che campiona il pin di full e ferma la trasmissione in caso di overflow.

Il conteggio delle tre velocità è fatto su un trasferimento complessivo di 200 trasferimenti. Sull'asse delle ascisse viene riportato il numero di pacchetti ricevuti per ogni trasferimento (cioè per ogni Bulk transfer), per calcolare quindi la

dimensione del singolo trasferimento è sufficiente moltiplicare il numero di pacchetti per trasferimento per la dimensione del pacchetto bulk (512 Byte).

I test sono stati effettuati in un calcolatore avente due processori dual core a 2.2 GHz e come sistemi operativi Windows 7 a 64 bit e Linux Ubuntu versione 10.10 installati in dual boot.

Da notare che la scelta dell'avviamento dual boot è obbligatoria poiché, montare un sistema operativo in macchina virtuale, ha evidenziato un notevole peggioramento delle prestazioni dell'USB.

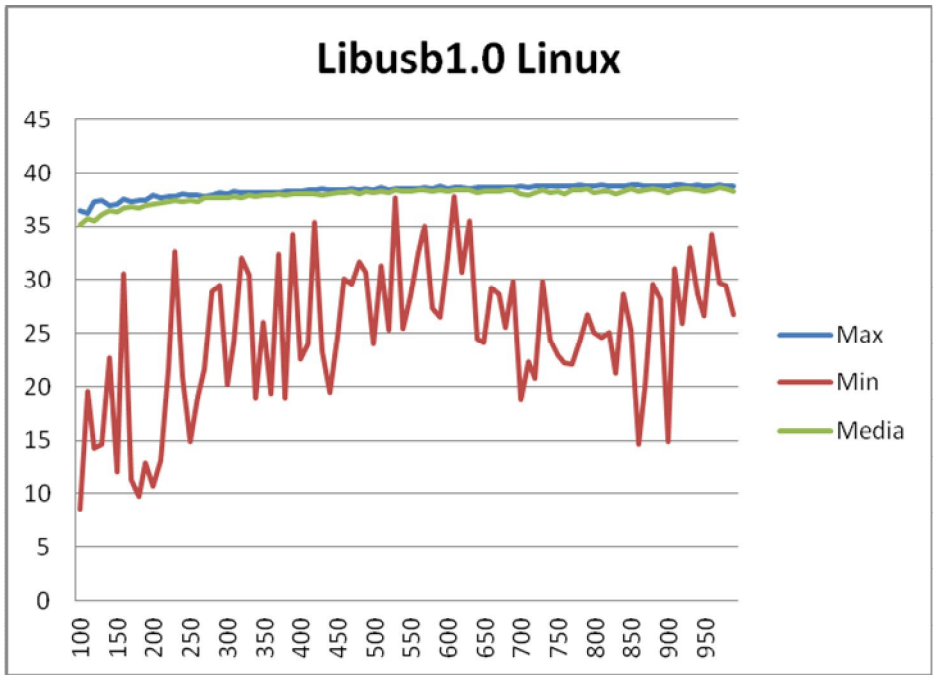

Figura 7.1 Grafico Libusb 1.0 Linux: velocità di trasferimento in MB/s

Massimo assoluto: 39,2 MByte/s ottenuto con trasferimenti di dimensione: 428*512

Minimo assoluto: 8,2 MByte/s ottenuto con trasferimenti di dimensione: 100*512

Media massima: 38,9 MByte/s ottenuto con trasferimenti di dimensione: 466*512

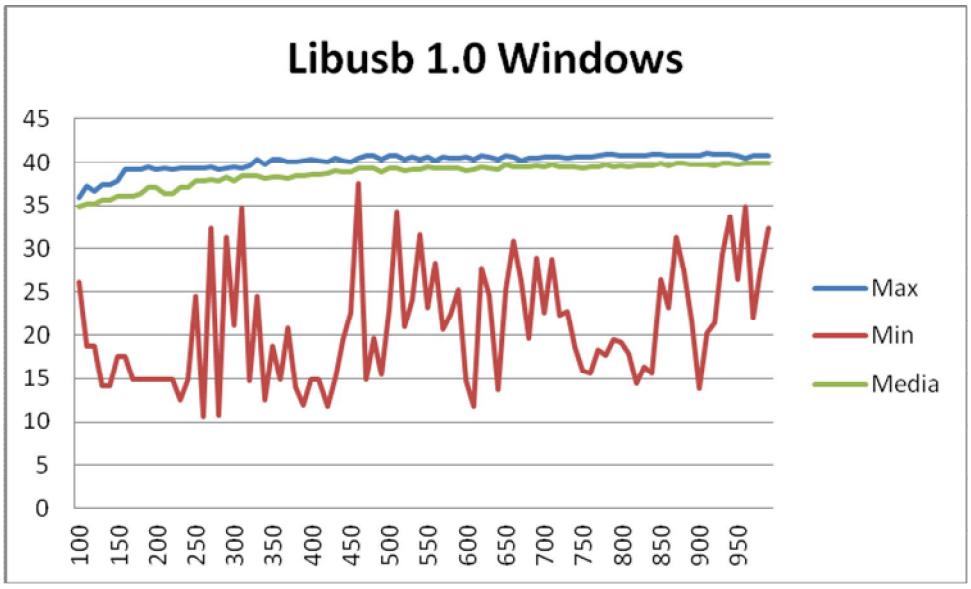

Figura 7.2 Grafico Libusb 1.0 Windows in MB/s

Massimo assoluto: 41,2 MByte/s ottenuto con trasferimenti di dimensione: 401*512

Minimo assoluto: 10,4 MByte/s ottenuto con trasferimenti di dimensione: 266*512

Media massima: 40,5 MByte/s ottenuto con trasferimenti di dimensione: 466*512

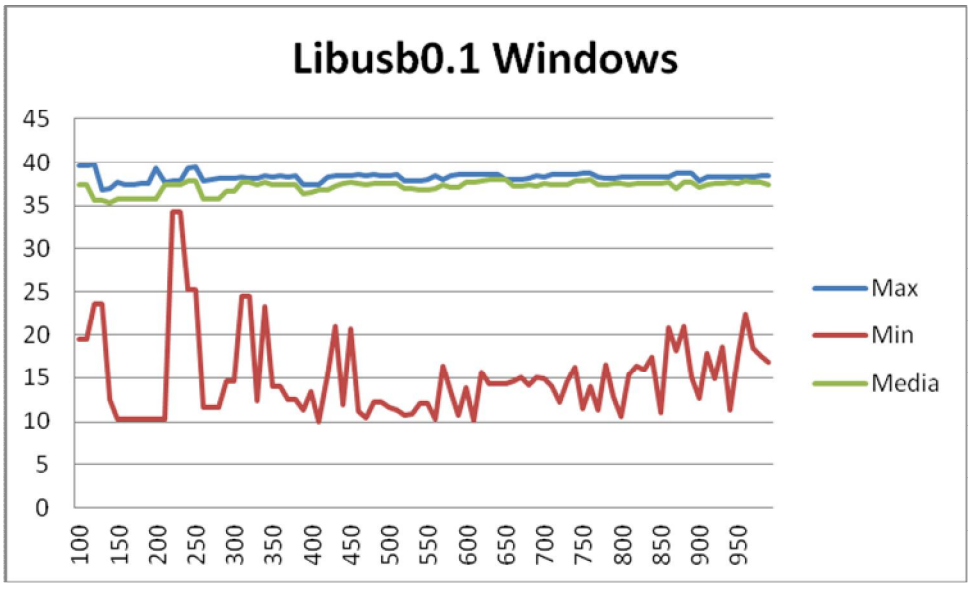

Figura 7.3 Grafico Libusb 0.1 Windows in MB/s

Massimo assoluto: 38,8 MByte/s ottenuto con trasferimenti di dimensione: 121*512

Minimo assoluto: 10,3 MByte/s ottenuto con trasferimenti di dimensione: 173*512

Media massima: 38,2 MByte/s ottenuto con trasferimenti di dimensione: 438*512

Uno dei possibili modi per ovviare al problema degli overflow è andare a modificare il Driver (messo a disposizione da Libusb) creandone uno ad Hoc per la nostra applicazione in modo da ridurre i tempi morti che generano l'overflow, questa strada però, è difficilmente percorribile data la complessità della struttura del driver.

Per capire quali altre soluzioni sono adottabili per risolvere il problema è stata eseguita una dettagliata analisi all'oscilloscopio dell'andamento dei segnali di empty e full dell'FX2 e dell'empty e del full della FIFO presente sull'FPGA.

