

Relazione di progetto: interfaccia USB-PATA/IDE su scheda STM32F4DISCOVERY

Esame di Progettazione di Sistemi Operativi

Gianluca Nitti (mat. 939851)

1 Introduzione

ATA (Advanced Technology Attachment) è uno dei protocolli di comunicazione più utilizzati all'interno degli elaboratori per connettere alla CPU un'unità di archiviazione di massa, come un hard disk o un lettore di dischi ottici. Questo standard ha subito varie evoluzioni nel corso della storia dell'informatica, e la sua variante ad oggi più diffusa è quella seriale (SATA), che lavora a velocità di trasferimento dati nell'ordine dei Gbit/s. Prima della sua introduzione, avvenuta nel 2003, lo standard prevedeva invece come livello fisico un bus parallelo a 16 bit, da cui il nome di Parallel ATA o PATA, interfaccia conosciuta anche con il nome di IDE (Integrated Drive Electronics).

USB (Universal Serial Bus) è invece l'interfaccia ad oggi più diffusa per interconnettere dispositivi e periferiche per computer di vario tipo; lo standard definisce un protocollo di comunicazione comune per vari tipi di dispositivi, rendendone possibile l'utilizzo "plug and play": una tastiera o un hard disk USB, ad esempio, possono essere utilizzati su un PC senza necessità di installare driver specifici del produttore poiché i loro firmware implementano rispettivamente l'interfaccia USB HID¹ o quella MSC², i cui driver sono comuni a tutti i dispositivi della stessa classe ed inclusi nei principali sistemi operativi.

Il progetto consiste nello sviluppo di un dispositivo che agisca da *host* PATA/IDE e contemporaneamente da *device* di archiviazione di massa USB, consentendo quindi di leggere e scrivere dati su hard disk con interfaccia IDE da un computer con solo porte USB.

Il dispositivo è stato testato con successo con i tre principali sistemi operativi desktop (Linux, Windows, MacOS) e con una Smart TV (su cui è stato possibile riprodurre files multimediali contenuti nel disco IDE), e con due diversi hard disk. È utile infine notare che sia lo standard ATA che quello USB MSC definiscono un dispositivo di storage come un semplice archivio di un certo numero di blocchi di dati di dimensione fissa (512 bytes ciascuno), senza alcun vincolo riguardo al layout delle partizioni o ai filesystem contenuti in ciascuna di esse; ne consegue che caratteristiche come il numero di partizioni o il tipo di filesystem presenti in un disco non influiscono sul funzionamento del dispositivo realizzato.

2 Architettura hardware

La piattaforma scelta per la realizzazione del progetto è la scheda di sviluppo STM32F4DISCOVERY, basata sul microcontrollore STM32F407VG prodotto da STMicroelectronics (la stessa utilizzata durante il corso). L'hardware che implementa l'interfaccia USB 1.1 (esposta con una porta di tipo micro-B sulla scheda di sviluppo, dal lato opposto a quello su cui si trova la porta mini-USB per programmazione e debugging) è già integrato nel microcontrollore, ed accessibile tramite apposite librerie³; l'interfaccia PATA/IDE è invece realizzata pilotando alcuni pin GPIO secondo quanto descritto dallo standard che definisce l'interfaccia.

¹Human Interface Device

²Mass Storage Class

³Sia come *host*, ovvero simulando un PC, sia, come in questo caso, come *device*, ovvero nel ruolo di una periferica

2.1 Interfaccia IDE: caratteristiche elettriche

Come specificato nello standard [1], il bus prevede la presenza di un host (in questo caso il microcontrollore) e di uno o due device (ad esempio un hard disk ed un lettore di dischi ottici) connessi tramite un cavo a nastro con tre connettori da 40 pin. Di seguito sono riportati i segnali rilevanti⁴:

- **DD0–DD15:** bus dati a 16 bit, bidirezionale (può essere pilotato sia dall'host che dal device, a seconda del tipo di operazione in corso).
- **CS0–CS1, DA0–DA2:** bus indirizzi; pilotato sempre dall'host, che lo utilizza per selezionare quale dei registri della periferica deve essere letto o scritto.
- **DIOR, DIOW:** comandi (rispettivamente) di lettura e scrittura, entrambi pilotati dall'host con logica negativa; vengono mantenuti entrambi normalmente alti, ed abbassati per leggere o scrivere il registro selezionato della periferica.
- **RESET:** consente all'host di richiedere un reset del device ai parametri di default (equivalente ad un *power cycle*).

Ognuno dei segnali viene mappato su un pin GPIO del microcontrollore; in particolare si è cercato di mappare il bus dati su pin contigui della stessa porta, in modo da potervi accedere con il minor numero di istruzioni possibili. Non è tuttavia stato possibile utilizzare un'unica porta da 16 bit a causa delle periferiche già presenti sulla scheda di sviluppo, che potrebbero interferire o danneggiarsi se i pin a cui sono connesse (documentati nella *Table 7* del manuale [10] e nello schema [9]) venissero condivisi con l'interfaccia IDE qui descritta. Si sono quindi utilizzati 4 bit della PORT D (PD0–PD3) per i segnali DD0–DD3, e 12 bit della PORT E (PE4–PE15) per DD4–DD15.

I segnali dello standard PATA sono a 5V, mentre i GPIO del microcontrollore sono progettati per operare a 3.3V; sul bus dati sono quindi interposti, tra microcontrollore e periferica, due convertitori di livelli logici bidirezionali da 8 bit, al fine di evitare che i segnali a 5V generati dalla periferica possano danneggiare l'MCU. L'integrato utilizzato è il TXS0108E di Texas Instruments [6]. Per gli altri segnali di controllo questa accortezza non è necessaria in quanto sono monodirezionali e sempre pilotati dal dispositivo a tensione più bassa (il microcontrollore), che è superiore alla soglia specificata dallo standard per rappresentare un 1 logico (2V). Su tutte le linee di segnale dell'interfaccia sono inoltre presenti le resistenze di terminazione in serie richieste dallo standard.

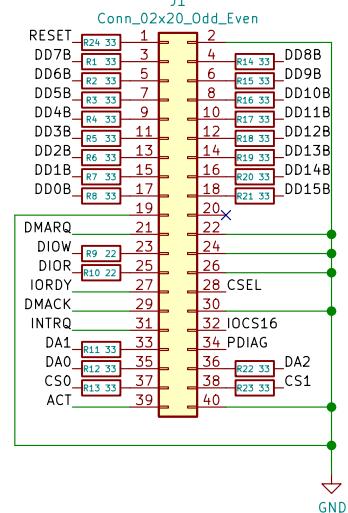
2.2 PCB e assemblaggio fisico

La circuiteria sopra descritta e mostrata nello schema elettrico nella pagina seguente è stata realizzata su un circuito stampato appositamente progettato per potervi installare sopra la scheda STM32F4DISCOVERY e collegare un cavo IDE standard (con connettore da 2×20 pin di cui il n. 20 rimosso per garantire l'inserimento nel verso corretto). Sono inoltre presenti due terminali a vite per collegare l'alimentazione a 5V che viene fornita ai TXS0108E; quando la scheda MCU non è connessa ad un PC per il debugging (in tal caso, riceve alimentazione dal cavo USB di debugging) è infine possibile alimentare anch'essa dalla stessa sorgente chiudendo il jumper J10.

⁴Lo standard definisce anche altri segnali, qui omessi poiché rilevanti solo in modalità di utilizzo dell'interfaccia non necessarie nel contesto di questo progetto

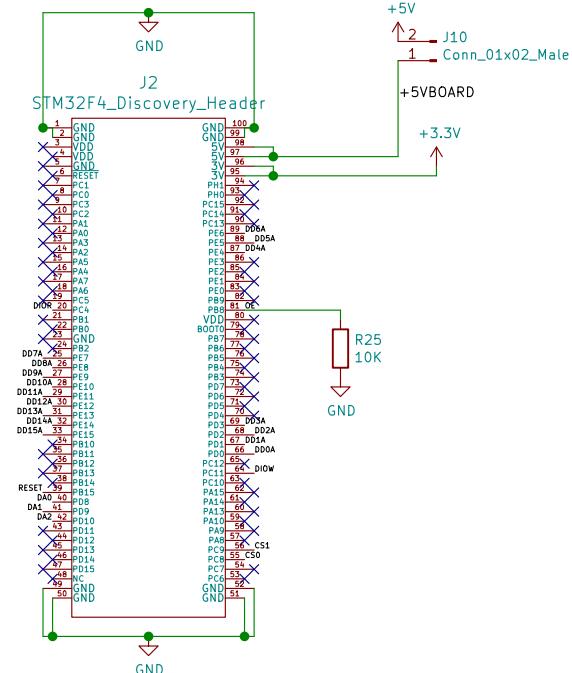
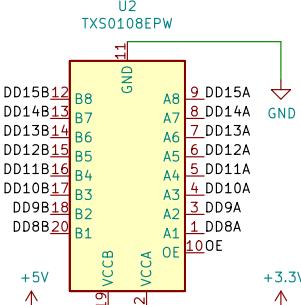
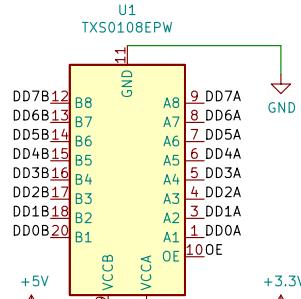
J5
Conn_01x01_Male
1 DMARQ
2 IORDY
3 DMACK
4 INTREQ