Questa analisi ha messo in evidenza due aspetti importanti: il primo è che il flusso dati sul canale USB è discontinuo, proprio come ci si aspettava; la seconda considerazione è che quando il controller FX2 è servito dal sistema operativo svuota troppo velocemente le due FIFO, questo da luogo a una serie di eventi di empty che portano a loro volta ad una catena di NACK (da parte dell'FX2) in risposta alla richiesta di nuovi dati dal sistema operativo.

Il protocollo USB da maggiore priorità alle periferiche che hanno un volume maggiore di dati da trasferire, perciò quando l'Host rileva una serie di NACK da parte della periferica, la abbassa sulla catena delle priorità e questo da luogo ai lunghi tempi morti (di cui si parlava in precedenza) in cui il sistema operativo non serve l'FX2 perché lo classifica come una periferica lenta.

Partendo da questa osservazione è attualmente in fase di sviluppo una complessa gestione dalla FIFO interna all'FPGA finalizzata ad eliminare questi eventi di empty.

Il nuovo sistema prevede che la FIFO venga mantenuta sempre ad un livello di riempimento costante grazie all'introduzione di dati *dummy* che andranno poi eliminati in fase di elaborazione dall'Host.

L'avere dati dummy da inviare nel momento in cui non si hanno dati fa si che non avvengano eventi di NACK e che quindi il sistema operativo dia elevata priorità all'FX2.

CAPITOLO VIII

Conclusioni e sviluppi futuri

L'implementazione dell'interfaccia USB 2.0 mediante il controller FX2 ha imposto lo sviluppo di diversi prototipi le cui funzionalità andassero via via complicandosi.
Sono emerse diverse problematiche, sia a livello software sia per quanto riguarda l'aspetto della progettazione hardware.
Il driver messo a disposizione dalla casa costruttrice Cypress si è rivelato complesso e con prestazioni decisamente sotto le aspettative.
Libusb invece, fornisce un driver semplice ed efficace anche se con poche possibilità di personalizzazione, soprattutto nella versione 0.1.
Anche il lato hardware non si è dimostrato esente da difetti, sono infatti emersi sia problemi elettrici, per quanto riguarda la distorsione subita dai segnali, sia problemi strettamente di progetto, inizialmente non era prevista una FIFO esterna che in realtà è risultata essere necessaria.
Tutto ciò ha contribuito a portare dei rallentamenti nel progetto complessivo, ma nonostante le diverse problematiche incontrate, la telecamera stereo funziona con una banda netta di 33MB/s a cui corrispondono circa 30 frame/s (netti) con un singolo sensore, cioè 15 fps in modalità stereo.
Sono ancora attualmente in corso d'opera degli algoritmi implementabili su FPGA per eseguire una compressione non *lossy* delle immagini catturate dai sensori e della mappa di disparità che attualmente viene creata dall' FPGA stessa.