J8
Conn_01x01_Male
1 ACT
2 CSEL
3 Conn_01x02_Male
4 TOCS16
5 PDIAG



+5V
1 J3 Screw_Terminal_01x02
2 GND

J4
USB_B
+5V
1 VBUS
2 Shield
3 GND
4 D+
5 D-



J1: connettore periferica PATA/IDE
J2: socket per la scheda STM32F4DISCOVERY
J3: connettore alimentazione 5V periferica
J4: (opzionale) porta usb per alimentazione 5V

Sheet: /
File: ide-usb-pcb.sch

Title: Scheda di interfaccia PATA/IDE per STM32F4DISCOVERY

Size: A4 Date:
KiCad E.D.A. kicad 5.1.10

Rev:
Id: 1/1

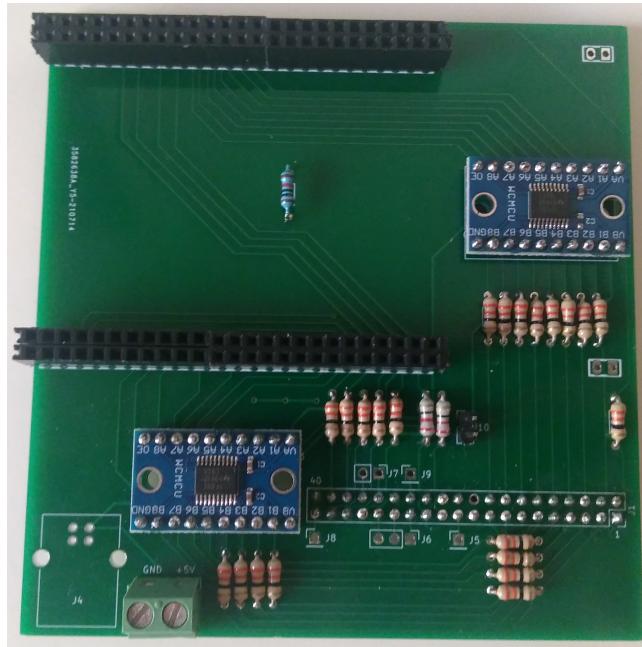


Figura 1: La scheda di interfaccia realizzata, senza scheda STM32F4 montata. I pin del connettore IDE (J1) sono rivolti verso il lato inferiore della scheda; i due moduli blu sono i convertitori TXS0108E.



Figura 2: L'hardware del progetto completamente assemblato: in alto, la scheda mostrata in Figura 1 con la STM32F4 montata e connessa ad un PC con due cavi USB: alimentazione e debugging (nero) e dati (blu). In basso, connesso con un cavo IDE, un hard disk PATA; il PCB marchiato *LGH-IDE-K* è un adattatore passivo dal connettore IDE standard per i dischi da PC fisso a quello utilizzato per i dischi da laptop. L'alimentazione a 5V per l'hard disk proviene da un alimentatore collegato al cavo nero in basso a sinistra, ed è propagata anche ai TXS0108E sulla scheda tramite il cavo bianco-grigio.

3 Architettura software

3.1 Interfaccia IDE

3.1.1 Protocollo di trasporto

La periferica è visibile all'host come un insieme di registri da 8 o 16 bit, che possono essere letti e scritti per impartire comandi e trasferire dati; ciascuno di essi è identificato dai 5 bit che l'host deve assegnare alle linee CS0, CS1, DAO, DA1, DA2 per selezionarlo.

La sequenza di operazioni utilizzata dall'host per eseguire una lettura è la seguente:

- Configurazione dei pin del bus dati in modalità input (con apposite chiamate alla funzione `HAL_GPIO_Init`)
- Selezione del registro da leggere (tramite scrittura dei pin DAO-DA2 e CS0-CS1)
- Scrittura di uno 0 logico sul pin DIOR, per indicare al device di porre sul bus il contenuto del registro
- Lettura del bus: tenendo conto della mappatura precedentemente descritta e delle seguenti definizioni,

```
#define PORTD_BUS_IDR_MASK 0b000000000000000000000000000000001111  
#define PORTE_BUS_IDR_MASK 0b000000000000000000000000000000001111111111110000
```

viene eseguita accedendo direttamente agli IDR (Input Data Register) delle due porte interessate:

```
uint16_t device_reg_content = (uint16_t)((GPIOD->IDR & PORTD_BUS_IDR_MASK)  
| (GPIOE->IDR & PORTE_BUS_IDR_MASK));
```

- Scrittura di un 1 logico sul pin DIOR, per indicare al device di rilasciare il controllo del bus (riportando le linee in condizione di alta impedenza)

I passaggi eseguiti per un'operazione di scrittura, invece, sono:

- Configurazione dei pin del bus dati in modalità output
- Selezione del registro da scrivere (come sopra)
- Scrittura del bus: tenendo conto delle definizioni,

```
#define PORTD_BUS_BSRR_MASK 0b0000000000001111  
#define PORTE_BUS_BSRR_MASK 0b1111111111110000
```

viene eseguita tramite i registri BSRR (Bit Set Reset Register) delle due porte interessate:

```
uint16_t word = /* 16-bit word to be written to the selected device register */;  
GPIOD->BSRR = ((~word & PORTD_BUS_BSRR_MASK) << 16) | (word & PORTD_BUS_BSRR_MASK);  
GPIOE->BSRR = ((~word & PORTE_BUS_BSRR_MASK) << 16) | (word & PORTE_BUS_BSRR_MASK);
```

- Scrittura di uno 0 logico sul pin DIOW, per indicare al device di acquisire il dato presente sul bus
- Scrittura di un 1 logico sul pin DIOW, per indicare al device di terminare il monitoraggio/lettura del bus

Lo standard prevede dei tempi minimi, in nanosecondi, per cui l'host deve mantenere invariato lo stato del canale di comunicazione tra due passaggi, stabilendo la massima frequenza di lavoro e quindi la massima velocità di trasferimento dati. Tali ritardi sono implementati con appositi cicli che eseguono un numero di istruzioni adeguato, calcolato a partire dai tempi specificati nello standard e dalla frequenza di clock dell'MCU; non è stato possibile, in questi casi, utilizzare la funzione `HAL_Delay` messa a disposizione dalle librerie HAL dell'ambiente, poiché ha una risoluzione (1 ms) non sufficientemente fine.

3.1.2 Registri e comandi

I principali registri di una periferica IDE, accessibili come descritto nel paragrafo precedente, sono i seguenti:

- **DATA**: utilizzato per il trasferimento, sequenziale, dei dati memorizzati nello storage persistente del device. È l'unico ad essere realmente a 16 bit: gli altri registri, quando acceduti, utilizzano solo gli 8 bit meno significativi del bus dati.
- **STATUS/COMMAND**: in lettura, restituisce informazioni sullo stato corrente del device; in particolare, un bit di **READY** (1 se la periferica è pronta ad accettare comandi), un bit **BUSY** (la periferica sta effettuando operazioni sullo storage persistente) ed un bit **DRQ** (la periferica è pronta a fornire o ricevere dati, a seconda dell'ultimo comando eseguito). In scrittura, invece, è utilizzato per impartire alla periferica il comando identificato dal valore scritto.
- Registri di indirizzamento settori: 4 in totale, devono essere popolati con le informazioni necessarie per identificare il settore del disco a cui la prossima operazione di lettura o scrittura dovrà accedere. Sono previste due modalità di indirizzamento alternative, **CHS** (Cylinder Head Sector) ed **LBA** (Logical Block Address); anche la modalità è selezionata da un apposito bit nel primo di questi registri.
- **SECTOR COUNT**: Consente di specificare quanti settori, a partire da quello specificato tramite LBA o CHS, dovranno essere interessati dalla prossima operazione di lettura o scrittura dati.

I principali comandi per lavorare con la periferica sono quelli di lettura e scrittura (rispettivamente 0x20 e 0x30). Per eseguire una lettura dal disco, l'host deve, come prima operazione, attendere che il bit **READY** del registro **STATUS/COMMAND** sia 1 (leggendolo, se necessario, ripetutamente); quindi, deve caricare i parametri del comando — in questo caso, l'indirizzo del primo settore da leggere ed il numero di settori richiesto — negli appositi registri, ed infine scrivere il codice del comando (0x20 nel caso di una lettura) nel registro **STATUS/COMMAND**. Non appena il registro viene scritto, il device inizia ad eseguire il comando. Per leggere il primo dei settori richiesti, l'host deve inizialmente attendere che il bit **DRQ** sia 1, e quindi leggere per 256 volte il registro **DATA**: ciascuna lettura restituirà 2 dei 512 bytes rappresentanti il contenuto del settore (come è possibile notare, quindi, la lettura del registro dati non è un'operazione idempotente). L'accesso ai successivi settori richiesti può essere eseguito nello stesso modo, con il solo vincolo di verificare (e se necessario attendere) il bit **DRQ** prima di procedere alla lettura di ogni settore.