APPENDICE A

Applicazioni utente

A.1 Ambiente Windows

```
#include "stdafx.h"
#include <string.h>
#include <time.h>
#include <windows.h>
#include <fstream>
#include <iostream>
#include "C:\Users\mattia\Desktop\libusb-win32-bin-1.2.2.0\include\usb.h"

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "cv.h"
#include "highgui.h"

#include "Time_Measurement.h"

#define VENDOR_ID 0x04b4
#define PRODUCT_ID 0x1003

const static int ENDPOINT_BULK_IN=0x86;       // Endpoint address for IN
const static int PACKET_BULK_LEN= 512*990 ;   // Byte for each transfer 200
const static int number_transfer= 200;        // Number of transfer

FILE *out;
FILE *tempi;
my_time p_start;

int main() {

 long int index,index1;
 int somma=0,errori=0,er[3],linea_corrente=1;
 long int n_byte;
 struct usb_bus *p;
 struct usb_device *q;
 struct usb_device *current_device;
 float time,frequncy;
 usb_dev_handle *current_handle;

 out = fopen("example.txt","w");
 tempi = fopen("tempi.txt","w");
 if (!out) printf("Error opening file \n");

 n_byte=PACKET_BULK_LEN*number_transfer;

 char * buffer = (char*)malloc(sizeof(char)*n_byte*2);
 //memset(&buffer,0,sizeof(buffer));
 if (buffer==NULL) printf("Error in malloc \n");
```

```
////////////////////////////////////////////////
//            INITIALIZE USB                  //
////////////////////////////////////////////////


usb_init();

er[0]=usb_find_busses();

er[1]=usb_find_devices();

p=usb_busses;

current_device=NULL;
while(p!=NULL)
     {q=p->devices;
     while(q!=NULL){
           if ((q->descriptor.idVendor==VENDOR_ID)&&(q-
                >descriptor.idProduct==PRODUCT_ID))
           current_device=q;
           q=q->next;
           }
     p=p->next;
     }
if (current_device==NULL){
      printf("\n\nCould not find a CY7C68013\n\n");
      system("pause");
      exit(0);
    }
else{
      printf("Find a CY7C68013\n\n");
    }


if((current_handle=usb_open(current_device))) {
  printf("Device open correctly\n");
  }
 else{
  printf("Usb_open failed\n");
  system("pause");
   exit(0);
 }


int set_configuration=usb_set_configuration(current_handle, 1);
printf("Set_configuration %d\n", set_configuration);

er[2]=usb_claim_interface(current_handle, 0);
printf("Claim  %d\n", er[2]);

er[3]=usb_set_altinterface(current_handle, 0);
printf("Set_altinterface %d\n", er[3]);


////////////////////////////////////////////////
//            START THE TRANSFER              //
////////////////////////////////////////////////
```

```
 printf("\n\nPress ok to read \n");
 system("pause");

 my_time start = Get_Time();

 for(index = 0;index < number_transfer; index++) {
    p_start = Get_Time();
    usb_bulk_read(current_handle, ENDPOINT_BULK_IN,buffer+index*PACKET_BULK_LEN,
    PACKET_BULK_LEN, 2000);
 }
 // ----------------------------------------------------
 //-----------------------------------------------------
 my_time end = Get_Time();
 float DeltaTime= ElapsedTime(start,end);
 frequncy=(PACKET_BULK_LEN*number_transfer)/DeltaTime;
 frequncy/=(1024*1024);
 printf("\n Medium Frequency: %3.10f MHz", frequncy);
 //-----------------------------------------------------
 //-----------------------------------------------------

  ////////////////////////////////////////////
 //     CHECK ERRORS AND WRITE IN FILE          //
 ////////////////////////////////////////////

 for(index = 0;index < n_byte; index++) {

            fprintf(out,"%d ",buffer[index]);
            somma+=buffer[index];
                  if(buffer[index]==50) {
                  fprintf(out,"       %d ",somma);
                        if ((somma!=1275) && (somma!=1530)&& (somma!=2500)&&
                              (somma!=50)){
                               errori++;
                               printf("\n Error in line: %d ", linea_corrente);
                        }
                  linea_corrente++;
                  somma=0;
                  fprintf(out,"\n");
                  }
 }

 printf("\n Total errors: %d \n\n", errori);



 ////////////////////////////////////////////
 //         CLOSE FILE AND USB                //
 ////////////////////////////////////////////

 fclose(out);
 fclose(tempi);
 free(buffer);

 usb_release_interface(current_handle, 0);
 usb_close(current_handle);

 system("pause");
 return 0;
}
```