Un'operazione di scrittura avviene in maniera analoga: si attende il **READY**, si preparano gli argomenti e si scrive il comando 0x30, quindi per ciascun settore si attende il **DRQ** (che in questo caso segnala che la periferica è pronta a ricevere e memorizzare un settore, ovvero 512 bytes) e si esegue una sequenza di 256 scritture del registro **DATA** contenenti i dati che si desidera scrivere sul disco.

L'ultimo comando indispensabile per usare correttamente la periferica è quello denominato **IDENTIFY DEVICE** (codice 0xEC), che opera in modo simile a quello di lettura, ma non richiede parametri e non restituisce dati provenienti dallo storage vero e proprio ma bensì una struttura dati (specificata in [2]) contenente varie informazioni sul device stesso, tra cui la capacità, rappresentata come numero di settori.

3.1.3 API

Il codice che gestisce l'interfaccia PATA/IDE è contenuto nel file *ide.c*, che espone nell'header *ide.h* le seguenti funzioni pubbliche:

```
// esegue un reset del device pilotando in modo appropriato l'apposito pin
void ide_init();

// ritorna il numero di settori del device
// (moltiplicando questo valore per 512 si ottiene la capacità in bytes del disco)
int ide_get_num_sectors();
```

```

// predisponde un'operazione di lettura di nsect settori a partire da lba
void ide_begin_read_sectors(uint32_t lba, uint16_t nsect);
// legge un settore salvando il risultato (512 bytes) in buf;
// dopo aver chiamato ide_begin_read, deve essere chiamata esattamente nsect volte
void ide_read_next_sector(uint8_t* buf);
// predisponde un'operazione di scrittura di nsect settori a partire da lba
void ide_begin_write_sectors(uint32_t lba, uint16_t nsect);
// scrive un settore copiando 512 bytes da buf;
// dopo aver chiamato ide_begin_write, deve essere chiamata esattamente nsect volte
void ide_write_next_sector(uint8_t* buf);

```

3.2 Interfaccia USB

Nelle fasi iniziali del progetto, per la gestione dell’interfaccia USB si è utilizzata la libreria fornita da STMicroelectronics appositamente per lo sviluppo di periferiche USB su microcontrollori nell’ambiente *STM32Cube* [7]. Al suo interno sono implementati sia il driver per la gestione a basso livello del controller che il protocollo di varie classi di dispositivo USB; ciò consente di sviluppare facilmente dispositivi appartenenti a tali classi implementando apposite funzioni che la libreria chiamerà in corrispondenza delle richieste effettuate dall’host USB (generalmente un PC).

In particolare, per un dispositivo di archiviazione (classe USB MSC, la cui specifica ufficiale è definita in [4] e [3]), le funzioni da implementare sono le seguenti⁵:

```

// restituisce il numero di dischi logici presenti in questo device
int8_t STORAGE_GetMaxLun(void);
// inizializza un disco
int8_t STORAGE_Init(uint8_t lun);
// popola n_blks e blk_size, rispettivamente, con il numero di blocchi
// di cui è composto il disco e la dimensione di un blocco
int8_t STORAGE_GetCapacity(uint8_t lun, uint32_t *n_blks, uint16_t *blk_size);
// restituiscono informazioni sullo stato del disco
int8_t STORAGE_IsReady(uint8_t lun);
int8_t STORAGE_IsWriteProtected(uint8_t lun);
// scrive/legge n_blks blocchi a partire da lba, copiando da/in buf
int8_t STORAGE_Read(uint8_t lun, uint8_t *buf, uint32_t lba, uint16_t n_blks);
int8_t STORAGE_Write(uint8_t lun, uint8_t *buf, uint32_t lba, uint16_t n_blks);

```

Nella prima versione funzionante del progetto, quindi, le funzioni di interfacciamento con il disco (descritte in Sezione 3.1.3) venivano chiamate direttamente dalle implementazioni delle funzioni appena elencate. Ciò, tuttavia, non è ottimale dal punto di vista delle prestazioni: in un’operazione di lettura, ad esempio, il trasferimento dei dati verso l’host USB inizia solo dopo che la funzione `STORAGE.Read` ha terminato l’esecuzione, e quindi solo dopo che la lettura da disco è stata completata e tutti gli N blocchi richiesti sono stati caricati nella memoria del microcontrollore. Ne consegue che in un dato istante una sola tra le interfacce USB e IDE sta trasferendo dati, mentre l’altra rimane in attesa; per migliorare le prestazioni del sistema si è quindi presentata la necessità di poter avviare il trasferimento sull’interfaccia USB prima che quello sulla IDE sia terminato (o viceversa in un’operazione di scrittura).

A livello hardware, affinché ciò sia possibile, è necessario che il core USB possa trasferire dati autonomamente senza tenere impegnata la CPU per tutta la durata del trasferimento, in modo che questa possa contemporaneamente pilotare i GPIO dell’interfaccia IDE come descritto nei paragrafi precedenti. La risposta affermativa a questo requisito è fornita nel capitolo 34 del manuale del microcontrollore [8]. Il core USB FS non è accessibile tramite DMA controller, ma ha un buffer interno

⁵Nel codice che segue, LUN (Logical Unit Number) indica su quale dei dischi eseguire l’operazione: un dispositivo USB MSC può infatti averne molteplici, ad esempio se è un lettore di memory card con più slot

di trasmissione e ricezione: per trasmettere verso l'host, quindi, la CPU riempie il buffer di trasmissione (operazione molto rapida, limitata solo dal clock della CPU stessa), e mentre il core USB lo consuma alla effettiva velocità di trasmissione la CPU può svolgere altri compiti, ricevendo appositi interrupt quando è necessario alimentare nuovamente il buffer. Analogamente, in ricezione i dati vengono memorizzati nel buffer senza intervento della CPU, che può recuperarli “istantaneamente” quando necessario.

La libreria di ST usata inizialmente, tuttavia, non consente di sfruttare questa caratteristica poiché le funzioni `STORAGE_Read` e `STORAGE_Write` devono trasferire l'intera quantità di dati specificata negli argomenti (il valore di ritorno è utilizzato solo per segnalare l'esito positivo o negativo dell'operazione), e vengono chiamate come parte del flusso di esecuzione della ISR che gestisce gli interrupt dell'interfaccia USB. Si è quindi adottata al suo posto la libreria open source TinyUSB [5], che svolge lo stesso compito di gestione dell'interfaccia USB per varie classi di dispositivi offrendo inoltre una maggiore flessibilità nella gestione delle funzioni callback, che per una periferica di archiviazione di massa sono le seguenti (implementate nel file `usb.c`):

```
// restituisce il numero di dischi logici presenti in questo device
uint8_t tud_msc_get_maxlun_cb(void);

// restituiscono informazioni su stato e capacità del disco
bool tud_msc_test_unit_ready_cb(uint8_t lun);

void tud_msc_capacity_cb(uint8_t lun, uint32_t* block_count, uint16_t* block_size);

// scrive/legge fino ad un massimo di bufsize bytes a partire da lba,
// ritornando il numero di bytes trasferiti
// (il numero di settori può essere calcolato come bufsize/512)
int32_t tud_msc_read10_cb(uint8_t lun, uint32_t lba, uint32_t offset,
                           void* buffer, uint32_t bufsize);

int32_t tud_msc_write10_cb(uint8_t lun, uint32_t lba, uint32_t offset,
                           uint8_t* buffer, uint32_t bufsize);
```

Come è possibile notare, le interfacce di programmazione fornite dalle due librerie sono molto simili. Vi sono però alcune importanti differenze: in primo luogo, con TinyUSB, una chiamata ad una delle funzioni di trasferimento dati non deve necessariamente completare l'intera operazione richiesta dall'host, ma può trasferire un numero inferiore di bytes, informando la libreria del progresso dell'operazione tramite il valore di ritorno; questa si occuperà di ripetere la chiamata (aggiornando adeguatamente i parametri `lba` ed `offset`) fino al completamento. Inoltre, contrariamente alla libreria ST, TinyUSB non lavora interamente all'interno della ISR che gestisce gli interrupt del controller USB, ma utilizza una coda di eventi per tenere traccia degli interrupt ricevuti e processarli da una funzione di “main loop” che deve essere chiamata periodicamente; è da essa che vengono poi chiamate le varie funzioni callback sopra mostrate.

Tali caratteristiche, unitamente al sistema operativo FreeRTOS, hanno reso possibile la parallelizzazione dei trasferimenti di dati via IDE e via USB, i cui dettagli implementativi vengono presentati nel prossimo paragrafo.