A.2 Ambiente Linux

```
// versione 1.0 -> serve per testare i trasferimenti ed i tempi dei
// trasferimenti e librerie opencv
// g++ -I/usr/include/opencv -L/usr/local/lib -lcv -lusb-1.0 test_USB.c -o prova
// -lcxcore -lhighgui

#define LINUX

#include <errno.h>
#include <signal.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#include "cv.h"
#include "highgui.h"

#include <libusb-1.0/libusb.h>

#define VENDOR_ID 0x04b4
#define PRODUCT_ID 0x1003
// no code above this point
#ifndef UNIQUE_IDENTIFIER
#define UNIQUE_IDENTIFIER

#if (MSC_VER > 1000)
  #pragma once
#endif

#endif
// no code beyond this point

#ifdef LINUX
#include <inttypes.h>
#endif

using namespace cv;

#ifndef LINUX
#ifndef mytime
#define my_time unsigned __int64
#endif
#endif

#ifdef LINUX
typedef uint64_t my_time;
#endif

#ifdef LINUX
// ritorna il tempo al momento della chiamata
my_time Get_Time() {
  return getTickCount();
}

// dato il tempo iniziale e quello finale calcola i secondi trascorsi
```

```
float ElapsedTime(my_time start, my_time end){
  return (end-start)/getTickFrequency();
}

#endif
// stampa il tempo trascorso con n cifre significative
void PrintElapsedTime(char *text, my_time start, my_time end) {
  printf("%s : %3.10f sec\n",text,ElapsedTime(start,end));
  return;
}

// oltre a stampare il tempo trascorso lo dà come valore d'uscita
float GetTimeAndPrintElapsedTime(char *text, my_time start) {
  my_time local_end = Get_Time();
  float res = ElapsedTime(start,local_end);
  printf("%3.10f sec : %s\n",ElapsedTime(start,local_end),text);
  return res;
}

// dato un tempo in secondi lo trasforma in ore e minuti
void Get_HH_MM_SS(int input_overall_seconds, int *hours, int *minutes, int
*seconds) {
      // Funzione che, fornito in input il numero di secondi, ritorna HH, MM, SS
  int overall_time_in_seconds =  (int)(input_overall_seconds);
  *hours    = overall_time_in_seconds/3600;
  *minutes  = (overall_time_in_seconds - *hours*3600)/60;
  *seconds  = (overall_time_in_seconds - *hours*3600 - *minutes*60);
}

const static int PACKET_BULK_LEN = 512*210;  // 200 Number of Byte to transfer
for each transfer
const static int number_transfer = 100;  // Number of transfer
const static int INTERFACE = 0;
const static int ENDPOINT_BULK_IN = 0x86;  // Endpoint 0x81 address for IN

const static int TIMEOUT = 2000;  // Timeout in ms

static struct libusb_device_handle *devh = NULL;
FILE *out, *out_tr;
my_time start;
my_time p_start;

//#############################################################################
//                              FUNCTION FOR TRANSFER
//#############################################################################
  static int test_bulk_transfer(void){
  long int r,index,index1;
  int transferred;
  int somma = 0, errori = 0;
  int linea_corrente = 1;
  long int n_byte;
  int decision;
  n_byte = PACKET_BULK_LEN*number_transfer;
  float ris, freq;
  float max = 0, min = 1000;
  // alloco spazio per leggere dalla scheda
  unsigned char * buffer = (unsigned char*)malloc(sizeof(char)*n_byte*2);
  float *transf = (float*)malloc(sizeof(float)*number_transfer);

  if (buffer == NULL)
    printf("Error in malloc \n");
```

```
  if (transf == NULL)
    printf("Error in malloc \n");

  out = fopen("temp_file.txt","w");
  out_tr = fopen("trans_file.txt","w");

  if (!out)
    printf("Error opening file \n");

  if (!out_tr)
    printf("Error opening file \n");

  start = Get_Time();
  libusb_clear_halt(devh,ENDPOINT_BULK_IN );

  for(index = 0;index < number_transfer; index++) {
    float elapsed_time = -1;
    my_time start_peak, end_peak;
    start_peak = Get_Time();
    // Read data
    r = libusb_bulk_transfer(devh,
    ENDPOINT_BULK_IN,buffer+index*PACKET_BULK_LEN,PACKET_BULK_LEN, &transferred,
    TIMEOUT);

    end_peak = Get_Time();
    elapsed_time = ElapsedTime(start_peak, end_peak);
    // Transfer rate
    freq = (PACKET_BULK_LEN) / elapsed_time;
    freq = freq/1000000;
    // printf("\n*** (Current) Transfer rate: %3.10f [MB/s] ***", freq);
    transf[index] = freq;
    if(freq > max)
      max = freq;
    if(freq < min)
      min = freq;
  }
  ris = GetTimeAndPrintElapsedTime("Total Time: \n ", start);
  freq = (PACKET_BULK_LEN*number_transfer) / ris;
  freq = freq/1000000;
  printf("\n*** (Max) Transfer rate: %3.10f [MB/s] ***\n", max);
  printf("\n*** (Medium) Transfer rate: %3.10f [MB/s] ***\n", freq);
  printf("\n*** (Min) Transfer rate: %3.10f [MB/s] ***\n", min);

//-------------------------------------------------------------------------
//       WRITE IN FILE AND CHECK ERRORS IN THE DATA THAT HAS BEEN RECEIVED
//-------------------------------------------------------------------------

  for(index = 0;index < n_byte; index++) {
    fprintf(out,"%d ",buffer[index]);
    somma += buffer[index];
    if(buffer[index] == 50) {
      fprintf(out,"            %d ",somma);
      if ((somma != 1275) && (somma != 1530) && (somma != 2500) && (somma !=
      50)){ // RICORDA DA GAUSS; n(n+1)/2=1275
      errori++;
      printf("\n Error in line: %d ", linea_corrente);
      }
      linea_corrente++;
      somma = 0;
      fprintf(out,"\n");
    }
```

```
  }
  for(index = 0;index < number_transfer; index++) {
    fprintf(out_tr,"%3.10f ",transf[index]);
    fprintf(out_tr,"\n");
  }
  printf("\n Total errors: %d ", errori);
  fclose(out);
  free(buffer);
  return 0;
}

int main(void) {
  int r = 1;
  // OPEN AND CONFIGURE THE USB DEVICE
  r = libusb_init(NULL);
  if (r < 0) {
    fprintf(stderr, "Failed to initialize libusb\n");
    exit(1);
  }
  devh = libusb_open_device_with_vid_pid(NULL, VENDOR_ID, PRODUCT_ID);
  if (devh == NULL) {
    fprintf(stderr, "Could not find/open device\n");
    goto out;
  }
  printf("\n Successfully find device\n");

  r = libusb_set_configuration(devh, 1);
  if (r < 0) {
    fprintf(stderr, "libusb_set_configuration error %d\n", r);
    goto out;
  }
  printf("Successfully set usb configuration 1\n");

  r = libusb_claim_interface(devh, 0);
  if (r < 0) {
    fprintf(stderr, "libusb_claim_interface error %d\n", r);
    goto out;
  }
  printf("Successfully claimed interface 0\n");

  r =  libusb_set_interface_alt_setting(devh, 0,0);
  if (r < 0) {
    fprintf(stderr, "libusb_set_interface_alt_setting error %d\n", r);
    goto out;
  }
  printf("Successfully set alt interface 0 \n");

  // START BULK TRANSFER
  test_bulk_transfer();
  // RELEASE THE DEVICE
  libusb_release_interface(devh, 0);

out:  //libusb_reset_device(devh);
    libusb_close(devh);
    libusb_exit(NULL);

    printf("\n ok to continue\n");
    getchar();

  return 0;
}
```

APPENDICE B

Firmware sviluppato

```
;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;++                                                                     ++
;;++             File:      dscr.a51                                     ++
;;++       Contents:  This file contains descriptor data tables.        ++
;;++                                                                     ++
;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


DSCR_DEVICE   equ   1  ;; Descriptor type: Device
DSCR_CONFIG   equ   2  ;; Descriptor type: Configuration
DSCR_STRING   equ   3  ;; Descriptor type: String
DSCR_INTRFC   equ   4  ;; Descriptor type: Interface
DSCR_ENDPNT   equ   5  ;; Descriptor type: Endpoint
DSCR_DEVQUAL  equ   6  ;; Descriptor type: Device Qualifier

DSCR_DEVICE_LEN   equ   18
DSCR_CONFIG_LEN   equ    9
DSCR_INTRFC_LEN   equ    9
DSCR_ENDPNT_LEN   equ    7
DSCR_DEVQUAL_LEN  equ   10

ET_CONTROL   equ   0   ;; Endpoint type: Control
ET_ISO       equ   1   ;; Endpoint type: Isochronous
ET_BULK      equ   2   ;; Endpoint type: Bulk
ET_INT       equ   3   ;; Endpoint type: Interrupt

public      DeviceDscr, DeviceQualDscr, HighSpeedConfigDscr,
FullSpeedConfigDscr, StringDscr, UserDscr

;; TGE  DSCR   SEGMENT   CODE

;;-----------------------------------------------------------------------------
;; Global Variables
;;-----------------------------------------------------------------------------

;; TGE      rseg DSCR                ;; locate the descriptor table in on-part
memory.

            cseg at 100H              ;; TODO: this needs to be changed before
release
DeviceDscr:
      db   DSCR_DEVICE_LEN      ;; Descriptor length
      db   DSCR_DEVICE          ;; Decriptor type
      dw   0002H                ;; Specification Version (BCD)
      db   00H                  ;; Device class
      db   00H                  ;; Device sub-class
      db   00H                  ;; Device sub-sub-class
      db   64                   ;; Maximum packet size
      dw  0B404H                ;; Vendor ID
```

```
      dw   0310H               ;; Product ID (Sample Device)
      dw   0000H               ;; Product version ID
      db   1                   ;; Manufacturer string index
      db   2                   ;; Product string index
      db   0                   ;; Serial number string index
      db   1                   ;; Number of configurations

DeviceQualDscr:
      db   DSCR_DEVQUAL_LEN    ;; Descriptor length
      db   DSCR_DEVQUAL        ;; Decriptor type
      dw   0002H               ;; Specification Version (BCD)
      db   00H                 ;; Device class
      db   00H                 ;; Device sub-class
      db   00H                 ;; Device sub-sub-class
      db   64                  ;; Maximum packet size
      db   1                   ;; Number of configurations
      db   0                   ;; Reserved

HighSpeedConfigDscr:
      db   DSCR_CONFIG_LEN     ;; Descriptor length
      db   DSCR_CONFIG         ;; Descriptor type
      db   (HighSpeedConfigDscrEnd-HighSpeedConfigDscr) mod 256 ;; Total Length
(LSB)
      db   (HighSpeedConfigDscrEnd-HighSpeedConfigDscr)  /  256 ;; Total Length
(MSB)
      db   1                   ;; Number of interfaces
      db   1                   ;; Configuration number
      db   0                   ;; Configuration string
      db   10100000b           ;; Attributes (b7 - buspwr, b6 - selfpwr, b5 -
rwu)
      db   50                  ;; Power requirement (div 2 ma)

;; Interface Descriptor
      db   DSCR_INTRFC_LEN     ;; Descriptor length
      db   DSCR_INTRFC         ;; Descriptor type
      db   0                   ;; Zero-based index of this interface
      db   0                   ;; Alternate setting
      db   2                   ;; Number of end points
      db   0ffH                ;; Interface class
      db   00H                 ;; Interface sub class
      db   00H                 ;; Interface sub sub class
      db   0                   ;; Interface descriptor string index

;; Endpoint Descriptor
      db   DSCR_ENDPNT_LEN     ;; Descriptor length
      db   DSCR_ENDPNT         ;; Descriptor type
      db   02H                 ;; Endpoint number, and direction
      db   ET_BULK             ;; Endpoint type
      db   00H                 ;; Maximum packet size (LSB)
      db   02H                 ;; Maximum packet size (MSB)
      db   00H                 ;; Polling interval


;; Endpoint Descriptor
      db   DSCR_ENDPNT_LEN     ;; Descriptor length
      db   DSCR_ENDPNT         ;; Descriptor type
      db   86H                 ;; Endpoint number, and direction
      db   ET_BULK             ;; Endpoint type
      db   00H                 ;; Maximum packet size (LSB)
      db   02H                 ;; Maximum packet size (MSB)
      db   00H                 ;; Polling interval
```

```
HighSpeedConfigDscrEnd:

FullSpeedConfigDscr:
      db   DSCR_CONFIG_LEN      ;; Descriptor length
      db   DSCR_CONFIG          ;; Descriptor type
      db   (FullSpeedConfigDscrEnd-FullSpeedConfigDscr) mod 256 ;; Total Length
(LSB)
      db   (FullSpeedConfigDscrEnd-FullSpeedConfigDscr)  /  256 ;; Total Length
(MSB)
      db   1                    ;; Number of interfaces
      db   1                    ;; Configuration number
      db   0                    ;; Configuration string
      db   10100000b            ;; Attributes (b7 - buspwr, b6 - selfpwr, b5 -
rwu)
      db   50                   ;; Power requirement (div 2 ma)

;; Interface Descriptor
      db   DSCR_INTRFC_LEN      ;; Descriptor length
      db   DSCR_INTRFC          ;; Descriptor type
      db   0                    ;; Zero-based index of this interface
      db   0                    ;; Alternate setting
      db   2                    ;; Number of end points
      db   0ffH                 ;; Interface class
      db   00H                  ;; Interface sub class
      db   00H                  ;; Interface sub sub class
      db   0                    ;; Interface descriptor string index

;; Endpoint Descriptor
      db   DSCR_ENDPNT_LEN      ;; Descriptor length
      db   DSCR_ENDPNT          ;; Descriptor type
      db   04H                  ;; Endpoint number, and direction
      db   ET_BULK              ;; Endpoint type
      db   40H                  ;; Maximum packet size (LSB)
      db   00H                  ;; Maximum packet size (MSB)
      db   00H                  ;; Polling interval

;; Endpoint Descriptor
      db   DSCR_ENDPNT_LEN      ;; Descriptor length
      db   DSCR_ENDPNT          ;; Descriptor type
      db   88H                  ;; Endpoint number, and direction
      db   ET_BULK              ;; Endpoint type
      db   40H                  ;; Maximum packet size (LSB)
      db   00H                  ;; Maximum packet size (MSB)
      db   00H                  ;; Polling interval

FullSpeedConfigDscrEnd:

StringDscr:

StringDscr0:
      db   StringDscr0End-StringDscr0      ;; String descriptor length
      db   DSCR_STRING
      db   09H,04H
StringDscr0End:

StringDscr1:
      db   StringDscr1End-StringDscr1      ;; String descriptor length
      db   DSCR_STRING
      db   'C',00
      db   'y',00
```

```
      db   'p',00
      db   'r',00
      db   'e',00
      db   's',00
      db   's',00
StringDscr1End:

UserDscr:
      dw   0000H
      end



;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;++                                                                      ++
;;++             File:      slave.c                                       ++
;;++             Contents:  Hooks required to implement USB               ++
;;++               peripheral function.                                   ++
;;++               Code written for FX2 REVE 56-pin and above.            ++
;;++               This firmware is used to demonstrate FX2 Slave FIFO    ++
;;++               operation.                                             ++
;;++                                                                      ++
;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#pragma NOIV                    // Do not generate interrupt vectors



#include "fx2.h"
#include "fx2regs.h"
#include "fx2sdly.h"            // SYNCDELAY macro

#define LED_ALL         (bmBIT0 | bmBIT1 | bmBIT2 | bmBIT3)

extern BOOL GotSUD;             // Received setup data flag
extern BOOL Sleep;
extern BOOL Rwuen;
extern BOOL Selfpwr;

BYTE Configuration;             // Current configuration
BYTE AlternateSetting;          // Alternate settings
static WORD xdata LED_Count = 0;
static BYTE xdata LED_Status = 0;


//-----------------------------------------------------------------------------
// Task Dispatcher hooks
// The following hooks are called by the task dispatcher.
//-----------------------------------------------------------------------------
void LED_Off (BYTE LED_Mask);
void LED_On (BYTE LED_Mask);

//-----------------------------------------------------------------------------
// Task Dispatcher hooks
//   The following hooks are called by the task dispatcher.
//-----------------------------------------------------------------------------
void TD_Init( void )
{ // Called once at startup
```

```
 REVCTL=0x03;
 SYNCDELAY;


  CPUCS = 0x10; // CLKSPD[1:0]=10, for 48MHz operation, output CLKOUT
//  FIFOPINPOLAR |= 0x03;


  EP6FIFOPFH=0x19;
  SYNCDELAY;
  EP6FIFOPFL=0xFC;
  SYNCDELAY;



  SYNCDELAY;
  PINFLAGSAB = 0x6A;
  SYNCDELAY;
  PINFLAGSCD = 0xCD;
  SYNCDELAY;


  IFCONFIG = 0x43;


  SYNCDELAY;
  FIFORESET = 0x80;             // activate NAK-ALL to avoid race conditions
  SYNCDELAY;                    // see TRM section 15.14
  FIFORESET = 0x02;             // reset, FIFO 2
  SYNCDELAY;                    //
  FIFORESET = 0x04;             // reset, FIFO 4
  SYNCDELAY;                    //
  FIFORESET = 0x06;             // reset, FIFO 6
  SYNCDELAY;                    //
  FIFORESET = 0x08;             // reset, FIFO 8
  SYNCDELAY;                    //
  FIFORESET = 0x00;             // deactivate NAK-ALL


  SYNCDELAY;
  EP2CFG = 0xA0;
  SYNCDELAY;
  EP6CFG = 0xE0;


  SYNCDELAY;
  EP2FIFOCFG = 0x00;            // AUTOOUT=0, WORDWIDE=0

  // core needs to see AUTOOUT=0 to AUTOOUT=1 switch to arm endp's

  SYNCDELAY;                    //
  EP2FIFOCFG = 0x10;            // AUTOOUT=1, WORDWIDE=0

  SYNCDELAY;                    //
  EP6FIFOCFG = 0x09;            // AUTOIN=1, ZEROLENIN=1, WORDWIDE=0

  SYNCDELAY;

  EP6AUTOINLENH = 0x02; // Auto-commit 512-byte packets
  SYNCDELAY;
  EP6AUTOINLENL = 0x00;
```

```
  SYNCDELAY;
  REVCTL=0x00;
   SYNCDELAY;


}

void TD_Poll( void )
{ // Called repeatedly while the device is idle

  // ...nothing to do... slave fifo's are in AUTO mode...


}

BOOL TD_Suspend( void )
{ // Called before the device goes into suspend mode
   return( TRUE );
}

BOOL TD_Resume( void )
{ // Called after the device resumes
   return( TRUE );
}

//-----------------------------------------------------------------------------
// Device Request hooks
//   The following hooks are called by the end point 0 device request parser.
//-----------------------------------------------------------------------------
BOOL DR_GetDescriptor( void )
{
   return( TRUE );
}

BOOL DR_SetConfiguration( void )
{ // Called when a Set Configuration command is received

  if( EZUSB_HIGHSPEED( ) )
  { // ...FX2 in high speed mode
    EP6AUTOINLENH = 0x02;
    SYNCDELAY;
    EP8AUTOINLENH = 0x02;   // set core AUTO commit len = 512 bytes
    SYNCDELAY;
    EP6AUTOINLENL = 0x00;
    SYNCDELAY;
    EP8AUTOINLENL = 0x00;
  }
  else
  { // ...FX2 in full speed mode
    EP6AUTOINLENH = 0x00;
    SYNCDELAY;
    EP8AUTOINLENH = 0x00;   // set core AUTO commit len = 64 bytes
    SYNCDELAY;
    EP6AUTOINLENL = 0x40;
    SYNCDELAY;
    EP8AUTOINLENL = 0x40;
  }

  Configuration = SETUPDAT[ 2 ];
  return( TRUE );        // Handled by user code
}
```

```
BOOL DR_GetConfiguration( void )
{ // Called when a Get Configuration command is received
   EP0BUF[ 0 ] = Configuration;
   EP0BCH = 0;
   EP0BCL = 1;
   return(TRUE);          // Handled by user code
}

BOOL DR_SetInterface( void )
{ // Called when a Set Interface command is received
   AlternateSetting = SETUPDAT[ 2 ];
   return( TRUE );        // Handled by user code
}

BOOL DR_GetInterface( void )
{ // Called when a Set Interface command is received
   EP0BUF[ 0 ] = AlternateSetting;
   EP0BCH = 0;
   EP0BCL = 1;
   return( TRUE );        // Handled by user code
}

BOOL DR_GetStatus( void )
{
   return( TRUE );
}

BOOL DR_ClearFeature( void )
{
   return( TRUE );
}

BOOL DR_SetFeature( void )
{
   return( TRUE );
}

BOOL DR_VendorCmnd( void )
{
  return( TRUE );
}

//-----------------------------------------------------------------------------
// USB Interrupt Handlers
//   The following functions are called by the USB interrupt jump table.
//-----------------------------------------------------------------------------

// Setup Data Available Interrupt Handler
void ISR_Sudav( void ) interrupt 0
{
   GotSUD = TRUE;         // Set flag
   EZUSB_IRQ_CLEAR( );
   USBIRQ = bmSUDAV;      // Clear SUDAV IRQ
}

// Setup Token Interrupt Handler
void ISR_Sutok( void ) interrupt 0
{
   EZUSB_IRQ_CLEAR( );
   USBIRQ = bmSUTOK;      // Clear SUTOK IRQ
```

```
}

void ISR_Sof( void ) interrupt 0
{
   EZUSB_IRQ_CLEAR( );
   USBIRQ = bmSOF;        // Clear SOF IRQ
}

void ISR_Ures( void ) interrupt 0
{
   if ( EZUSB_HIGHSPEED( ) )
   {
      pConfigDscr = pHighSpeedConfigDscr;
      pOtherConfigDscr = pFullSpeedConfigDscr;
   }
   else
   {
      pConfigDscr = pFullSpeedConfigDscr;
      pOtherConfigDscr = pHighSpeedConfigDscr;
   }

   EZUSB_IRQ_CLEAR( );
   USBIRQ = bmURES;       // Clear URES IRQ
}

void ISR_Susp( void ) interrupt 0
{
   Sleep = TRUE;
   EZUSB_IRQ_CLEAR( );
   USBIRQ = bmSUSP;
}

void ISR_Highspeed( void ) interrupt 0
{
   if ( EZUSB_HIGHSPEED( ) )
   {
      pConfigDscr = pHighSpeedConfigDscr;
      pOtherConfigDscr = pFullSpeedConfigDscr;
   }
   else
   {
      pConfigDscr = pFullSpeedConfigDscr;
      pOtherConfigDscr = pHighSpeedConfigDscr;
   }

   EZUSB_IRQ_CLEAR( );
   USBIRQ = bmHSGRANT;
}

}
```

APPENDICE C

Cenni sul bus I^2C

La comunicazione fra due o più dispositivi elettronici si realizza attraverso un insieme di linee, chiamate bus. Affinché gli elementi interagiscano correttamente è necessario stabilire delle specifiche di comunicazione, l'insieme delle regole di trasmissione prende il nome di protocollo.

La Philips ha brevettato un protocollo di comunicazione, chiamato I^2C, il quale consente il colloquio fra più dispositivi compatibili. Questo bus è di tipo seriale sincrono e prevede una linea per il clock ed una per i dati. Le due linee sono di tipo PULL-UP, ovvero sono normalmente allo stato logico alto e vengono portate a quello basso da una qualunque delle periferiche collegate.

Ci sono due tipi di periferiche collegate al bus, quella master e quella slave.

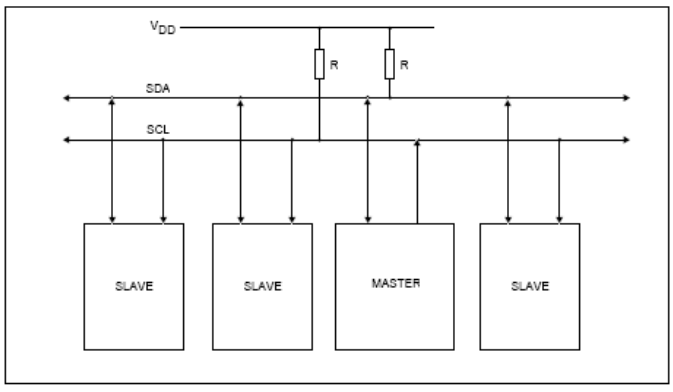

Figura A.1 Schema di collegamento in un bus I^2C

Il dispositivo che si comporta come master genera le sequenze di start e di stop, oltre al clock che viene utilizzato per la comunicazione. Il segnale di clock è presente solo nel momento in cui un byte deve essere trasferito, mentre normalmente la linea ha un livello basso.

La comunicazione sul bus avviene (salvo casi particolari) fra un master e alcune periferiche slave. Il trasferimento comincia quando il master segnala lo start, cioè porta SDA da alto a basso con SCL stabilmente alto.

Per identificare il dispositivo con cui il master intende comunicare, il protocollo prevede che, subito dopo una sequenza di start, venga trasmesso un byte che contiene l'indirizzo dello slave con il quale la comunicazione si deve svolgere. Tale indirizzo è specificato dai sette bit più significativi del byte, mentre il bit meno significativo ha il compito di specificare la funzione che il master richiede: 0 = WRITE, 1 = READ (entrambi riferiti al master).

I successivi dati su SDA vengono trasmessi un byte alla volta, a cui segue una conferma di avvenuta ricezione da parte del dispositivo ricevente interessato, che al nono colpo di clock porta la linea SDA bassa per segnalare l'*Acknowledge*, se questo non dovesse accadere, è necessario ricominciare la trasmissione.

Il dispositivo slave può anche mettere in stato di attesa il master mantenendo a livello basso la linea di clock SCL. Al termine del periodo necessario al slave per rendersi di nuovo disponibile per la comunicazione, viene rilasciata la linea SCL, che può quindi essere riportata alta dal master per l'inizio della comunicazione del byte successivo.

La transazione termina con uno stop, inviato dal master portando SDA da basso ad alto con la linea di clock alta.

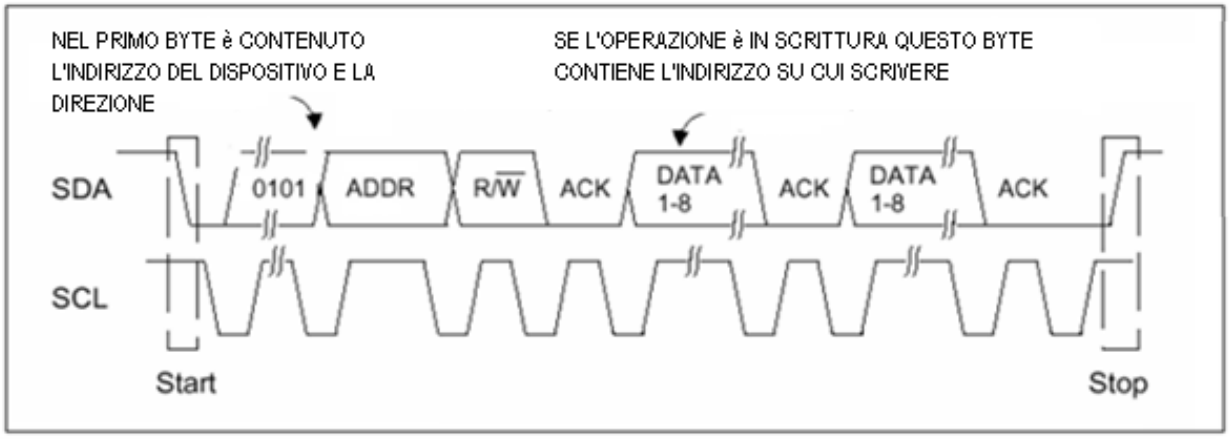

Figura A.2 Andamento dei segnali in un bus I^2C

Come già detto ogni dispositivo presente sul bus è selezionabile tramite un indirizzo contenuto nel primo byte della trasmissione; questo primo byte di controllo è chiamato select code ed è strutturato in questo modo:

- i primi quattro bit rappresentano l'indirizzo standard: i dispositivi I^2C sono suddivisi in classi, a ogni gruppo è preassegnato un indirizzo, per esempio quello delle eprom è 1010.
- gli altri tre bit sono l'indirizzo dello specifico dispositivo.

  Riportandoci all'esempio precedente possiamo connettere al bus fino a otto banchi di eprom.
- l'ultimo bit è il read write (negato).

Riferimenti Bibliografici

[1] Universal Serial Bus Specification, http://www.usb.org

[2] LibUsb, http://www.libusb.org/

[3] EZ-USB FX2 Technical Reference Manual, http://www.cypress.com

Application note: AN50963,AN4067, EP63787, AN4053

[4] CY7C68013 EZ-USB FX2, http://www.comsec.com

[5] Spartan-3E FPGA Family: Data Sheet, http://www.xilinx.com

[6] Spartan-3Starter Kit Board User Guide, http://www.digilentinc.com

[7] Spartan-6Hardware User Guide, http://www.xilinx.com

[8] EZ-USB General Purpose Driver Specification, http://www.cypress.com

[9] CY22393 Datasheet (Programmable PLL), http://www.cypress.com

[10] DS92LV16 *Datasheet* (*16Bit Serializer/Deserializer*), *http://www.national.com*

[11] MT9V032 Datasheet (Image sensor VGA CMOS), http://www.aptina.com

[12] PIC18F4680 Datasheet (Microcontroller), http://www.microchip.com

[13] Braintechnology, http://www.braintechnology.de

Elenco delle figure