3.3 Gestione in multitasking delle due interfacce

Una volta introdotta la libreria TinyUSB il software, inizialmente “bare-metal”, è stato convertito in un progetto basato su FreeRTOS al fine di avere a disposizione delle primitive di concorrenza e sincronizzazione pronte all'uso. Per realizzare il parallelismo di trasferimento dati sulle due interfacce vengono utilizzati due task. Uno è dedicato a TinyUSB ed esegue semplicemente il “main loop” della libreria, che si occupa di chiamare i callback di trasferimento dati precedentemente illustrati; l'altro processa invece le richieste di lettura e scrittura generate dalle implementazioni di tali callback.

Il sistema ha 3 possibili stati: `READY`, `READING` e `WRITING`; sono inoltre parte del suo stato una “cache” di dati di capacità massima pari a 32 settori (16KiB), un intero che tiene traccia della sua dimensione attuale e l'informazione sull'LBA del disco corrispondente al primo settore presente nella cache (tutti i dati hanno significato nel contesto dell'operazione attualmente in corso e non sono quindi validi nello stato `READY`).

Il codice che gestisce la macchina a stati è contenuto nel file *ide_async.c*, che espone nell'header *ide_async.h* le seguenti funzioni pubbliche:

```
// lettura/scrittura dei settori specificati
int32_t ide_async_read(uint32_t lba, uint32_t offset, uint8_t* buf, uint32_t bufsz);
int32_t ide_async_write(uint32_t lba, uint32_t offset, uint8_t* buf, uint32_t bufsz);

// inizializza lo stato interno ed il device IDE
void ide_async_init();
// fa avanzare l'eventuale operazione in corso
void ide_async_main_loop_step();
```

ide_async_read e *ide_async_write* vengono eseguite all'interno del task dedicato all'interfaccia USB (poichè chiamate, come descritto di seguito, dai callback della libreria TinyUSB), mentre *ide_async_init* (chiamata una volta) e *ide_async_main_loop_step* (chiamata in un loop infinito) costituiscono il task di gestione dell'interfaccia IDE. Dove necessario, gli accessi alle variabili di stato sopra descritte vengono sincronizzati con un apposito mutex.

3.4 Lettura

Un'operazione di lettura ha inizio con una richiesta inviata dal PC host, che viene gestita da TinyUSB con la chiamata all'apposito callback (*tud_msc_read10_cb*); la sua implementazione consiste semplicemente in una chiamata ad *ide_async_read* con gli stessi parametri. Tale funzione verifica lo stato del sistema; se è *READY*, lo cambia a *READING* dopo aver salvato nello stato l'LBA iniziale richiesto e ritorna il valore 0, segnalando così alla libreria che, per il momento, i dati richiesti non sono pronti. Prima di ritornare, utilizza un semaforo per segnalare all'altro task (che è in attesa nella funzione *ide_async_main_loop_step*) che è pronta una richiesta da gestire. Il task IDE avvia quindi la lettura di 32 (dimensione della cache) settori a partire dall'LBA richiesto tramite la funzione *ide_begin_read_sectors* (introdotta in 3.1.3), e popola la cache un settore alla volta chiamando *ide_read_next_sector*. Nel task USB, invece, non essendo stati trasferiti tutti i dati, la libreria chiamerà nuovamente il callback (in questo caso, con gli stessi argomenti poichè l'invocazione precedente non ha trasferito alcun dato); essendo lo stato *READING*, *ide_async_read* verificherà che l'LBA richiesto sia nel range dei settori (contigui) presenti in cache e trasferirà i settori pronti. La libreria, infine, ripeterà la chiamata con l'LBA aggiornato in base al numero di settori trasferiti dall'ultima invocazione, fino al completamento del trasferimento di quanto richiesto dall'host.

3.5 Scrittura

Analogamente, una richiesta di scrittura su disco (da parte del PC host) viene gestita da TinyUSB con il callback *tud_msc_write10_cb*, la cui implementazione è una chiamata ad *ide_async_write*. Questa funzione, se lo stato è *READY*, copia i dati da scrivere nella cache e, cambiando lo stato in *WRITING* e notificando l'apposito semaforo, segnala al task che gestisce l'interfaccia IDE di eseguire la scrittura; quindi, ritorna il numero di byte copiati nella cache (la cui capacità è sufficiente a contenere qualsiasi richiesta di scrittura eseguita dall'host, dato che queste non sono mai superiori a 32 settori). L'esecuzione di *ide_async_write* è quindi quasi istantanea, dovendo effettuare solo una copia in memoria e non la scrittura su disco vera e propria; appena essa ritorna e mentre il task IDE sta eseguendo la scrittura, quindi, il task USB può ricevere i dati della richiesta successiva.

4 Analisi delle prestazioni

L'indicatore di prestazioni più significativo per questo tipo di applicazione è, chiaramente, la massima velocità di trasferimento dati tra hard disk e PC che è in grado di sostenere. Il limite superiore è imposto dal controller USB a bordo del microcontrollore utilizzato, che implementa la versione 1.1 (anche

denominata *Full-Speed*) dello standard, la cui massima velocità teorica è di 1.5MB/s. Sul microcontrollore è presente anche un controller USB *High-Speed*, che non è stato possibile utilizzare poiché richiede un PHY (il dispositivo che implementa il layer fisico dell’interfaccia USB) esterno da connettere al microcontrollore tramite molteplici pin, alcuni dei quali sulla scheda *STM32F4DISCOVERY* sono condivisi con altre periferiche (come è possibile verificare dalla tabella 7 di [10]: tutti i pin con *alternate function OTG_HS_ULPI* sarebbero necessari).

La massima velocità di trasferimento ottenuta, sia in lettura che in scrittura, è di circa 1.1MB/s. I test sono stati eseguiti su un PC Linux, con il dispositivo connesso direttamente ad una porta USB (senza hub) ed agendo direttamente sul device a blocchi non montato (`/dev/sdx`) con il tool `dd` per non avere overhead da parte del filesystem. Nelle fasi iniziali in cui non era implementata la gestione concorrente delle interfacce, la massima velocità raggiunta (nelle stesse condizioni e con lo stesso hard disk) era di soli 350kB/s.

La potenza dissipata dal microcontrollore è di circa 0.23W, valore ottenuto moltiplicando la corrente misurata durante un trasferimento di dati, 70mA, per la tensione di lavoro dell’MCU, 3.3V. La misurazione della corrente è stata ottenuta, come indicato nel manuale [10], inserendo un amperometro al posto del jumper JP1 della scheda *STM32F4DISCOVERY*. Trattandosi di un dispositivo non pensato per essere alimentato a batteria e che viene sempre utilizzato unitamente a dispositivi dal consumo generalmente molto superiore (tutti gli hard disk utilizzati per le prove riportano nei dati di targa consumi di corrente nell’ordine delle centinaia di mA), non si è ritenuto rilevante integrare l’utilizzo delle modalità a basso consumo del microcontrollore.

Riferimenti bibliografici

- [1] American National Standard for Information Technology. ATA/ATAPI Parallel Transport.
https://www.t13.org/system/files/Project%20Drafts/2013/d1698r4c-1698D%3A%20AT%20Attachment-8%20-%20Parallel%20Transport%20%28ATA8-APT%29_2.pdf, 2013. Accessibile previa registrazione sul sito t13.org.
- [2] American National Standard for Information Technology. ATA Command Set.
https://www.t13.org/system/files/Project%20Drafts/2017/di529r20-ATA/ATAPI%20Command%20Set%20-%204_2.pdf, 2017. Accessibile previa registrazione sul sito t13.org.
- [3] USB Implementers Forum. Universal Serial Bus Mass Storage Class Bulk-Only Transport.
https://usb.org/sites/default/files/usbmassbulk_10.pdf, 1999.
- [4] USB Implementers Forum. Universal Serial Bus Mass Storage Class Specification Overview.
https://usb.org/sites/default/files/Mass_Storage_Specification_Overview_v1.4_2-19-2010.pdf, 2010.
- [5] Thach Ha. An open source cross-platform USB stack for embedded system.
<https://github.com/hathach/tinyusb>.
- [6] Texas Instruments. TXS0108E 8-Bit Bi-directional, Level-Shifting, Voltage Translator for Open-Drain and Push-Pull Applications.
https://www.ti.com/lit/ds/symlink/txs0108e.pdf?ts=1622528719627&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FTXS0108E.
- [7] STMicroelectronics. Middleware USB Device MCU Component.
https://github.com/STMicroelectronics/stm32_mw_usb_device.
- [8] STMicroelectronics. RM0090 Reference manual STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs.
https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.
- [9] STMicroelectronics. STM32F4DISCOVERY Schematic.
https://www.st.com/content/ccc/resource/technical/layouts_and_diagrams/schematic_pack/group1/83/aa/1d/e9/d8/3e/40/a6/MB997-F407VGT6-C01_Schematic/files/MB997-F407VGT6-C01_Schematic.pdf/jcr:content/translations/en.MB997-F407VGT6-C01_Schematic.pdf.
- [10] STMicroelectronics. UM1472 User manual Discovery kit with STM32F407VG MCU.
https://www.st.com/resource/en/user_manual/dm00039084-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf.