POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Automation and control engineering

AUTOMOTIVE DATA LOGGER: STUDIO E SVILUPPO DI UN DISPOSITIVO DI REGISTRAZIONE DATI AD ALTE PRESTAZIONI E A BASSO COSTO

Relatore: Prof. Ivan Rech

Correlatore: Ivan Labanca

Tesi di laurea di:

Filippo Marconcini

Matr.879093

Anno accademico 2017/2018

Ringraziamenti

Arrivando alla fine del tuo percorso universitario suscita emozioni indescrivibili fermarsi per un istante e guardare al passato: dal giorno del test d'ammissione, in cui la curiosità per questo mondo allora sconosciuto era travolgente e faceva sbiadire la preoccupazione, sono cambiate così tante cose che non si riescono neppure a contare. Il tempo, che sembrava scorrere troppo lentamente nonostante la prima laurea, è arrivato.
È difficile immaginare ciò che non si conosce: sicuramente le nozioni di cui mi sono arricchito, che pur essendo essenziali per una buona formazione costituiscono solo una parte del bagaglio culturale di una persona. non possono essere il solo aspetto.

Vorrei ringraziare i professori che con la loro passione mi hanno contagiato nella voglia di non limitarmi solo a ciò che è necessario, ma di guardare sempre a nuovi orizzonti.
In particolare ringrazio il professor Ivan Rech, relatore di questa tesi, per l'entusiasmo con cui mi ha proposto l'argomento , che mi ha spronato nella sua realizzazione, e la libertà esecutiva concessami, che mi ha permesso di crescere più di quanto avessi potuto immaginare.
Ringrazio Ivan Labanca, correlatore, per avermi saputo dare i consigli giusti quando il lavoro sembrava arenato.

Ringrazio Ivan, per la sua amicizia gentile e duratura, che mi fa capire che non importa tanto quando si raggiunge un obiettivo ma come.
Ringrazio Katherine, che mi ha anticipato (ma non di molto) e mi ha supportato anche quando il mio carattere non è stato dei migliori.
Ringrazio chi ha avuto il coraggio di leggere questa tesi pur essendo consapevole di non sapere nulla riguardo all'argomento: è stato indispensabile per cercare di essere il più chiaro possibile

Vorrei infine ringraziare i miei genitori per i loro saggi consigli e la loro capacità di ascoltarmi. Siete sempre stati al mio fianco.

Per ultimi ma non meno importanti, i miei amici. Ci siamo sempre sostenuti a vicenda, nella buona e nella cattiva sorte, sia durante le fatiche e lo sconforto che hanno caratterizzato il nostro percorso nei momenti di gioia e soddisfazione al raggiungimento del traguardo.

Un sentito grazie a tutti!

Filippo Marconcini

Sintesi

Questa tesi riguarda l'implementazione di un data logger per ambito automotive, cioè un dispositivo in grado di memorizzare in modo strutturato i dati prodotti dai sensori presenti in un veicolo. L'obiettivo che si vuole raggiungere è ottenere un modello ad alte prestazioni ma a costo ridotto, abbastanza flessibile da poter essere utilizzato in vari contesti.

Nel trattare tale sperimentazione si procede inizialmente ad introdurre i data logger, prima con alcuni cenni storici e poi riferendosi all'attuale stato dell'arte: alcuni modelli in commercio sono analizzati, evidenziandone le principali caratteristiche.

Con tali riferimenti si espone il processo di scelta dell'hardware necessario: si approfondisce la necessità di una soluzione basata su microcontrollore invece che su microprocessore. Valutando le caratteristiche necessarie si arriva alla scelta di un modello adatto allo scopo e si individua una development board per lo sviluppo.

Vengono quindi discusse separatamente e nel dettaglio le varie parti dell'implementazione, tra cui la gestione dei messaggi in arrivo dal bus CAN, l'elaborazione e l'organizzazione dei dati, la memorizzazione sulla memoria di massa tramite Universal Serial bus (USB OTG) e la gestione del file system. Per ciascuna di esse si presenta una realizzazione base per poi discuterne le successive modifiche ed ottimizzazioni. Vengono esposti i risultati ottenuti e le ipotesi di funzionamento.

Infine viene presentata la struttura del sistema completo, proponendo applicativi specifici per l'uso in laboratorio, per la registrazione continua di dati (nel motorsport e/o nella guida comune) e per la registrazione dei dati di incidente.

Indice dei contenuti

CAPITOLO 1 INTRODUZIONE **1**

1.1 DATA LOGGING 1

1.1.1 Definizioni e descrizione generale *1*

1.1.2 Storia *2*

1.2 SETTORE *AUTOMOTIVE* 4

1.2.1 Storia *4*

1.2.2 Regolamentazione *6*

1.2.3 Comunicazione *7*

1.3 STATO DELL'ARTE 8

1.3.1 EVO4S *8*

1.3.2 CANSTICK1C *9*

1.3.3 EVO5 .... *10*

1.3.4 CL2000 .... *11*

1.3.5 Confronto *11*

CAPITOLO 2 IMPLEMENTAZIONE **13**

2.1 SCELTA DEL MICROCONTROLLORE 13

2.1.1 Caratteristiche necessarie *14*

2.1.2 Inadeguatezza del microprocessore *14*

2.1.3 Scelta del modello *16*

2.2 MEMORIZZAZIONE DEI DATI 18

2.2.1 USB OTG *19*

2.2.2 File System *22*

2.2.3 Buffer *26*

2.3 STRUTTURA DEI *FILE* 34

2.3.1 Organizzazione dei dati *34*

2.3.2 Tipologie di file *35*

2.3.3 Ridondanza e integrità dei dati *37*

CAPITOLO 3 CONCLUSIONI......**41**

3.1 APPLICAZIONI......41

3.1.1 Strumento da laboratorio......*41*

3.1.2 Data logger per MotorSport......*42*

3.1.3 Registratore a ciclo continuo......*43*

3.1.4 Event Data Recorder......*43*

3.2 SVILUPPI FUTURI......45

3.2.1 Trasmissione dei dati......*45*

3.2.2 Flussi Video......*47*

APPENDICE A PROTOCOLLI DI TRASMISSIONE......**49**

A.1 CAN......49

A.1.1 Architettura fisica......*50*

A.1.2 Trasmissione dati e arbitrato......*52*

A.1.3 Tipologie di frame......*53*

A.1.4 Temporizzazione e bit stuffing......*56*

A.2 USB......58

A.2.1 Architettura fisica......*59*

A.2.2 Trasmissione dei dati ed endpoint......*59*

A.2.3 Tipologie trasferimento e frame......*61*

A.2.4 OTG......*62*

APPENDICE B *FILE SYSTEM* FAT......**65**

B.1 ORGANIZZAZIONE......66

B.2 UTILIZZO......69

Indice delle figure

Figura 1.1: Costo percentuale dell'elettronica in un'automobile....5
Figura 1.2: Evoluzione nel tempo degli standard di comunicazione in ambito *automotive*.....7
Figura 1.3: EVO4S - Dettagli della piedinatura....9
Figura 1.4: CANSTICK1C - Dettagli della piedinatura....10
Figura 1.5: EVO 5 - Disegno tecnico....10
Figura 2.1: Panoramica delle caratteristiche della serie PIC32MZ....17
Figura 2.2: Struttura del modulo USB – PIC32MZEF....19
Figura 2.3: Struttura della memoria del circuito integrato TC58NVG0S3HTA00....23
Figura 2.4: Latenza in scrittura, generica....25
Figura 2.5: Latenza in scrittura, memoria deframmentata....25
Figura 2.6: Latenza in scrittura, *file* preallocato....26
Figura 2.7: Utilizzo di un *buffer* LIFO....27
Figura 2.8: Utilizzo di un *buffer* FIFO....28
Figura 2.9: Utilizzo di un *buffer* doppio....30
Figura 2.10: Utilizzo di un *buffer* circolare....31
Figura 2.11: Mappatura della memoria - PIC32MZEF....33
Figura 2.12: Schema RAID 4....39
Figura 3.1: Età media in anni degli autoveicoli in Unione Europea....45
Figura A.1.1: *High Speed* CAN - topologia....50
Figura A.1.2: *High Speed* CAN - esempio di trasmissione....51
Figura A.1.3: *Low Speed* CAN - topologia....51
Figura A.1.4: High Speed CAN - esempio di trasmissione....52
Figura A.1.5: Esempio di divisione temporale del bit in 10 quanti....57
Figura A.1.6: Esempio di messaggio prima e dopo il bit *stuffing*....57
Figura A.2.1: Transazioni USB, IN ed OUT....60
Figura A.2.2: Esempio di un trasferimento di controllo....60
Figura B.2.1: Riepilogo grafico della struttura di una memoria con *file system* FAT32....71

Indice delle tabelle

Tabella 1.1: Principali standard di comunicazione cablata....4
Tabella 1.2: Confronto fra vari *data logger* commerciali....12
Tabella 2.1 Latenza di vari sistemi operativi....15
Tabella 2.2: Velocità di scrittura su memoria flash....24
Tabella 2.3: dimensione di alcuni tipi di dato, in formato binario e testuale....35
Tabella 2.4: Prestazioni di conversione di dati in testo....36
Tabella A.1: Esempio di arbitrato a tre contendenti....53
Tabella A.2: Struttura del frame di base....54
Tabella A.3: Struttura del frame esteso....55
Tabella A.4: Velocità del *bus* CAN 2.0A....58
Tabella A.5: Comparazione trasferimenti USB *High-Speed*....62
Tabella A.6: Velocità di trasmissione delle versioni USB....63
Tabella B.1: Confronto dei *file system* FAT....65
Tabella B.2: Campi di interesse - Boot Sector....66
Tabella B.3: Esempio di *file* Allocation Table....67
Tabella B.4: Struttura di una voce della radice degli indici....68

CAPITOLO 1
INTRODUZIONE

In questo capitolo introduttivo viene fornito il contesto in cui si colloca il lavoro svolto. In particolare nel primo paragrafo il campo del *data logging* è presentato attraverso un percorso cronologico evidenziando le tappe più significative. Nel secondo paragrafo sono contestualizzati lo sviluppo e le crescenti necessità del settore *automotive*. A conclusione del capitolo è mostrato l'attuale stato dell'arte, confrontando le caratteristiche di alcuni modelli in commercio.

1.1 Data logging

1.1.1 Definizioni e descrizione generale

Il *data logging,* o registrazione dei dati, è il processo che si occupa della raccolta e memorizzazione dei dati, che possono così essere elaborati ed analizzati in un secondo momento. Il dispositivo attraverso cui viene effettuato tale processo è chiamato *data logger* ed è spesso indipendente e separato dal resto del sistema: in molti ambiti infatti è importante che continui la sua funzione anche in situazioni anomale come la mancanza di alimentazione.
In generale l'acquisizione dati è un'attività distinta e può essere eseguita da dispositivi specifici, anche se spesso alcuni o tutti i dati raccolti sono memorizzati direttamente dallo stesso dispositivo che li acquisisce. Tipicamente è prevista la possibilità di convertire, filtrare o elaborare i dati. Un caso particolare è la

decimazione, o *downsampling*, che è usata nei contesti dove la quantità di dati generata è più elevata rispetto a ciò che è necessario memorizzare.
I *data logger* possono essere completamente meccanici o elettrici, analogici o digitali. Esistono anche implementazioni puramente *software*, usate nei sitemi informatici, che permettono di analizzare andamenti specifici in un sistema o una rete, di valutarne le prestazioni e di tracciare le interazioni tra utenti, dati e richieste.

1.1.2 Storia

Storicamente il primo *data logger* moderno può essere considerato il registratore grafico: è composto da un pennino che lascia una traccia di inchiostro su un supporto di carta millimetrata mossa a velocità costante nel tempo. Nel tempo si sono sviluppate molte varianti e ormai possono essere meccanici, elettromeccanici o interamente elettronici, possono avere circuiti di retroazione per compensare l'attrito del pennino, possono avere uno stadio di acquisizione digitale come anche più canali (con più pennini). Rimangono tutt'ora in uso in alcuni campi, come nel monitoraggio della temperatura in sistemi di refrigerazione e nel monitoraggio ambientale. Ad esempio i sismografi sono particolari registratori grafici.
I principali pregi sono la loro economicità, la possibilità di funzionare senza alimentazione (con la carica a molla), la praticità di lettura e conservazione dei dati. Tra i difetti ci sono la difficoltà di ottenere registrazioni accurate e la bassa velocità di acquisizione dei dati: per questi motivi sono sempre più spesso sostituiti da supporti di memorizzazione digitali.
È interessante notare come il primo utilizzo nel settore dei trasporti risalga al 1839 quando Charles Babbage[1] propose di inserirlo nei carri ferroviari per registrare dati da usare nella ricostruzione degli incidenti, in quanto le testimonianze differivano spesso tra di loro e dalle evidenze meccaniche. Vennero create carrozze dinamometriche, specializzate nelle misure, ma non entrarono in servizio nell'uso quotidiano bensì per diagnosi della linea e per lo studio di nuovi modelli di locomotori. Furono usate anche per certificare i record di velocità, come ad esempio quello del 3 giugno 1938 della *Mallard*[2].

1 Babbage C. "Passages from the life of a philosopher", Longman, Roberts and Green, 1864, pp. 329–334

2 Googelberg "Classic British Steam Locos",Lulu.com, 2012, ISBN:9781291079739, pp.238

Con l'avvento dell'aeronautica nacque l'esigenza di registrare i dati di volo: i primi prototipi[3] utilizzavano un fascio di luce deviato da uno specchio a seconda dell'ampiezza della grandezza da misurare, che infine impressionava una pellicola fotografica. Questo sistema venne rapidamente sostituito con l'introduzione del nastro magnetico che, sebbene fosse nato per applicazioni audio, poteva registrare qualsiasi tipo di dato elettrico analogico. Per approfondimenti sulla storia della scatola nera si rimanda al [7]. L'aspetto più importante che si vuole evidenziare è la registrazione a ciclo continuo: il nastro è avvolto in un'unica bobina e le due estremità sono giuntate fra di loro. In questo modo vengono memorizzate solamente le informazioni più recenti, sovrascrivendo quelle più datate.

Negli anni '60, grazie all'introduzione dei *computer*, il *data logging* si diffuse nell'industria: nel 1963 venne presentato il primo *computer* IBM specializzato in raccolta dati, denominato IBM7700[4]. Disponeva di 32 canali di acquisizione e poteva trasmettere i dati raccolti a 16 tra stampanti e schermi. Dopo meno di un anno era già stato prodotto un nuovo modello, IBM1800: grazie allo sviluppo dell'elettronica questi sistemi sono diventati sempre più piccoli, economici e potenti. Per molti anni l'unico standard di comunicazione disponibile è stato RS-232 e spesso ogni produttore utilizzava un proprio formato proprietario. Solamente a metà degli anni '80 ci fu un impulso che portò alla definizione e diffusione di alcuni standard, che nel corso del tempo hanno ricevuto delle revisioni e miglioramenti e che sono utilizzati tutt'ora.

La Tabella 1.1 mostra i principali, con un'indicazione degli ambiti prevalenti di utilizzo.

3 Fayer J. "Vols d'essais: Le Centre d'Essais en Vol de 1945 à 1960", E.T.A.I. (Paris), 2001, ISBN 2-7268-8534-9

4 Archivi IBM, https://www.ibm.com/ibm/history/exhibits/dpd50/dpd50_chronology2.html (novembre 2018)

Tabella 1.1: Principali standard di comunicazione cablata

Nome	Anno	Nodi per *bus*	Ambito di utilizzo
RS-232	1960	2	Macchinari industriali, strumenti scientifici, dispositivi di rete
Ethernet	1980	2	Informatica, infrastruttura di rete
I²C	1982	112	Collegamenti tra dispositivi sulla stessa scheda
RS-485	1983	256	Controlli industriali, aeronautica
SPI	1985	-	Collegamenti tra dispositivi sulla stessa scheda
CAN	1986	2031	Automazione industriale, *automotive*, aeronautica, ascensori, apparecchiature mediche
SDI-12	1988	62	Strumentazione meteorologica
USB	1996	2	Informatica, video, memorie di massa
MOST	1998	64	*automotive*
LIN	2002	16	*automotive*, sensoristica

1.2 Settore *automotive*

Nel settore *automotive* i *data logger* hanno avuto una diffusione minore che in altri mezzi di trasporto. Questo è dovuto sicuramente anche dall'aspetto normativo, poichè non sono obbligatori in molti paesi (fra cui l'Italia) mentre lo sono in ambito ferroviario e aeronautico. In questa sezione vengono citati altri dispositivi presenti comunemente nei veicoli, in modo da mostrare come molti dati siano già raccolti e perciò come sia naturale la creazione di un *data logger* per memorizzarli.

1.2.1 Storia

Inizialmente è stata sviluppata la diagnostica a bordo (OBD*, On-Board Diagnostics*) che, tramite dei sensori, permette di individuare le anomalie e quindi mostrare al conducente un segnale attraverso delle spie luminose. All'OBD nel tempo si sono aggiunti molti altri dispositivi elettronici (*Electronic Control Unit,* ECU) tra cui il sistema anti bloccaggio ruote (*Antilock Braking system,* ABS), il sistema controllo trazione (*Acceleration Slip Regulation,* ASR), il sistema controllo stabilità (*Electronic Stability Program,* ESP), l'unità di controllo motore (E*ngine Control*

Unit, ECU, chiamata anche *Engine Control Module,* ECM), il registratore dati di incidente (*Event Data Recorder,* EDR) o *data logger* a ciclo continuo.

I sistemi presentati possono avere la capacità di memorizzare dei dati, anche se tipicamente in forma aggregata e relativi al loro scopo. Questi possono essere ad esempio dei parametri che evolvono nel tempo (a causa di usura, condizioni climatiche ecc.).

Diverse compagnie assicurative propongono polizze in cui è previsto l'utilizzo di scatole nere. Ad esempio in Italia "Il 20,7 per cento delle polizze stipulate nel secondo trimestre 2018 prevede una clausola con effetti di riduzione del premio legata alla presenza della scatola nera"[5]. Questi dispositivi registrano dati come le marce inserite, le velocità, la posizione (tramite GPS), l'accelerazione. I dati vengono quindi aggregati per fornire una valutazione della guida dell'assicurato. In caso di rilevamento di incidente, oltre alla possibilità di memorizzare i dati relativi senza elaborazione, può essere inviata una segnalazione automatica ai servizi di soccorso. Esiste uno standard per l'uniformità dei dati registrati, approvato da IEEE[6] nel 2004 e confermato nel 2010.

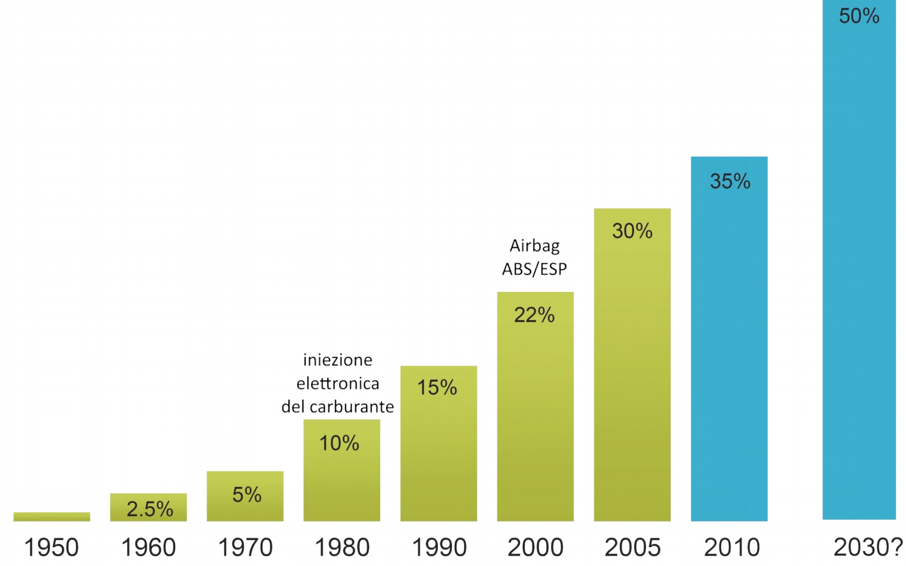


Figura 1.1: Costo percentuale dell'elettronica in un'automobile[7]

Come visibile in Figura 1.1 i sistemi elettronici sono diventati pervasivi all'interno dei veicoli e, affidandosi alle ricerche riguardo alla guida autonoma e alla trazione

5 "Bollettino Statistico IPER: L'andamento dei prezzi effettivi per la garanzia r.c. auto nel terzo trimestre 2018", IVASS, 1 febbraio 2018

6 "IEEE 1616-2004 - IEEE Standard for Motor Vehicle Event Data Recorder (MVEDR)", IEEE, 2005

7 Elaborazione grafica tradotta da presentazione "*automotive* Market and Industry Update", Nelson S., Freescale

elettrica o ibrida, lo saranno ancora di più in un prossimo futuro. In particolare l'acquisizione e l'elaborazione di dati è insita in queste applicazioni di controllo, perciò si pongono nuovi orizzonti anche nell'ambito del *data logging:* sarà indispensabile per migliorare le *performance* e fondamentale nella ricostruzione degli incidenti e per la determinazione delle responsabilità.

1.2.2 Regolamentazione

Le prime normative riguardo all'introduzione di sistemi elettronici a bordo riguardano il tema ambientale: nel 1991 con "*The California Air Resources Board*" (CARB) venne richiesto che tutti i nuovi veicoli fossero dotati di alcune funzionalità base. Infatti una modalità per ridurre drasticamente le emissioni è quella di mantenere il rapporto di miscela (kg aria/kg com*bus*tibile) entro l'intervallo di efficienza ottimale del catalizzatore dei veicoli. Questo obiettivo può essere raggiunto utilizzando una sonda lambda, che è in grado di rilevare la concentrazione di ossigeno all'interno dei gas di scarico, e regolando l'immissione di aria e carburante nella camera di com*bus*tione. Nonostante i componenti degradino naturalmente nel tempo si riesce a garantire una buona efficienza proprio grazie al controllo elettronico. Per una spiegazione più approfondita del funzionamento di questo sistema e della sonda lambda si rimanda alla nota bibliografica [1]. Nel 1994 con l'introduzione della specifica OBD-II, venne imposto un connettore unificato, in modo da facilitare i test di emissione e renderli disponibili su larga scala. Due anni dopo è diventato obbligatorio per tutte le nuove automobili prodotte e acquistate negli Stati Uniti d'America. In Europa questo connettore venne imposto con la *Direttiva 98/69/EC*, la quale definisce i limiti di emissione comunemente detti *Euro 2*. Nel 2017 sono stati sostituiti tutti i precedenti standard, in modo da ottenere una maggiore uniformità[8].

Per quanto riguarda gli EDR esiste l'obbligo di installazione in paesi quali la Svizzera e gli Stati Uniti d'America, mentre in ambito sportivo sono diventati obbligatori nel 1997 per il campionato di *Formula 1* [9].

8 "supplementing Regulation (EC) No 715/2007 of the European Parliament and of the Council on type-approval of motor vehicles with respect to emissions from light passenger and commercial vehicles (Euro 5 and Euro 6) and on access to vehicle repair and maintenance information", Ufficio delle pubblicazioni ufficiali delle Comunità europee, 1 Giugno 2017

9 Wright P. "The Analysis of Accident Data Recorder (ADR) Data in Formula 1", SAE Technical Paper, 2000

1.2.3 Comunicazione

Nonostante i sistemi presentati siano progettati per poter operare indipendentemente dagli altri per una maggiore ro*bus*tezza e sicurezza, esiste la necessità di scambiare dati tra dispositivi diversi: come accennato nel paragrafo precedente, molti standard di comunicazione sono stati realizzati e applicati dalle diverse aziende, incluse le case automobilistiche. Tra i protocolli più usati nel settore per le comunicazioni ci sono CAN, ethernet, LIN, MOST, FlexRay. Quest'ultimo, pur essendo diventato standard ISO nel 2013[10] non viene preso in considerazione in questa tesi perché il consorzio che l'ha sviluppato si è sciolto.
La discussione sui protocolli di trasmissione è di primaria importanza: in un'automobile il cablaggio elettrico è la terza componente in peso[11], dopo telaio e motore. Per ogni tipologia di dato bisognerebbe scegliere il mezzo trasmissivo migliore, valutandone i pregi e difetti. Se più protocolli risultassero adatti andrebbe scelto il più economico o leggero. Ovviamente anche il numero totale di protocolli presente in un veicolo è uno dei parametri da considerare.

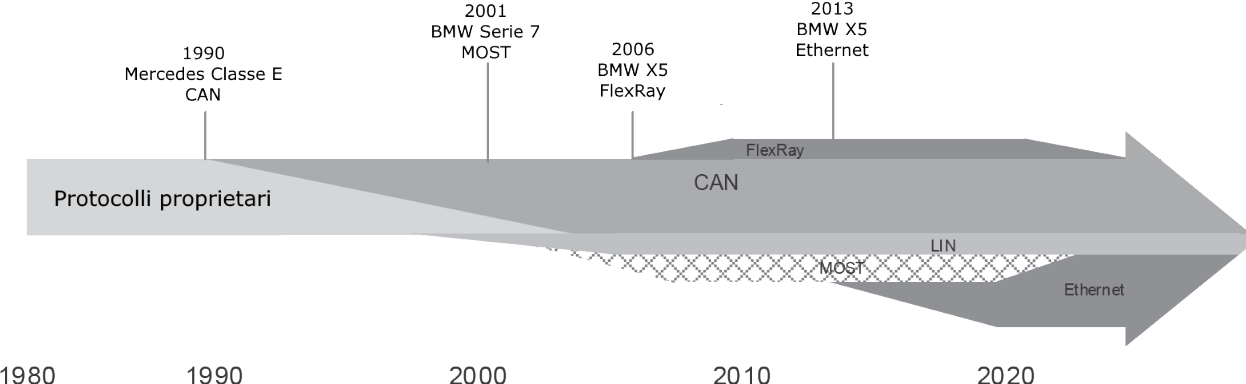


Figura 1.2: Evoluzione nel tempo degli standard di comunicazione in ambito *automotive*[12]

Lo standard CAN ha il vantaggio di essere molto ro*bus*to ai disturbi ed essere *multimaster* e *multicast*: questo vuol dire che qualunque dispositivo può iniziare la trasmissione e che ogni messaggio viene ricevuto da tutti i dispositivi connessi (eventualmente ignorato). Eventuali conflitti sono gestiti con un sistema di arbitraggio ed è possibile aggiungere o rimuovere dispositivi senza modificare la topologia della rete. Può avere un *bitrate* teorico di 1Mbit/s che, pur essendo

10 "Road vehicles - FlexRay communications *system*", ISO 17458
11 "*automotive* ethernet: An Overview", Ixia, 915-3510-01 Rev. A, 2014
12 Immagine tradotta da [2], pag.24

elevato, è il principale limite allo standard. Per questo è stato introdotto nel 2012 lo standard CAN FD[5] che migliora le prestazioni e consente di gestire flussi di dati più consistenti, come quelli delle applicazioni video. È importante sottolineare che può essere garantito che i messaggi rispettino vincoli temporali, come ampiamente descritto in [8]. Per ulteriori informazioni si può consultare il paragrafo A.1 .
Lo standard LIN nasce come alternativa a basso costo per la comunicazione sensoristica non critica ed a basso *bitrate*, dato che permette fino a 20 kbit/s[13].
Infine l'ethernet non è stato sviluppato specificatamente per applicazioni *automotive* però, avendo una vastissima diffusione in ambito informatico, si è esteso soprattutto per le applicazioni video e per gestire comunicazioni fra computer di bordo, per i quali è più naturale implementare tale standard. Il principale difetto è la struttura *end to end*, che richiede un cavo per ogni dispositivo connesso ed eventualmente dispositivi di *switching* o *routing*. *Q*uesto inevitabilmente influisce sul peso, che è un fattore molto importante nell'ambito *automotive*. Nel riferimento bibliografico [2] viene discussa nel dettaglio la potenziale espansione dell'ethernet anche se si riconosce che non arriverà a sostituire le tradizionali reti *automotive* (CAN, LIN) perché sono più economiche, ro*bus*te e sufficienti per gli scopi per i quali sono usate.

1.3 Stato dell'arte

Vengono ora considerate le caratteristiche di alcuni prodotti commerciali recenti: si è cercato di considerare entrambi gli estremi variando per costo e *performance* . Ovviamente fra questi due si presenta un *trade off*.

1.3.1 EVO4S[14]

Il prodotto ha un approccio che tende a separare i vari canali di dati, anche fisicamente. La struttura è modulare: oltre ai dati provenienti dal *bus CAN*, possono essere aggiunti dei moduli per la raccolta di dati specifici (velocità, *beacon*, ingressi analogici). I moduli possono essere anche di *output*: tra di essi c'è il modulo USB, che permette di salvare i dati su un'unità esterna. In ogni caso il *logger* è dotato di

13 "Road vehicles - Local Interconnect Network (LIN)", ISO 17987
14 https://www.aim-sportline.com/en/products/evo4s/ (20 settembre 2018)

4GB di memoria interna. È inoltre dotato di accelerometro e giroscopio interni, entrambi a 3 assi.

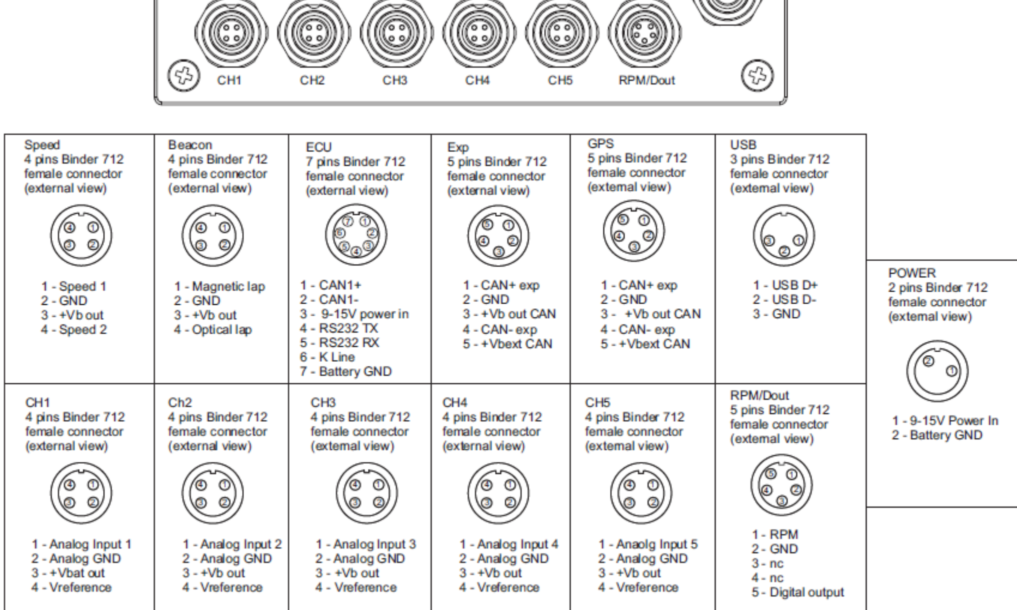


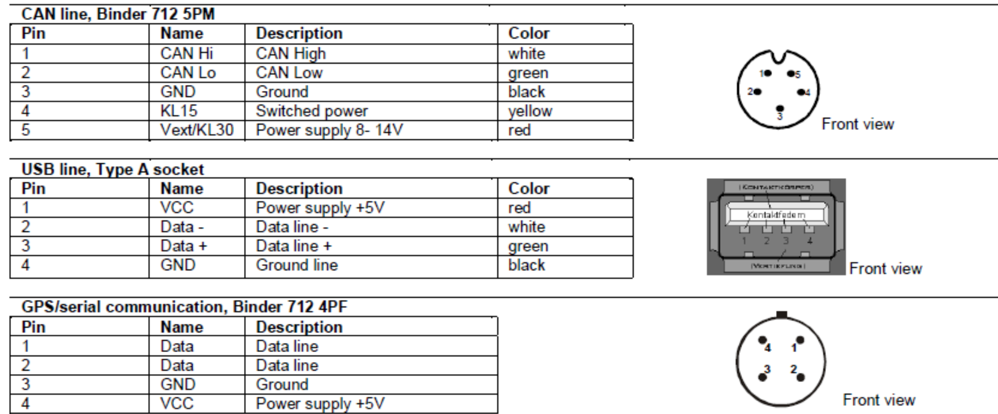
Figura 1.3: EVO4S - Dettagli della piedinatura

1.3.2 CANSTICK1C[15]

Il prodotto si presenta con un'interfaccia fisica molto semplificata: tutti i dati arrivano unicamente dal *bus* CAN, ad eccezione del GPS (opzionale). Anche per questo motivo ha dimensioni inferiori rispetto agli altri *logger*. Non ha memoria interna, perciò per salvare i dati è necessario usare una chiavetta USB oppure, in alternativa, è possibile trasferirli via seriale. È uno dei *data logger* approvati per il campionato Moto 2[16]

15 http://2d-datarecording.com/en/produkte/*hardware*/*logger*/can-memory-2/usb-stick-*logger*/ (23 settembre 2018)

16 “Moto2 approved *logger* & sensor list”, Fédération Internationale de Motocyclisme, 2016



CAN line, Binder 712 5PM

Pin	Name	Description	Color
1	CAN Hi	CAN High	white
2	CAN Lo	CAN Low	green
3	GND	Ground	black
4	KL15	Switched power	yellow
5	Vext/KL30	Power supply 8- 14V	red

Front view

USB line, Type A socket

Pin	Name	Description	Color
1	VCC	Power supply +5V	red
2	Data -	Data line -	white
3	Data +	Data line +	green
4	GND	Ground line	black

Front view

GPS/serial communication, Binder 712 4PF

Pin	Name	Description
1	Data	Data line
2	Data	Data line
3	GND	Ground
4	VCC	Power supply +5V

Front view

Figura 1.4: CANSTICK1C - Dettagli della piedinatura

1.3.3 EVO5[17]

È sostanzialmente molto simile a EVO4S, essendo una sua versione più recente. Ne implementa tutte le funzionalità con alcune aggiunte e miglioramenti. In particolare sono presenti un secondo canale CAN, la connettività WiFi e la possibilità di usare una scheda SD (per il salvataggio dei dati) senza ricorrere a moduli esterni.

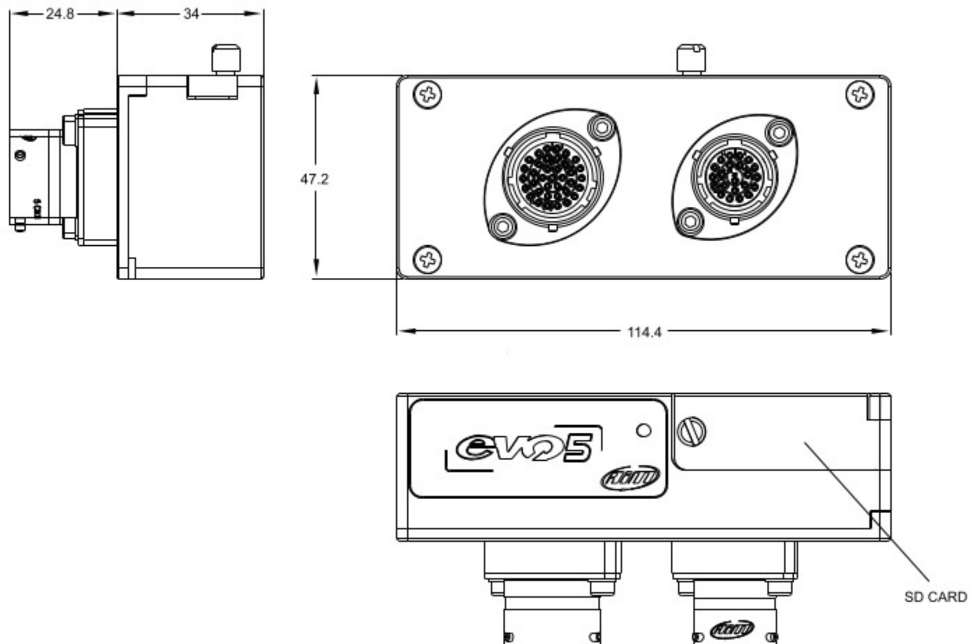


Figura 1.5: EVO 5 - Disegno tecnico

17 https://www.aim-sportline.com/en/products/evo5/ (20 settembre 2018)

1.3.4 CL2000[18]

Questo prodotto ha il *bus* CAN come unica sorgente dati. Si affida ad una scheda SD per il salvataggio dei dati ed ha un consumo ridotto, 0.5W. Implementa sia una divisione ad intervalli regolari (impostabile) dei dati sia una memorizzazione ciclica: quando la memoria risulta piena vengono eliminati i dati meno recenti.

1.3.5 Confronto

Le analisi tecniche di questo paragrafo sono riassunte nella Tabella 1.2.
È stato aggiunto al confronto il prezzo, perché è un utile indicatore di quanto possa essere realizzabile un prototipo nella realtà. Come accennato in precedenza e spiegato meglio in seguito, in questo elaborato si descriverà lo sviluppo del codice, e quindi non si saranno presi in considerazione aspetti meccanici o fisici.
Si nota come solo EVO5 dispone di 2 canali CAN, mentre sembra essere una scelta comune scegliere come granularità del tempo 1 ms, eventualmente decimando i campioni in arrivo. Nei modelli di fascia più alta, EVO4S ed EVO5, sono disponibili degli ingressi analogici, un giroscopio ed un accelerometro incorporati. Queste informazioni diventano prima obiettivo da raggiungere e poi da superare per l’implementazione che vuole essere fornita in questo elaborato.

18 https://www.csselectronics.com/screen/product/can-*bus-logger*-can*logger*2000

	EVO4S	EVO5	CANSTICK1C	CL2000
dimensioni [mm]	130 x 35 x 46.6	114.4 x 47.2 x 58.8	70 x 40 x 12.8	66.7 x 42.7 x 23.5
CAN canali BaudRate [kbit/s] frequenza acquisizione per canale [Hz] terminazione	1 2031 1000 - programmabile	2 2031 1000 - programmabile	1 32-128 1000 800 no	1 8 1000 - no
GPS*	CAN	CAN	Seriale	CAN
ingressi di velocità (digitali)*	2	4	0	0
ingressi analogici * frequenza [kHz] risoluzione [bit]	5 1 12	8 1 12	0	0
giroscopio (3 assi)	1	1	0	0
accelerometro (3 assi)	1	1	0	0
memoria interna [GByte]	4	4	0	0
memoria esterna massima [GByte]	128	128	32	32
prezzo[19]	1274.90€[20]	1680.55€[21]	600€[22]	229€[23]

Tabella 1.2: Confronto fra vari *data logger* commerciali

19 I prezzi sono stati verificati il 21 marzo 2019, a titolo comparativo, riferendosi ai siti indicati nelle seguenti note

20 http://www.euro-racing.it/it/acquisizione-dati/559-evo4s-aim-acquisitore-dati.html

21 http://www.euro-racing.it/it/acquisizione-dati/560-evo-5-aim-acquisitore-dati.html

22 http://www.racingon.it/page1906.htm

23 https://www.csselectronics.com/screen/product/can-*bus-logger*-can*logger*2000

CAPITOLO 2
IMPLEMENTAZIONE

In questo capitolo viene affrontata l'implementazione delle singole parti che compongono il *data logger*. Per ciascuna di esse viene presentata inizialmente una realizzazione generale da cui successivamente viene creata, assumendo le giuste ipotesi, una versione modificata ed ottimizzata per il contesto. Dove significativo sono mostrati i risultati reali delle prove di funzionamento: in questo caso, anche se il microcontrollore scelto ha la possibilità di sfruttare frequenze di *clock* più elevate, sarà supposto un *clock* di 200 Mhz.

2.1 Scelta del microcontrollore

Questa scelta è la più delicata perché su ciò si basa il lavoro svolto: se fatta in modo oculato permette facilità di implementazione ed una maggiore longevità del sistema, garantendo *performance* e caratteristiche adeguate o adeguabili all'evolversi della tecnologia e dei requisiti.
In futuro potrebbe essere necessario scegliere un nuovo microcontrollore ma nonostante ciò i risultati ottenuti sono del tutto generali qualitativamente, pur essendo legati quantitativamente al modello del microcontrollore ed al codice sviluppato. Perciò, dopo aver aggiornato i dati di questo paragrafo con quelli del nuovo microprocessore, potrà essere implementato un nuovo sistema seguendo i passaggi esposti ed aggiornando quantitativamente i risultati.

2.1.1 Caratteristiche necessarie

Dato che non si ha necessità di basarsi su implementazioni precedenti, si vuole selezionare un microcontrollore che abbia nativamente tutte le caratteristiche necessarie. In questo modo si riducono al minimo i componenti esterni e di conseguenza costi e consumi. Inoltre generalmente si riescono ad ottenere prestazioni migliori e definire più facilmente i vincoli temporali, in quanto è tuttto gestito all'interno del microcontrollore stesso.
Si sceglie come requisito necessario la presenza di tutti i moduli specifici per l'applicazione, con caratteristiche uguali o superiori rispetto al migliore dei prodotti presenti nel paragrafo 1.3.
Perciò devono essere presenti almeno:

- un Modulo USB OTG
- due Moduli CAN 2.0B
- un *Timer*
- *Direct Memory Access* (DMA)
- *Analog to Digital Converter* (ADC), 12bit

Come si vedrà più avanti, sarà importante avere RAM capiente e frequenza di *clock* elevata.

2.1.2 Inadeguatezza del microprocessore

I principali vantaggi di un microprocessore sono l'elevata capacità di gestione ed elaborazione dei dati. A ciò si può aggiungere la possibilità di eseguire più programmi contemporaneamente (*multithreading*) e di sviluppare applicazioni grafiche e di interfaccia con l'utente. Questo è agevolato dal sistema operativo e dall'utilizzo di linguaggi di alto livello, pur non essendo prerogative del microprocessore.
Tra gli svantaggi ci sono un consumo energetico maggiore ma soprattutto la necessità di integrare le funzionalità (come il modulo *ADC* o *CAN*) con circuiti esterni, aumentando costi, ingombro e ancora una volta, consumi.
È doveroso dunque domandarsi che cosa sia meglio scegliere per quest'applicazione: a questo si risponderà nel corso dell'elaborato, mostrando come un microcontrollore sia più che sufficiente per la creazione di un *data logger*. Perciò con la scelta di un

microprocessore non ci sarebbero significativi vantaggi, mentre resterebbero da gestire gli svantaggi.
Riguardo all'uso del sistema operativo, pur semplificando il processo di scrittura del codice ed eliminando la necessità di entrare nel dettaglio di ogni singolo aspetto, implica il non poter conoscere o controllare eventuali *bug* ed essere subordinati ad un agente esterno indipendente.
Inoltre non tutti i sistemi operativi sono progettati per essere in tempo reale, cioè in grado di rispettare determinate scadenze temporali. In particolare, per eguagliare lo stato dell'arte attuale bisogna garantire una granularità temporale di 1 ms, che richiede una latenza trascurabile rispetto a tale valore. Si può rilassare tale ipotesi nel caso si conosca la variazione massima della latenza e quest'ultima sia trascurabile: in questo caso si potrebbe sottrarre, se necessario, la latenza media.

Tabella 2.1 Latenza di vari sistemi operativi[24]

Sistema operativo	Latenza media [μs]	Latenza massima [μs]
Windows 10	550	17170
OSX 10.9.5	320	12650
Ubuntu 16.04	100	3030
Scientific Linux 6.6-rt	80	150

Come si può osservare nella Tabella 2.1 i sistemi operativi tradizionali non rispettano tale requisito. Con alcuni sistemi operativi si riesce a limitare questo inconventiente, a spese di una maggiore latenza media[25] e di un maggiore utilizzo della CPU da parte dello *scheduler*. Si evidenzia come nei sistemi operativi i casi peggiori di latenza sono generalmente molto rari ma distanti dai valori medi[26], motivo per cui non si avrebbe un significativo vantaggio a rilassare l'ipotesi precedente. Inoltre, come mostrato negli articoli in nota, questi risultati vengono ottenuti tramite test specifici.

24 Enner F. "A Practical Look at Latency in Robotics : The Importance of Metrics and Operating *systems*", 2016, https://ennerf.github.io/2016/09/20/A-Practical-Look-at-Latency-in-Robotics-The-Importance-of-Metrics-and-Operating-*systems*.html (20 marzo 2019)

25 Koolwal K. "Investigating latency effects of the Linux real-time Preemption Patches (PREEMPT RT) on AMD's GEODE LX Platform", VersaLogic Corporation

26 Cerqueira F. · Brandenburg B. "A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUSRT™", Max Planck Institute for *software systems*

Nei sistemi a microcontrollore invece, data la semplicità e ridotta lunghezza del codice, si può procedere con un approccio più diretto, che consiste nell'analizzare i punti nei quali può essere scatenato un *interrupt,* associare ad ogni istruzione eseguita per il cambio contesto il relativo tempo di esecuzione ed infine ricavare il tempo totale di esecuzione. A questo si deve aggiungere l'eventuale tempo di esecuzione di istruzioni in cui gli *interrupt* sono disabilitati. Ovviamente questa procedura si può verificare attraverso un test di validazione.

2.1.3 Scelta del modello

Sebbene non sia indispensabile, per ridurre il campo di ricerca si è scelta la famiglia di microcontrollori PIC32: questa scelta è stata effettuata per poter sfruttare la catena di sviluppo già in uso (in particolare il *software MPLAB®*) e per la conoscenza pregressa di architettura, registri e la loro gestione. Tuttavia questa scelta non risulta riduttiva data la grande varietà e diversità dei microcontrollori disponibili.

Dal confronto delle caratteristiche rimangono le serie PIC32MX7 e PIC32MZ EF: si nota che la serie PIC32MZ EF ha prestazioni migliori, dall'*ADC* (bit e sample rate) alla velocità *USB*, dalla frequenza di clock massima alla memoria (*RAM* e *Flash*).

Di questa serie, filtrando solo i modelli effettivamente con *USB* e *CAN* (non in tutti sono implementati) il meno costoso risulta PIC32MZ0512EFF064, con 512 kByte di Flash e 128 kByte di RAM. Tuttavia, per una maggiore flessibilità e dato il costo solo leggermente più alto, per la fase di sviluppo è maggiormente indicato il modello PIC32MZ2048EFH064, con 2048 kByte di memoria *Flash* e 512 kByte di RAM. Le altre caratteristiche dei due microcontrollori sono identiche: hanno una frequenza di base di 200 MHz, 64 pin (fattore di forma *QFN*).

Altre caratteristiche di questo modello sono mostrate in Figura 2.1, oltre alla particolarità di garantire il funzionamento tra -40°C e +125°C, *range* abbastanza esteso per un dispositivo elettronico.

High Performance: PIC32MZ Series

PIC32MZ Series

Parameter	PIC32MZ EF	PIC32MZ DA
Speed	252 MHz	200 MHz
Floating Point Unit (FPU)	Yes	–
2D Graphics Processing Unit (GPU)	–	Yes
3-Layer Graphics Controller	–	Yes
DDR2 SDRAM	–	32 MB
Flash	512 KB/1 MB/2 MB	1/2 MB
SRAM	128/256/512 KB	256/640 KB
Boot Flash	160 KB	
DMA	26 ch.	
Ethernet	10/100 Ethernet MAC	
USB	Hi-Speed Device, Host and OTG	
CAN	Dual CAN 2.0B	
ADC	12-bit, 18 MSPS, 48 channel	12-bit, 18 MSPS, 45 channel
Analog Compare	Two AC with 32 programmable voltage references	
TRNG	Yes	
Crypto Engine	AES 256, DES/TDES, SHA1/256, MD-5, AES GCM	
Timers/Compare/Capture	9/9/9	
AEC-Q100	Grade1	Grade 2
RTCC	Yes	
PMP	Yes	
SQI	Yes	
SD/SDIO/eMMC bus interface	–	Yes
DDR2 SDRAM I/F	–	Yes
EBI	Yes	
SPI/I²S	6	
I²C	5	
UART	6	
Pin Count	64, 100, 124, 144	169, 176, 288
Packages	QFN, TQFP, VTLA, LQFP, TFBGA	LFBGA, LQFP

Figura 2.1: Panoramica delle caratteristiche della serie PIC32MZ[27]

Per lo sviluppo del codice e per la validazione dei risultati è stato scelto di non iniziare con una scheda *hardware* personalizzata, in modo da ottenere un *software* stabile e testato prima e separare le problematiche relative allo sviluppo *hardware*. Per questo motivo è stata individuata una scheda di sviluppo, PIC32-EMZ64 Olimex, che dispone sia del microprocessore selezionato sia di una serie di componenti utili

27 "32-bit Microcontroller Families", Microchip, DS30009904T

per lo sviluppo, tra cui un *transceiver* CAN, i circuiti di alimentazione e *clock* e due connettori USB.

2.2 Memorizzazione dei dati

La memorizzazione dei dati riveste un ruolo fondamentale in un *data logger*, in quanto obiettivo della sua esistenza. La scelta del supporto fisico è molto rilevante nella fase di progettazione. Fra i fattori di scelta non vi sono solo elementi elettronici, ma anche persistenza dei dati, resistenza e dimensione meccanica.
Si sceglie perciò una memoria a stato solido poichè ha tipicamente valori di resistenza meccanica molto più elevati rispetto ad *hard disk* o nastri magnetici. Inoltre, date le dimensioni ridotte, è facilitato il posizionamento all'interno del *data logger*, il che può fornire un'ulteriore protezione.
Viene scelto di utilizzare una memoria removibile, in quanto si hanno alcuni vantaggi rispetto ad una memoria integrata.
In particolare la memoria può essere:

- sostituita se danneggiata, lasciando inalterato il *data logger*
- selezionata e modificata dall'utente in base all'applicazione
- rimossa più facilmente rispetto all'intero *data logger*. La rimozione è necessaria qualora si debba poter scaricare ed elaborare i dati indipendentemente dalla posizione del veicolo
- Sostituita "al volo", in modo da poter proseguire nell'acquisizione nel caso sia piena.

Le memorie rimovibili possono avere vari protocolli, i più comuni dei quali sono *Secure Digital* (SD) e USB *Mass Storage*. Pur essendo comunemente più compatto e leggero, lo standard SD ha un principale limite, come descritto nelle relative specifiche al paragrafo 7.2.15:
"As opposed to SD mode, the card cannot guarantee its Speed Class. In SPI mode, host shall treat the card as Class '0' no matter what Class is indicated in SD Status"[28]

Le classi dello standard SD indicano le prestazioni minime da garantire. In particolare la classe '0' non garantisce prestazioni minime: questo significa che, a

28 "SD Specifications Part 1: Physical Layer Specification", Versione 3.00 ,SD Group, 2009

meno di affidarsi ad un dispositivo che gestisca in modo nativo la comunicazione in modalità SD e non tramite protocollo SPI (che invece ha una grande diffusione nei microcontrollori), le memorie SD non sono adatte a questa applicazione, non potendo fare affidamento alle relative velocità di lettura e scrittura indipendentemente dalla classe di appartenenza.
Per questo motivo verranno scelte memorie di massa USB, che dovranno essere almeno *High Speed*, cioè versione 2.0, per poter garantire un'adeguata velocità di trasmissione.
Resta comunque possibile inserire il supporto per memorie SD, anche se si consiglia di restringerne l'utilizzo solamente in ambiti in cui il tempismo non è critico.

2.2.1 USB OTG

Prima di proseguire la lettura di questo paragrafo si consiglia di consultare l'Appendice A.2 se non si conosce in dettaglio il funzionamento di un *bus* USB.
Un dispositivo USB può disporre al massimo di 32 *endpoint* (16 in ingresso e 16 in uscita): il microcontrollore scelto ne implementa 16 (8 in ingresso e 8 in uscita). L'*endpoint* '0' è riservato in entrambe le direzioni per le operazioni di controllo.

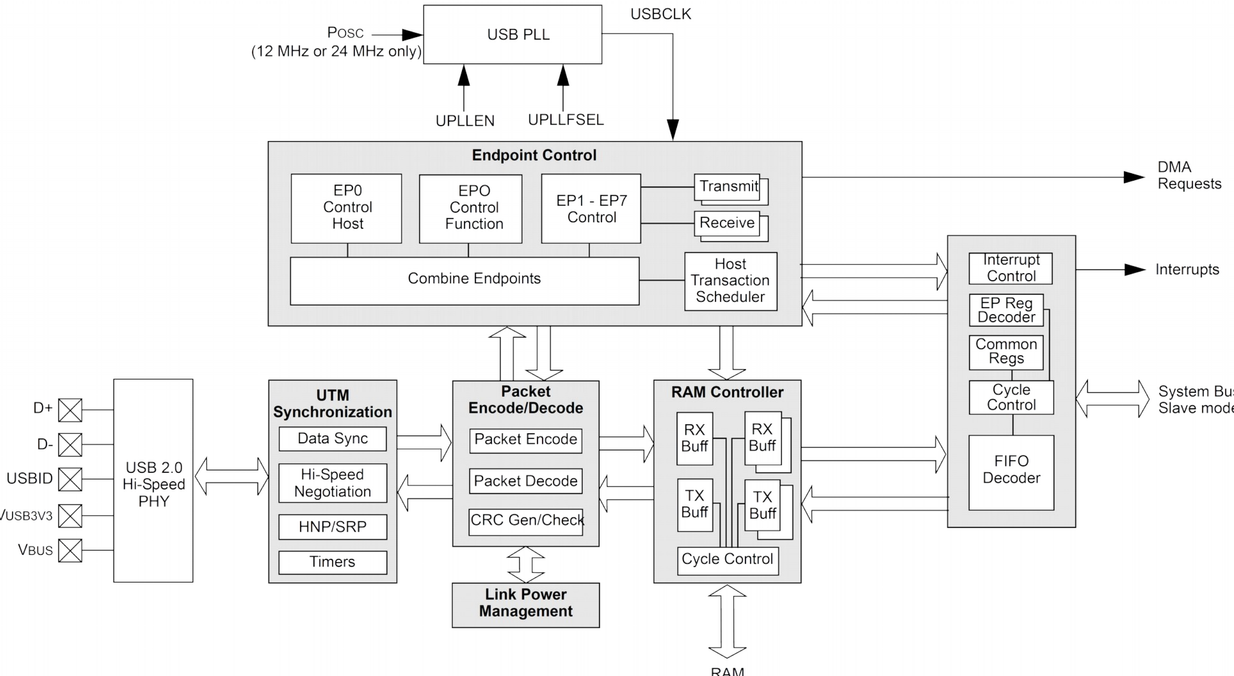


Figura 2.2: Struttura del modulo USB – PIC32MZEF

Per effettuare un trasferimento in uscita è necessario precedentemente riempire il *buffer* associato all'*endpoint* (che dev'essere precedentemente configurato); viceversa un trasferimento in ingresso viene ricevuto alla fine della ricezione del messaggio e letto dal rispettivo *buffer*. Il codice seguente realizza un trasferimento in uscita, cioè dal *master* (in questo caso il microcontrollore) allo *slav*e (la memoria di massa).

```
 1. void USBHS_EndpointFIFOLoad_Default
 2. (
 3.   USBHS_MODULE_ID index , //indirizzo fisico associato al modulo USB
 4.   uint8_t endpoint , //numero dell'endpoint
 5.   void * source , //indirizzo dei dati da trasmettere
 6.   size_t nBytes  //numero di Byte da trasmettere
 7. )
 8. {
 9.   /* assegna alla variabile l'indirizzo del modulo, in modo da associare
    la relativa struttura e poter accedere ai vari campi più chiaramente */
10.   volatile usbhs_registers_t * usbhs = (usbhs_registers_t *)(index);
11.   volatile uint8_t * endpointFIFO;
12.   size_t i;

13.   //Ottiene il puntatore al buffer FIFO relativo all'endpoint
14.   endpointFIFO = (uint8_t *)(&usbhs->FIFO[endpoint]);

15.   //carica i dati nel buffer
16.   for(i = 0; i < nBytes; i ++)
17.   {
18.     *endpointFIFO = *((uint8_t *)(source) + i);
19.   }

20.    /*Imposta il bit TXPKTRDY, in modo da iniziare la trasmissione. La
    posizione del bit è diversa per l'endpoint 0
    */
21.   if(endpoint == 0)
22.   {
23.     usbhs->EPCSR[0].CSR0L_DEVICEbits.TXPKTRDY = 1;
24.   }
25.   else
26.   {
27.     usbhs->EPCSR[endpoint].TXCSRL_DEVICEbits.TXPKTRDY = 1;
28.   }
29. }
```

Si vuole evidenziare che nella versione USB 2.0 in un trasferimento di massa, detto anche *bulk*, la dimensione massima dei dati trasferiti per ogni transazione è 512 Byte. Dato che, con il codice mostrato, per ogni Byte è necessaria un'assegnazione, un incremento ed un confronto, sarebbero richiesti 7.68 µs solamente per riempire la coda di trasmissione. Si può mitigare questo effetto trasferento 4 Byte per ciclo, sfruttando al massimo la dimensione del *bus* di sistema che è infatti 32 bit.

Un approccio totalmente diverso consente di utilizzare il DMA, modulo richiesto nel paragrafo 2.1.1 proprio per la sua utilità. Il suo funzionamento è semplice: prende il controllo del *bus* di sistema e trasferisce dati tra due periferiche, una di sorgente ed una di destinazione. Ad ogni trasferimento viene decrementato un contatore interno, precedentemente impostato con il numero di trasferimenti totali da eseguire. Quando questo contatore raggiunge il valore '0' il trasferimento può ritenersi concluso ed il *bus* viene rilasciato. Questo permette di trasferire dati senza l'intervento della CPU, il che significa che non vengono modificati in nessun modo i suoi registri interni. In caso contrario questi andrebbero prima memorizzati (ad esempio sullo *stack*) e poi reimpostati. Un secondo vantaggio riguarda la possibilità di eseguire del codice durante il trasferimento, nel caso non venga richiesto il controllo del *bus*. Questo aspetto è ancora più significativo data la presenza di 32 KByte di memoria *cache* per le istruzioni e 4 KByte di memoria *cache* per i dati.

Di seguito è riportato il codice relativo al trasferimento DMA.

```
 1. void USBHS_DMAOperationEnable_Default
 2. (
 3.   USBHS_MODULE_ID index,//indirizzo fisico associato al modulo USB
 4.   uint8_t endpoint,           //numero dell'endpoint
 5.   uint8_t dmaChannel,   //numero del canale di DMA
 6.   void * address,       //indirizzo dei dati da trasmettere
 7.   uint32_t count,       //numero di Byte da trasmettere
 8.   bool direction        //direzione del trasferimento
 9. )
10. {
11.   volatile usbhs_registers_t * usbhs = (usbhs_registers_t *)(index);
12.   //ottiene l'indirizzo fisico dell'inizio dei dati da tramettere
13.   usbhs->DMA_CHANNEL[dmaChannel].DMAADDR = KVA_TO_PA(address);
14.   //imposta il registro del conteggio dei trasferimenti del DMA
15.   usbhs->DMA_CHANNEL[dmaChannel].DMACOUNT = (uint32_t)(count);
```

```
16.   //imposta gli altri campi del DMA
17.   usbhs->DMA_CHANNEL[dmaChannel].DMACNTLbits.DMABRSTM = 3;
18.   usbhs->DMA_CHANNEL[dmaChannel].DMACNTLbits.DMAIE = 1;
19.   usbhs->DMA_CHANNEL[dmaChannel].DMACNTLbits.DMAMODE = 0;
20.   usbhs->DMA_CHANNEL[dmaChannel].DMACNTLbits.DMAEP = endpoint;

21.   //il bit di direzione del DMA è invertito rispetto a quello dell’USB
22.   usbhs->DMA_CHANNEL[dmaChannel].DMACNTLbits.DMADIR = !direction;
23.   usbhs->DMA_CHANNEL[dmaChannel].DMACNTLbits.DMAEN = 1;
24. }
```

Si vogliono aggiungere ai commenti già presenti le seguenti osservazioni:

- Viene utilizzato il qualificatore di tipo “*volatile*” per le variabili per le quali non si può garantire la conservazione del valore in accessi successivi. Questo quantificatore impedisce al compilatore di ottimizzare il codice eliminando accessi apparentemente ridondanti a queste variabili. Viene utilizzato ad esempio per i valori dei registri.
- Il campo `DMACOUNT` (riga `15`) può contenere solamente valori multipli di 4. Questo è strettamente collegato alla dimensione del *bus* di sistema, ed è segnalato nel *datasheet*. Il controllo di questo requisito avviene precedentemente la chiamata della funzione mostrata
- Il canale DMA deve essere già stato acquisito in precedenza. Nel caso tutti i canali DMA risultino occupati è necessario attendere o utilizzare il metodo di trasferimento tradizionale.
- Il modulo DMA necessita di un indirizzo fisico, mentre la CPU utilizza indirizzi virtuali. Perciò è inserita nel codice la funzione `KVA_TO_PA`, che effettua la traduzione necessaria.

2.2.2 *File System*

Il *file system* è una struttura dati necessaria per organizzare e recuperare i dati memorizzati su un supporto digitale. Per la sua facilità di gestione e la sua diffusione è stato scelto il FAT32, descritto in Appendice B. Nel paragrafo precedente è stato descritto come impostare il trasferimento e ridurre le istruzioni eseguite dal processore, mentre in questo paragrafo ci si occuperà di ridurre il numero ed ottimizzare le richieste.

Infatti un uso poco attento del *file system* rischia di diminuire la velocità di scrittura, aumentare la latenza della richiesta e rovinare precocemente il supporto di memorizzazione.

Il primo aspetto da considerare è che tipicamente la memoria è organizzata in pagine e blocchi, come mostrato in Figura 2.3. Ogni azione di lettura o scrittura dev'essere eseguita sulla pagina intera.

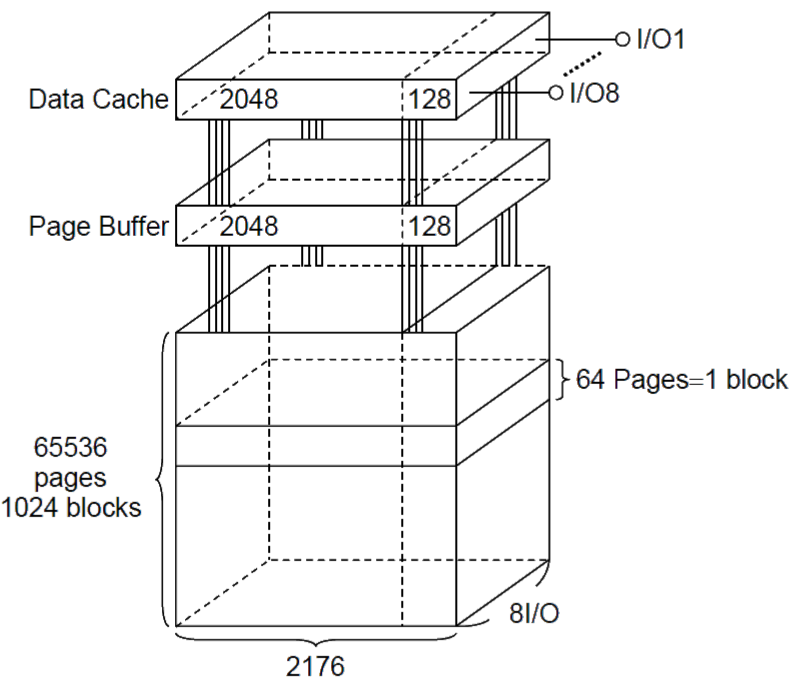


Figura 2.3: Struttura della memoria del circuito integrato TC58NVG0S3HTA00[29]

In lettura è possibile ottenere una pagina intera utilizzando solamente poco tempo in più rispetto a quello occorrente per un singolo Byte. Per questo motivo è preferibile recuperare una pagina, memorizzarla in un *buffer* in memoria volatile ed eseguire su questo le successive operazioni. Questo *buffer* dev'essere aggiornato qualora vengano richiesti indirizzi localizzati fuori dalla pagina corrente. Anche in scrittura si può adottare questa tecnica: si possono scrivere i dati in un *buffer* ed inviare la richiesta al dispositivo quando il *buffer* è pieno o quando si vuole scrivere in un'altra pagina. Per la scrittura adottare questi accorgimenti è importante in quanto la scrittura è il principale fattore di degrado di una memoria a stato solido. Solitamente vengono garantite un numero limitato di scritture per pagina.

29 "Datasheet TC58NVG0S3HTA00", Toshiba, 2018

È da evidenziare però che la dimensione della pagina non è nota a priori, perciò di norma si utilizza un *buffer* grande quanto un settore del *file system*, che per il FAT32 è di 512 Byte. Si noti che questa è anche la dimensione massima del trasferimento USB *bulk*. Invece la dimensione di un *cluster*, anche se con le impostazioni predefinite è legata alla dimensione della memoria, generalmente è un multiplo della dimensione di una pagina.
Un secondo aspetto da considerare per migliorare la velocità di scrittura può essere quello di aumentare la quantità di dati scritti in ogni richiesta. In Tabella 2.2 si mostrano i risultati ottenuti su una memoria con dimensione di *cluster* di 16 kByte per una durata del test di 10 secondi. Sono stati scelti valori multipli di 2 in modo tale da sfruttare l'allineamento delle pagine, dei *buffer*, dei blocchi ecc.

Tabella 2.2: Velocità di scrittura su memoria flash

Dimensione del blocco [Bytes]	Velocità media [kByte/s]	Blocchi al secondo	Miglioramento relativo
1024	42,6	41,6	
2048	85	41,5	1,99
4096	145,6	35,5	1,71
8192	270,4	33,0	1,86
16384	681,6	41,6	2,52
32768	1129,6	34,5	1,66
65536	1651,2	25,2	1,46
131072	2342,4	17,9	1,42
262144	2790,4	10,6	1,19

Si può notare come il massimo miglioramento si ha con una quantità di dati trasferita uguale alla dimensione del *cluster*.
L'ultimo aspetto che si vuole trattare è anche il più importante per questa applicazione: la latenza. Infatti avere una velocità di scrittura maggiore della quantità di dati da memorizzare è una condizione necessaria ma non sufficiente. Bisogna garantire che i dati non vengano persi tra l'istante di ricezione e l'istante di scrittura, che dunque non possono essere troppo distanti fra di loro. Non essendo possibile eseguire trasferimenti istantanei verrà introdotto un *buffer* che andrà a compensare questi brevi periodi di tempo in cui il dispositivo è già impegnato nella scrittura. Prima però è bene quantificare la latenza fra una scrittura e l'altra: per

questo è stato eseguito un test, i cui risultati sono mostrati in Figura 2.4. Si noti che, per meglio rappresentare i dati raccolti, entrambi gli assi sono logaritmici. Gli istogrammi hanno 20 classi distribuite uniformemente (in scala logaritmica).

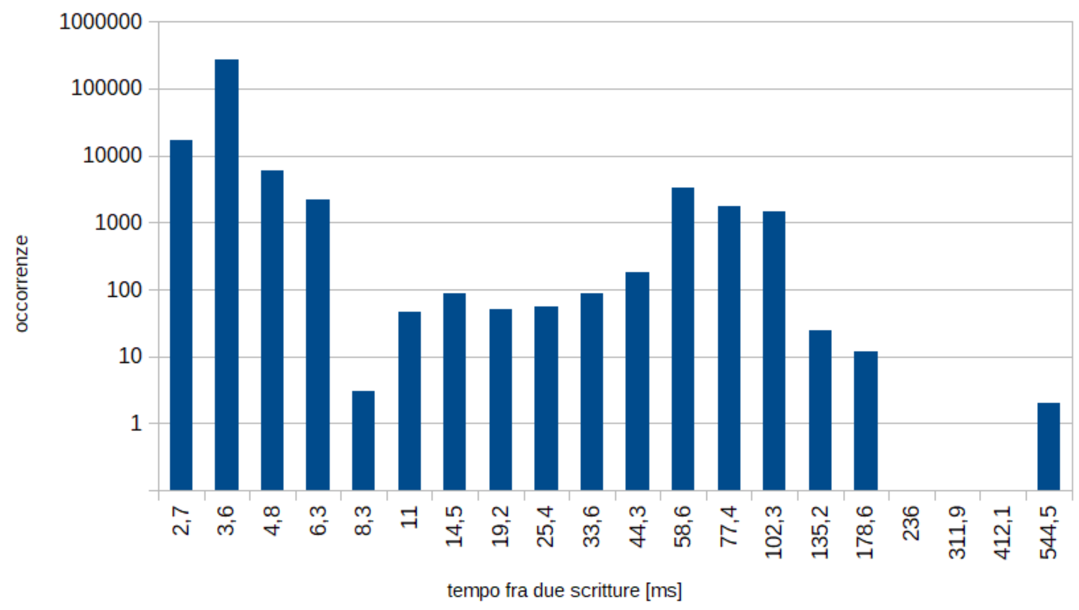


Figura 2.4: Latenza in scrittura, generica

Per eliminare il problema il ritardo dovuto dalla ricerca di blocchi di memoria liberi è stata eseguita una deframmentazione della memoria.

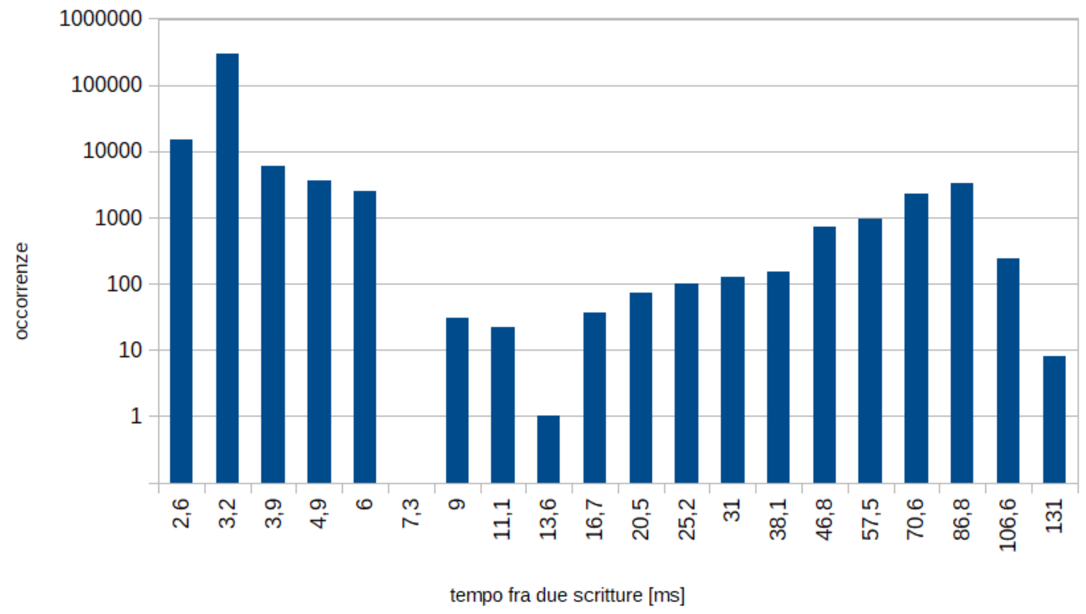


Figura 2.5: Latenza in scrittura, memoria deframmentata

Come si può notare dalla Figura 2.5 è stato ottenuto un miglioramento evidente.

Nonostante ciò si è voluto esaminare più in dettaglio la *routine* di scrittura. Proprio per la struttura del *file system FAT* ad ogni richiesta viene controllato se si è rimasti nello stesso settore, in caso contrario viene aumentato l'indirizzo. Successivamente viene controllato se si è rimasti nello stesso *cluster,* in caso contrario è necessario cercarne uno da poter utilizzare. Questa rappresenta la situazione peggiore in quanto la memoria di massa deve prima completare la scrittura dei dati presenti sulla pagina, anche se incompleti. Quindi viene letta la tabella di allocazione finché non viene trovato un *cluster* disponibile; si torna all'indirizzo del *cluster* di partenza e si accoda il *cluster* libero. Solo a questo punto si può riprendere la scrittura dei dati, indirizzandoli al *cluster* appena allocato. Con la deframmentazione i cluster occupati vengono spostati in posizioni adiacenti, rendendo adiacenti anche le locazioni libere: per questo la loro ricerca richiede meno richieste, risultando in latenze minori. Si evidenzia che risultati non significativamente diversi dalla Figura 2.5 sono stati ottenuti dopo una formattazione.

È quindi possibile evitare questo ritardo allocando lo spazio necessario prima dell'inizio del salvataggio dei dati, eventualmente ridimensionando il *file* alla fine dell'acquisizione. I risultati sono mostrati in Figura 2.6.

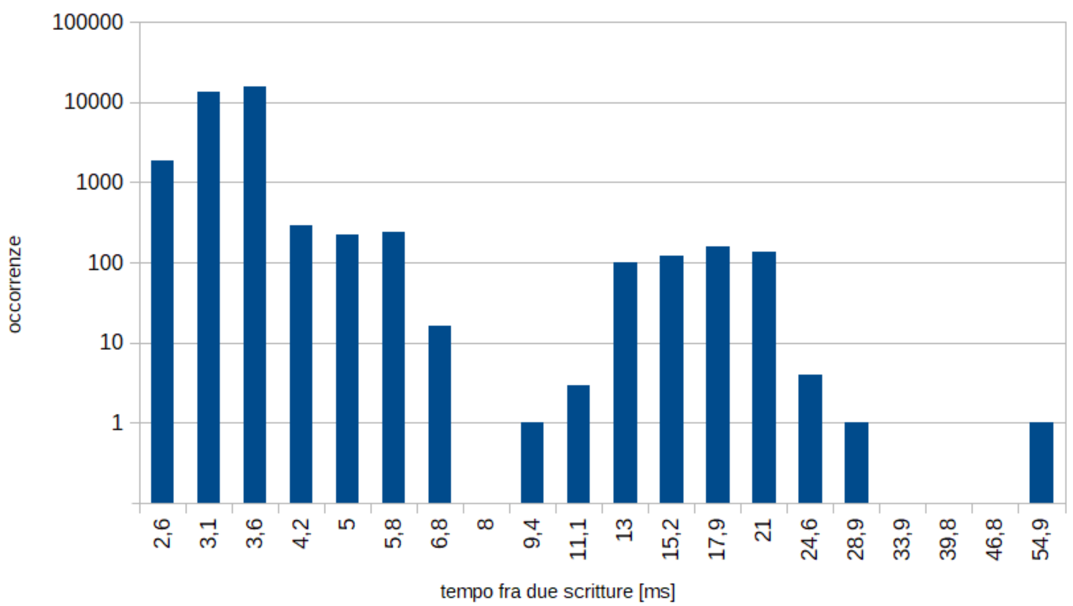


Figura 2.6: Latenza in scrittura, *file* preallocato

2.2.3 *Buffer*

Come accennato nello scorso paragrafo la presenza di un *buffer* è indispensabile per compensare eventuali ritardi di scrittura sul dispositivo di massa. In aggiunta si può rivelare molto utile in caso di disconnessione temporanea, ad esempio a causa di vibrazioni o contatti elettrici deteriorati. Infatti un *buffer* può immagazzinare piccole quantità di dati per il tempo necessario a ristabilire il collegamento, dati che altrimenti andrebbero irrimediabilmente perduti.
Mentre sulla scelta delle dimensioni si rimanda alla specifica applicazione che si vuole realizzare, ora vengono forniti i dettagli implementativi.

Buffer LIFO

Il *buffer* più semplice che si può implementare consiste in un blocco di memoria ed un indice. Quando viene inserito un dato l'indice incrementa, quando viene rimosso decrementa. Quella descritta è una coda *Last In First Out* (LIFO) che purtroppo, nonostante la semplicità, non è adatta a questa applicazione in quanto modifica l'ordine dei blocchi, come si può osservare in Figura 2.7.
Nell'esempio i valori sono scritti nell'ordine 0-1-2-3-4-5-6 mentre sono letti nell'ordine 4-3-2-6-5-1-0. Se il *buffer* dovesse risultare pieno, con questa tipologia i dati andrebbero perduti.

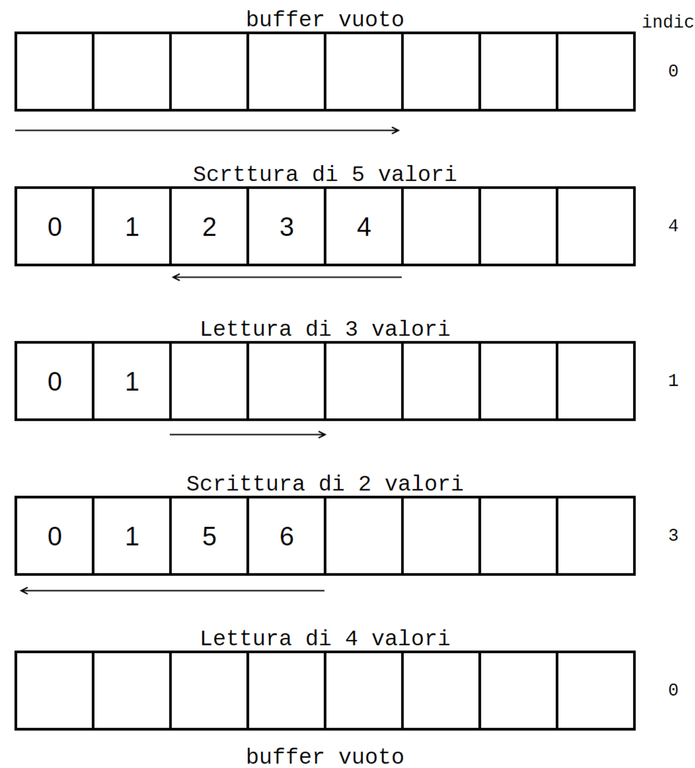


Figura 2.7: Utilizzo di un *buffer* LIFO

Buffer FIFO

Per rendere *First In First Out* (FIFO) il precedente *buffer* si può inserire un secondo indice da utilizzare per la lettura e periodicamente (ad esempio ad ogni lettura o al raggiungimento della fine del *buffer*) traslare tutto il contenuto del valore dell'indice di lettura. Ovviamente questo approccio è poco pratico, anche se sarebbe possibile sfruttare il DMA per ridurne l'impatto. Se il *buffer* dovesse risultare pieno, anche con questa tipologia i dati andrebbero perduti, a meno di simulare una lettura traslando tutti i valori. In questo secondo caso verrebbero persi i dati meno recenti.

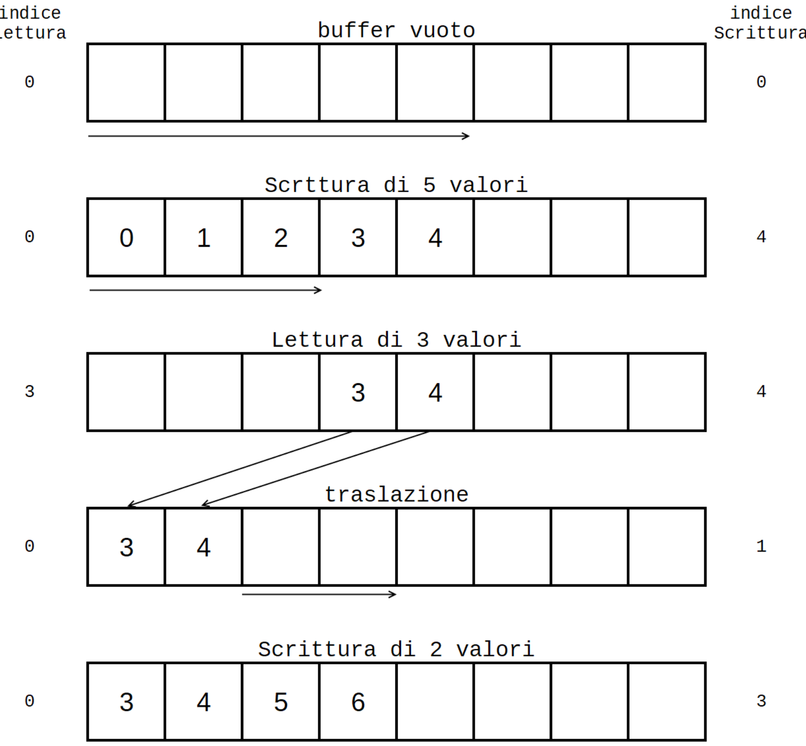


Figura 2.8: Utilizzo di un *buffer* FIFO

Buffer doppio

Un approccio differente è quello di utilizzare due *buffer*, uno in lettura ed uno in scrittura. Quando il primo è pieno vengono scambiati (se il secondo è vuoto o "libero"), come mostrato in Figura 2.9. In questo modo la lettura potrà incominciare e la scrittura continuare, e le due operazioni potranno essere svolte anche contemporaneamente (dato che operano su aree separate). Ogni *buffer* ha il proprio indice ed entrambi possono solo aumentare. Questa tecnica è utilizzata ad esempio per la visualizzazione di immagini: in quest'ambito infatti il *buffer* in lettura è sempre "libero" perché i dati in esso contenuti possono essere sempre essere sovrascritti perchè hanno esaurito la loro funzionalità, in quanto meno recenti. Il *buffer* in scrittura è dedicato all'aggiornamento dei dati, mentre quello in lettura mantiene i dati costanti per evitare indesiderati effetti visivi, come il parziale aggiornamento della schermata. Il *buffer* in lettura può essere letto più volte: questo dipende da quanto tempo viene impiegato per preparare una nuova immagine. Ogni volta che una nuova immagine è pronta i ruoli dei due *buffer* vengono scambiati ed il

processo si ripete. La frequenza di questo aggiornamento viene anche detta *frames per second* (FPS).

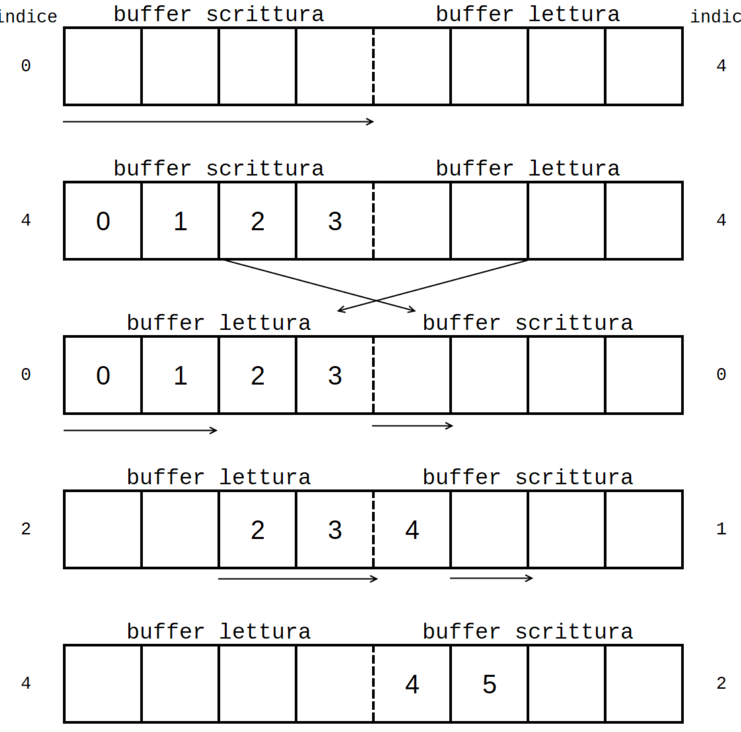


Figura 2.9: Utilizzo di un *buffer* doppio

Nonostante questo *buffer* sia abbastanza semplice da gestire ed utilizzare ha lo svantaggio di dimezzare la quantità di dati immagazzinabile a parità di memoria occupata. Con questo approccio al riempimento del *buffer* in scrittura si può scegliere se attendere la disponibilità del *buffer* in lettura, perdendo i dati più recenti, oppure se effettuare comunque lo scambio, perdendo i dati più antichi. In Figura 2.9 si è scelto di adottare lo stesso numero di blocchi degli esempi precedenti e di marcare la divisione tra i due *buffer* tramite una linea tratteggiata.

Buffer circolare

Questa implementazione ha molti aspetti in comune con quella del *buffer* FIFO. Sono presenti due indici, uno di scrittura (testa) e uno di lettura (coda) che vengono incrementati ad ogni utilizzo. La sua particolarità è evidente quando viene raggiunta

la fine del *buffer*: l'indice di testa viene azzerato e la scrittura ricomincia dall'inizio fisico del *buffer*. Virtualmente dunque l'ultimo ed il primo blocco sono contigui.

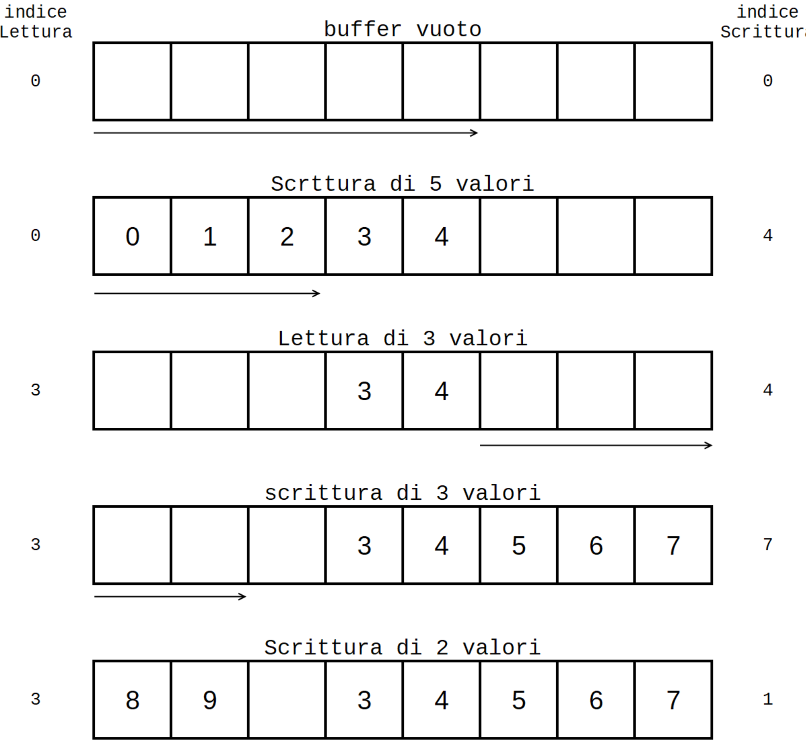


Figura 2.10: Utilizzo di un *buffer* circolare

Si prenda ad esempio la Figura 2.10 : prima dell'ultima scrittura l'indice di testa viene azzerato ed i successivi dati vengono collocati all'inizio del *buffer*.
Alternativamente per ottenere questo comportamento è possibile utilizzare l'operatore modulo come in Formula 1. Questo operatore, indicato in molti linguaggi di programmazione con il carattere % , restituisce il resto di una divisione.

$$indirizzo\ fisico = indirizzo\ virtuale \,\%\, dimensione\ del\ buffer$$

Formula 1: Indirizzamento di un *buffer* circolare

Nell'esempio mostrato incrementando il penultimo indice da 7 a 9 e applicando la Formula 1 si ottiene 1. Ovviamente l'indirizzo virtuale non può incrementare all'infinito, quindi dev'essere ridimensionato prima che diventi troppo grande per essere rappresentato con il suo tipo di dato. Questa condizione è detta *overflow*.

Un altro approccio sfrutta proprio l'*overflow*: infatti un'operazione che porta ad un risultato non rappresentabile viene comunque eseguita, mentre la cifra più significativa viene persa. Questo comportamento porta il risultato a coincidere con il resto: dunque si può utilizzare facendo coincidere la dimensione del *buffer* con il massimo numero rappresentabile per l'indice. Nel nostro caso si potrebbe scegliere una rappresentazione a 16 bit, che permetterebbe di avere un *buffer* di 65536 Byte.
Volendo generalizzare questo procedimento è possibile applicare una maschera all'indice: questo permette di riprodurre il comportamento dell'*overflow*, e quindi dell'operatore modulo. Questa maschera deve essere composta da '0' fino alla cifra desiderata, successivamente da '1'. Per applicare questa maschera all'indice è sufficiente eseguire un'operazione AND bit a bit: in questo modo rimarranno solamente le cifre più significative. Il codice è molto semplice, ed è sufficiente applicarlo solamente agli incrementi degli indici (lo stesso procedimento vale sia per la testa che per la coda):

```
1. indice = (indice + 1) & (DIMENSIONE_buffer-1)
```

oppure

```
1. indice++;
2. indice &= (DIMENSIONE_buffer-1);
```

Si noti che se `DIMENSIONE_buffer` è una potenza di 2, `DIMENSIONE_buffer-1` ha la forma 0...01...1 ed in ogni caso può non essere calcolata ogni volta.
Questa implementazione, semplicità ma al tempo stesso flessibile, è quella che verrà utilizzata in seguito.
Esiste un'ulteriore possibilità, che è dovuta dal fatto che il processore non utilizza indirizzi fisici ma virtuali: è dunque presente un modulo, solitamente chiamata unità di gestione della memoria (in inglese *Memory Management Unit*, MMU), che traduce gli indirizzi in modo del tutto trasparente rispetto all'esecuzione dei programmi (Figura 2.11). Solo alcune periferiche, come il DMA accennato in precedenza, utilizzano gli indirizzi fisici.
In linea teorica sarebbe dunque possibile mappare blocchi di memoria virtuale adiacenti alla stessa locazione nella memoria fisica, della dimensione del *buffer*. Così facendo la richiesta di qualsiasi indirizzo ricadrebbe sugli stessi blocchi fisici,

realizzando in modo efficiente e trasparente il *buffer* circolare. Purtroppo il microcontrollore scelto non ha la possibilità di impostare questa mappatura, se non per l'utilizzo di componenti esterni.
Anche in questo caso al riempimento del *buffer*, che si verifica quando l'indirizzo di coda è identico a quello di testa, si può scegliere se perdere i dati più recenti, lasciando invariati gli indici, o quelli meno recenti, facendoli avanzare entrambi.

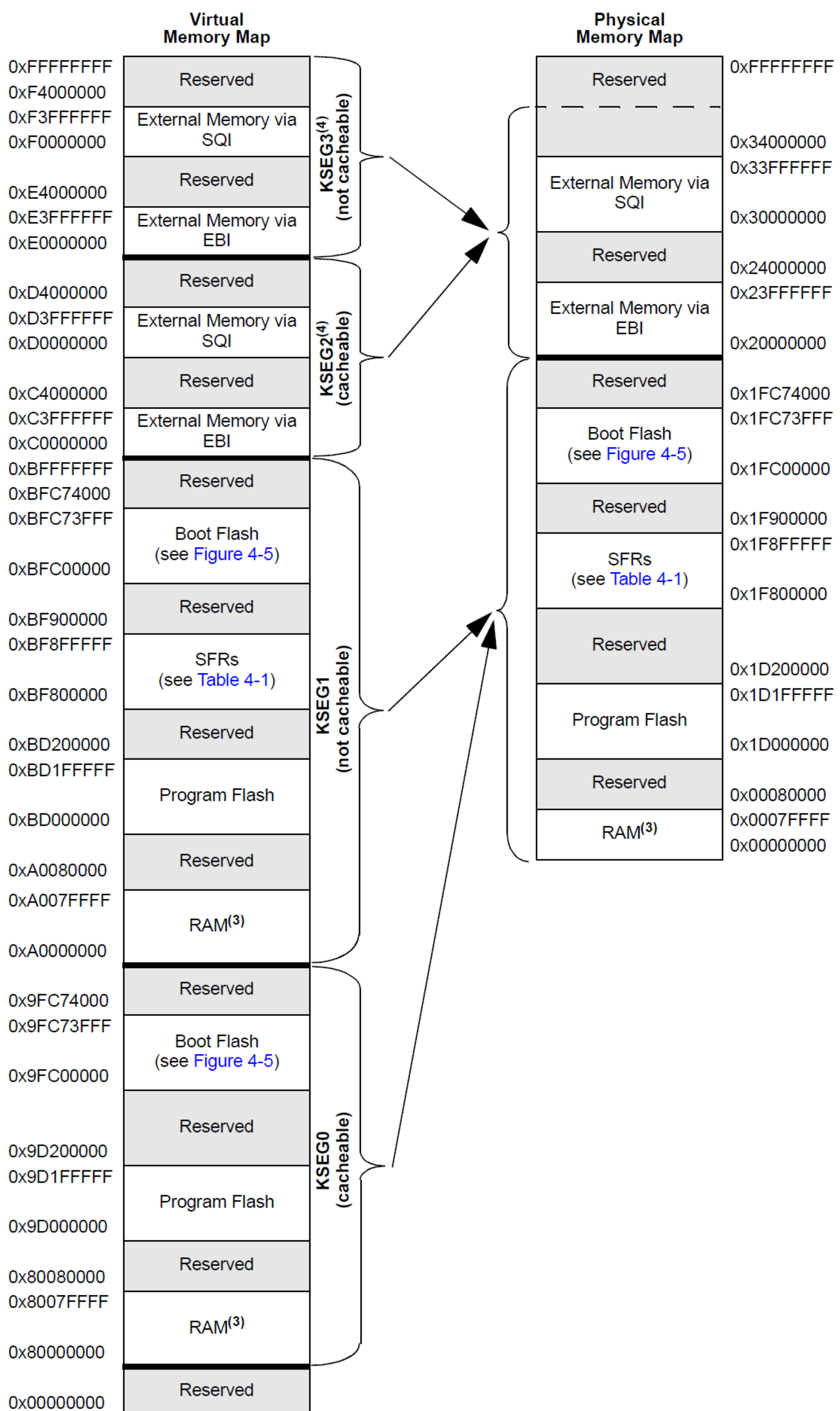


Figura 2.11: Mappatura della memoria - PIC32MZEF

2.3 Struttura dei *file*

2.3.1 Organizzazione dei dati

Come evidenziato nel paragrafo 2.2.2 avere strutture di dati complessi non è ottimale né per la latenza né per l'usura del dispositivo di memorizzazione. L'approccio più semplice, che è quello tipico di un *file* di *log*, consiste nel memorizzare i dati in sequenza temporale. Per ciascun dato dev'essere possibile risalire sia alla fonte, o canale, da cui proviene sia all'istante temporale in cui è stato ricevuto.
È possibile utilizzare una struttura composta da tempo, canale e dato. Questa, anche se non è la più compatta, permette di associare con facilità le tre informazioni.
Per ridurre l'utilizzo di spazio è possibile raggruppare i dati. Per preservare la sequenza temporale è da scartare il raggruppamento per dato (in diversi istanti temporali), mentre è possibile il raggruppamento temporale (dati raccolti nello stesso istante). Questa strategia è tanto più efficace quanti più sono dati che vengono raccolti nello stesso istante temporale. Si presenta quindi un *trade-off* fra la granularità del tempo e la quantità di dati da memorizzare. Aumentando precisione temporale diventa più raro che due eventi possano considerarsi avvenuti contemporaneamente, diminuendo l'utilizzo dell'ottimizzazione appena presentata.
Questa può essere migliorata nel caso i dati si presentino con una certa regolarità, definendo a priori l'ordine di memorizzazione dei dati e dunque l'associazione col rispettivo canale. Questa tecnica è l'ideale nel caso di un sottocampionamento, dato che entrambe le condizioni necessarie sono verificate.

2.3.2 Tipologie di *file*

File di testo

Il *file* di testo è un *file* contenente solamente testo puro, ossia la codifica binaria di caratteri comprensibili a un lettore umano. Nel caso non ci sia la necessità di rappresentare caratteri particolari la scelta più comune è quella di adottare la codifica ASCII estesa (acronimo di *Ameri*CAN *Standard Code for Information*

Interchange) che, con 8 bit per carattere, permette di rappresentare 33 caratteri non stampabili (detti di controllo) e 223 caratteri stampabili, per un totale di 256 combinazioni.
È possibile rappresentare anche valori numerici, scrivendo per ogni cifra il corrispondente carattere. Questa conversione fornisce un vantaggio di interoperabilità poichè i caratteri non dipendono né dalla dimensione del campo né dall'ordine in cui vengono gestiti i dati in una particolare architettura (*Endianess*).
Un *file* di testo è meno compatto in confronto alla rispettivi valori binari. In Tabella 2.3 è mostrato l'incremento di dimensione per alcuni tipi comuni di dati numerici. È da notare che la dimensione di rappresentazione di un dato di tipo a virgola mobile (detto *floating point*), come può essere un numero *float* o *double*, dipende dalle cifre significative che vogliono essere mostrate. Comunque generalmente viene preferito limitarne il numero perdendo risoluzione. In Tabella 2.3 il confronto dei numeri a virgola mobile è effettuato supponendo di usare la rappresentazione esponenziale standard, che è nella forma *[-]d.ddd e[+/-]ddd* , dove *d* rappresenta una cifra, *e* il carattere 'e' ed il primo segno (fra parentesi quadre) è opzionale.

Tabella 2.3: dimensione di alcuni tipi di dato, in formato binario e testuale

	Dimensione in bit	Byte necessari per la rappresentazione binaria	Byte necessari per la rappresentazione testuale (con segno)	Byte necessari per la rappresentazione testuale (senza segno)
interi	8	1	4	3
	12	2	5	4
	16	2	6	5
	32	4	11	10
	64	8	20	20
virgola mobile	32	4	11	10
	64	8	11	10

Anche se la differenza fra le due rappresentazioni è minima in termini assoluti, può diventare significativa in applicazioni in cui è necessario gestire un'elevata quantità di dati.
Si evidenzia come in generale la lunghezza dei campi sia variabile, per cui si rende necessario inserire ulteriori caratteri per la loro separazione. Anche se è possibile rendere fissa la lunghezza dei campi definendo gli eventuali caratteri di riempimento

è comunque consigliabile inserire dei delimitatori per aumentare la leggibilità dei dati.
Un altro aspetto da considerare è che la conversione in testo di un dato non avviene istantaneamente. In Tabella 2.4 sono mostrate a titolo esemplificativo le prestazioni della funzione di conversione `sprintf`, così come implementata sul microcontrollore. È inserito come riferimento anche il tempo di esecuzione di un ciclo `for`.

Tabella 2.4: Prestazioni di conversione di dati in testo

Formato[30]	Tempo di esecuzione [μs]
%d	8,66
%010d	9,84
%e	12,77
%f	24,01
%.4f	21,22
%010u\n%f,%.2f,%e\n	62,96
ciclo semplice	0,05

Solamente l'utilizzo della conversione limita il numero di campioni processabili a circa 100 ksps nel caso di interi e 50 ksps nel caso di numeri a virgola mobile, pur considerando questa come unica attività svolta dal microcontrollore.
Questa tipologia di *file* è invece particolarmente indicata per l'interazione umana: per questo motivo verrà adottata per il *file* di configurazione, che dev'essere facilmente modificabile dall'utente e non richiede grandi quantità di dati. L'ultima condizione permette di utilizzare linguaggi di interscambio dei dati, i quali possono aumentare ulteriormente la comprensibilità ed interoperabilità del *file*. È presente la possibilità di una descrizione i campi direttamente all'interno del *file*. Due linguaggi particolarmente diffusi sono XML e JSON: il primo è piuttosto pesante e complesso da implementare su un microcontrollore, soprattutto nel caso se ne volessero rispettare pienamente le specifiche. Il secondo invece è adatto per l'applicazione e perciò è stato implementato.

File binari

Diversamente dai *file* di testo non sono interpretabili senza una descrizione della loro struttura che sia contestuale, in un *file* separato o nota a priori. Possono

30 Per informazioni sul formato consultare http://www.cplusplus.com/reference/cstdio/printf/

essere implementate tutte le organizzazioni dei dati presentate nel paragrafo 2.3.1, anche se la mancanza di delimitatori li rende molto fragili. Infatti in caso di corruzione dei dati tutto o parte del contenuto diventa illeggibile. Tra le cause ci sono gli errori di scrittura e di trasferimento, il disallineamento degli indici, il degrado del supporto di memoria.
In alcuni casi è possibile recuperare i dati sfruttando le informazioni sulla struttura dei dati: si ricercano le sequenze possibili e coerenti, scartando quelle che non rispettano i vincoli. Il recupero è tanto più probabile quanto più è elevata la dimensione del *file*, in quanto si riducono le probabilità di avere un falso positivo fino ad arrivare, in linea teorica, ad avere una sola combinazione valida. Questo ragionamento suppone che sia presente una sola irregolarità, che la struttura abbia lunghezza costante e che si conosca la posizione dell'errore. Inoltre non tiene conto del costo computazionale. Infine, nel caso sia presente più di un errore o la struttura abbia lunghezza variabile, il recupero non garantisce una soluzione univoca, neanche con la dimensione del *file* tendente all'infinito.
Per questo motivo il *file* binario sequenziale non è adatto in applicazioni in cui è importante garantire l'integrità dei dati, come quella di un *data logger*. Possono essere elaborate strutture particolari resistenti agli errori, però queste in generale non rispettano la sequanzialità temporale dei dati, caratteristica desiderata per la semplicità di gestione e per gli aspetti relativi ai dispositivi di memoria discussi in precedenza.

2.3.3 Ridondanza e integrità dei dati

Per verificare l'integrità dei dati viene utilizzato un messaggio, chiamato *checksum*, che può essere ottenuto da due tipologie di algoritmi: quelli irreversibili, e quelli reversibili.
Gli algoritmi irreversibili sono utilizzati in genere per garantire la protezione dei dati contro l'alterazione volontaria, in quanto è computazionalmente impossibile creare intenzionalmente due messaggi diversi con lo stesso *checksum*. Di questa categoria fanno parte gli algoritmi *SHA* e *MD5*, che possono essere calcolati facilmente su alcuni microcontrollori come il PIC32MZ2048EFM064 grazie alla presenza di un apposito modulo crittografico.
Gli algoritmi reversibili sono invece utilizzati in contesti dove un'eventuale intrusione è fisicamente impossibile o non rilevante. Inoltre, data la loro

reversibilità, possono essere utilizzati per il recupero di dati in caso di corruzione. Di questa categoria fa parte il *Cyclic Redundancy Check*[31] (CRC), che è ampiamente utilizzato nel *layer* fisico delle comunicazioni. Esempi di utilizzo sono nel *bus* CAN o nel protocollo ethernet. Anche il bit di parità utilizzato nelle comunicazioni può essere considerato come un particolare CRC (ad un bit).
Non avendo necessità crittografiche potrebbe essere utilizzato il CRC, che nel microcontrollore scelto può essere calcolato direttamente dal modulo DMA durante un trasferimento.
Nonostante ciò viene ricercato un algoritmo che produca un *checksum* che possa essere usato per il recupero dei dati, oltre che per la verifica dell'integrità. Infatti non sarebbe utile calcolare il CRC o il bit di parità, in quanto sarebbe memorizzato nella stessa *pagina* e quindi corrotto allo stesso modo dei dati che si vorrebbero recuperare.
Per ottenere questo risultato si è adottato uno schema concettualmente uguale a quello di un RAID di livello 4. Un *Redundant Array of Independent Disks* è una tecnica di installazione raggruppata di diversi dischi rigidi in un computer (o collegati ad esso) che fa sì che gli stessi appaiano nel sistema e siano utilizzabili come se fossero un unico volume di memorizzazione.
Il funzionamento è molto semplice: il contenuto del disco N viene ottenuto come valore di parità degli altri dischi. Logicamente questa operazione corrisponde ad uno XOR, indicato nella Formula 2 con il simbolo ⊕.

$$X_{N(i)} = X_{2(i)} \oplus X_{1(i)} \oplus X_{0(i)}$$

Formula 2: Ottenimento blocco di parità

Questa formula è reversibile: in caso di danneggiamento di un disco, si possono scambiare gli indici del disco di parità e del disco danneggiato.
Nel nostro caso si ha una sola unità di memorizzazione, ma il funzionamento può essere replicato per le *pagine*. Questo calcolo può essere anche estremamente efficiente nel caso di una scrittura sequenziale: richiede due sole istruzioni aggiuntive per ogni 4 Byte aggiunti al *buffer* circolare, se effettuate contestualmente al trasferimento. Il blocco di parità calcolato può avere anch'esso la struttura di *buffer* circolare, semplificando in questo modo i calcoli. In questo caso la sua dimensione dev'essere sottomultipla del *buffer* dei dati (ma multipla di quella di una

31 Peterson, W. Wĸ. and Brown, D. T., "Cyclic Codes for Error Detection, in Proceedings of the IRE", DOI:10.1109/JRPROC.1961.287814, ISSN 0096-8390 (WC · ACNP)

pagina, per essere efficace): in scrittura bisogna solamente avere l'accortezza di scrivere il blocco di parità ad ogni "ritorno a capo" dell'indice di scrittura.

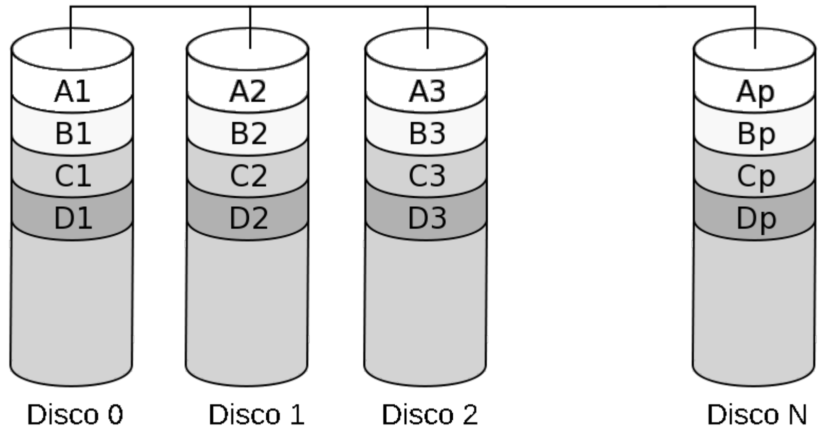


Figura 2.12: Schema RAID 4

Con questo metodo, nel caso di corruzione di una pagina può esserne recuperato il contenuto in modo rapido.
La ridondanza introdotta può supplire ad un solo errore: se necessario può essere introdotto un numero arbitrario di blocchi di parità, che permette di correggere un pari numero di errori, che possono essere situati nei blocchi di parità stessi.
Un'altra particolarità è la regolarità apportata da questo approccio al *file* binario: è possibile allineare la struttura dati al blocco, eventualmente riempiendo di dati casuali la coda (se non utilizzabile). Questo a permettere il loro utilizzo nel campo del *data logging*.
L'implementazione che si propone è di 8 kByte oppure 16 kByte di controllo ogni 256 kByte di dati, cioè in rapporto 1 a 32 o 1 a 16. Si noti che è introdotto un *overhead*. In particolare, supponendo un numero casuale di errori si possono ritenere scorrelate le probabilità di ogni singolo bit. Perciò la probabilità di ottenere un falso positivo è rispettivamente di $9\ 10^{-2467}$ e di $8\ 10^{-4933}$. Si noti che, in generale, è un evento molto raro che si verifichino errori su più pagine contemporaneamente, dunque la probabilità reale è ancora più bassa. Inoltre è possibile correggere fino ad 1 errore ogni 32 blocchi o 1 errore ogni 16 blocchi, rispettivamente.

CAPITOLO 3
CONCLUSIONI

3.1 Applicazioni

In questo paragrafo viene mostrata la flessibilità del dispositivo realizzato proponendo alcune applicazioni nelle quali può essere utilizzato. Alcune funzioni sono solamente proposte ma non implementate perché dipendono fortemente dal contesto e dalle specifiche. L'implementazione che si è voluta fornire è invece la più generale possibile.

3.1.1 Strumento da laboratorio

Questa applicazione richiede un collegamento con un *computer* per funzionare correttamente. Le uniche operazioni svolte dal microcontrollore sono il *time stamping* (cioè l'aggiunta dell'indicazione del tempo in cui ogni dato è stato ricevuto), il *buffering* ed eventualmente una decimazione o filtraggio. È da evidenziare che non viene utilizzato il modulo USB OTG per la comunicazione. A tal proposito ci sono due possibilità:

- trasferimento dei dati tramite protocollo USB, impostando il *data logger* come normale dispositivo *slave*. Data la possibilità di gestire direttamente il protocollo USB, possono essere raggiunte le velocità di trasferimento elevate,

cioè quelle indicate dallo standard. In particolare è possibile di valutare i trasferimenti *bulk* o *isochronous*.

- trasferimento dei dati tramite protocollo ethernet. È necessario gestire l'indirizzamento ed avere una rete in cui trasmettere i dati. Non essendo stato implementato sarebbe da valutare la massima velocità di trasferimento ottenibile. La velocità teorica ottenibile sarebbe un quinto rispetto allo standard USB, anche se viene richiesto un *overhead* complessivo leggermente minore. Si sottolinea che questo limite è relativo alla versione dello standard supportata dal microcontrollore.

È importante realizzare una struttura per lo scambio dei dati, che sia comune tra *data logger* e computer. Si esclude la conversione dei campi in testo, per le limitazioni descritte nel sotto-paragrafo 2.3.2. Può essere scelta la struttura esposta in precedenza, in cui inizialmente vengono trasmesse le informazioni sui vari campi (in particolare numero identificativo e dimensione). È da notare che non si pongono i problemi della ridondanza e dell'identificazione del significato dei singoli bit. La ridondanza diventa superflua perché implementata in entrambi gli standard di trasmissione. Il significato dei bit, che non può essere dedotto a priori posizionandosi in un punto casuale di un *file* binario, è facile da ricavare da messaggi di lunghezza variabile. Infine dev'essere creato il programma del computer che si occupi di elaborare i dati ed eventualmente memorizzarli.

3.1.2 *Data logger* per MotorSport

Le sessioni in ambito motorsport hanno una durata compresa fra i 40 ed i 90 minuti[32]. Data l'avanzata tecnologia impiegata è necessario registrare una grande quantità di dati, tipicamente 35 MByte a giro[33], da cui si può stimare un *bitrate* di 0.5 - 1 MByte/s. Per questo motivo si può garantire che la dimensione massima permessa dal *file system* FAT32, cioè 4 GByte, sia sufficiente. Come evidenziato nel paragrafo 1.3 attualmente è richiesta una granularità temporale di 1 ms. Con lo sviluppo presentato può essere abbassato notevolmente questo limite: in particolare viene proposto un tempo di riferimento di 500 ns. Il motivo di questa scelta risiede nella possibilità di analizzare lo stato di utilizzo del *bus* CAN e di sapere se il

32 Ferretti C. · Frasca A. "Enciclopedia dello sport" Garzanti Libri, 2008, p. 1670.

33 https://it.motorsport.com/f1/news/formula-1-la-telemetria-ferrari-coinvolge-cento-persone/598644/

messaggio ricevuto è stato trasmesso in tempo reale. Infatti, dato che ad ogni messaggio è associata la relativa lunghezza ed il tempo di arrivo è possibile risalire all'istante di inizio della trasmissione. Se due messaggi sono distanziati di almeno 1 μs si ha la certezza che il secondo non possa aver perso un arbitraggio o che, più in generale, non abbia dovuto attendere per essere trasmesso.
Per minimizzare l'impatto di una risoluzione così elevata per ogni dato vengono registrati solamente i primi due Byte relativi al tempo, mentre i restanti vengono registrati come campi separati seguendo gli stessi principi del resto dei dati. Questo avviene ad ogni loro aggiornamento, che corrisponde all'*overflow* dei Byte meno significativi)

3.1.3 Registratore a ciclo continuo

Dall'applicazione precedente è possibile ricavare un registratore dati per uso comune. Viene supposto un *bitrate* minore, 0.2 - 0.5 MByte/s, che corrisponde ad un uso non intensivo di due *bus* CAN, ai dati raccolti da un accelerometro, giroscopio e magnetometro a 3 assi, solidali con la vettura (integrati nel *data logger*) ed altri dati a bassa frequenza. Non è possibile stabilire a priori la durata massima di un utilizzo continuo, perciò il *file system* scelto risulta inadeguato. Sotto queste ipotesi può essere garantito solo un intervallo di registrazione compreso tra 2h15' e 3h30' per *file*. Per aggirare questo problema può essere preallocata tutta la memoria disponibile sulla memoria di massa, separata in diversi *file*: quando necessario si può eseguire una routine che cambi il *file* attualmente in uso. Il cambio di *file* dev'essere gestito correttamente, in modo da evitare la perdita di dati a causa della latenza introdotta. Dato che si desidera un'acquisizione continua, i dati meno recenti possono essere sovrascritti. Utilizzando una memoria di 32GByte possono essere registrate in modo certo più di 15 ore di guida: ovviamente questo rappresenta il caso peggiore. Viene anticipato che come sviluppo futuro può essere aggiunto il supporto al *file system* ExFat, che rimuove le limitazioni dei *file* indicate e permette l'utilizzo di memorie di dimensione maggiore.

3.1.4 Event Data Recorder

Come si è visto nel Capitolo 2 sono stati adottati vari accorgimenti per diminuire l'utilizzo del processore, che rimane dunque libero per altre elaborazioni. Una di queste può essere l'individuazione di situazioni anomale, in particolare degli impatti, per i quali sono disponibili numerosi algoritmi basati principalmente sulle misure di accelerazione e velocità. Ad esempio si possono facilmente ricostruire l'angolo di impatto e il PDOF, *principal direction of force* (PDOF). Non si entra nei dettagli del significato di questi indici e del loro calcolo in quanto sono stati effettuati molti studi approfonditi e specifici sul tema (come il [3]). Quello che si vuole evidenziare è che gli algoritmi più semplici richiedono pochissima potenza di calcolo e possono essere eseguiti *realtime*, anche se la loro priorità dev'essere inferiore a quella della gestione e memorizzazione dei dati in arrivo e perciò eseguiti in *background*.
È anche possibile associare delle azioni agli eventi individuati. Un esempio potrebbe essere quello di interrompere l'acquisizione di alcuni dati (accelerazioni e velocità angolari) in caso di prolungata velocità lineare nulla, caso che si verifica ad esempio durante l'attesa ad un semaforo. Accorgimenti di questo genere possono estendere la durata delle tracce registrate. Si ricorda comunque che lo scopo principale dei dispositivi presentati è quello di registrare dati, perciò si esclude l'implementazione di eventuali azioni che non siano a tale indirizzo. Come discusso nell'introduzione è pratica comune e significativa separare tutti i dispositivi per obiettivo, in modo tale che siano dedicati ad un solo compito o ad un insieme ristretto di azioni simili.
Un'altra funzione che è possibile implementare oltre quelle già proposte è quella di effettuare calcoli di indici di valutazione della guida, eventualmente associando i dati non aggregati degli eventi più importanti.

3.2 Sviluppi futuri

In questo studio è stato preso come riferimento l'attuale stato dell'arte. Non ci si è voluti sbilanciare sulla direzione futura del settore in quanto dipendente da sviluppi politici, economici e tecnologici. Questo però non rappresenta una forte limitazione all'applicabilità di quanto esposto dato che l'introduzione di nuove tecnologie richiede tempo e che solitamente le normative sono applicate solamente ai nuovi veicoli. Con queste considerazioni, osservando la Figura 3.1, si può dedurre l'aspettativa di vita dell'implementazione fornita. I concetti presentati invece saranno validi per un periodo più lungo, opportunamente integrati.

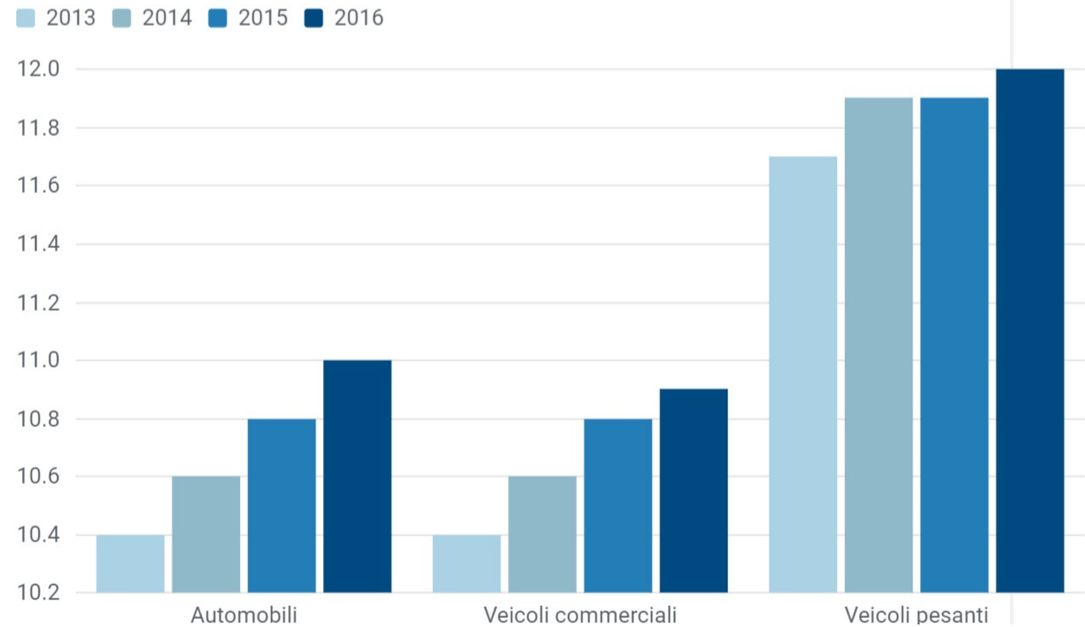


Figura 3.1: Età media in anni degli autoveicoli in Unione Europea[34]

Vengono ora esposti alcuni sviluppi che sembrano plausibili.

3.2.1 Trasmissione dei dati

La principale limitazione dello standard CAN 2.0 è la sua velocità, 1 Mbit/s. Anche se attualmente si è lontani da un utilizzo così intensivo del *bus*, si prospettano due alternative, il CAN-FD e l'*ethernet*.

34 Fonte: https://www.acea.be/statistics/article/average-vehicle-age (consultato il 20/03/2019)

Il CAN-FD pur essendo stato presentato nel 2012 non è ancora presente in nessuna autovettura, anche se a livello industriale esistono molte soluzioni che lo implementano. Nell'ambito *automotive* alcune previsioni suggerivano come biennio di comparizione il 2017-2018[35] o il 2019-2020[36]. Tale standard permette di aumentare di 8 volte la quantità effettiva di dati scambiati, presentata in Tabella A.4, pur mantenendo inalterata la durata di ogni *frame*.

In caso di introduzione del CAN-FD non varierebbe in alcun modo la discussione presentata nell'elaborato, ad eccezione della scelta di un microcontrollore che lo supporti.

Se invece dovesse prevalere l'uso dell'ethernet gli scenari disponibili sarebbero più variegati. Infatti, per basse quantità di dati (ethernet 10/100) non sarebbe neppure necessario cambiare dispositivo, potendo riprogrammare il *device* proposto in questo studio a tale scopo. La gestione dei dati e la memorizzazione sarebbe identica a quella presentata nei precedenti capitoli. Con una quantità più imponente di dati (Gigabit ethernet) andrebbe invece rivisto il significato e gli obiettivi di un *data logger*: cosa dev'essere memorizzato? Con quale risoluzione temporale? Quanto è importante la sincronizzazione temporale con gli altri dati? Considerando che l'uso del Gigabit ethernet sarebbe subordinato ad un'introduzione massiccia dell'informatica nei veicoli, probabilmente la soluzione più adatta per riprodurre le finalità di *logging* sarebbe gestire una sincronizzazione temporale dei vari dispositivi e decentralizzare in maniera totale o parziale il compito della memorizzazione. Tale assunzione si basa sul fatto che una generazione di dati di questa entità prevede l'esistenza di dispositivi in grado di gestirli. Perciò vengono meno le motivazioni sullo svantaggio in termini di consumi e di costi rispetto all'utilizzo di microprocessori, in quanto già presenti. Se la sincronizzazione fra i vari dispositivi avviene correttamente si può dunque garantire che i dati generati e memorizzati da questi siano coerenti fra di loro.

35 https://www.dspace.com/en/inc/home/news/blog-0515-can-fd.cfm

36 https://www.can-cia.org/ru/news/cia-in-action/view/can-2020-the-future-of-can-technology/2016/3/21/

APPENDICE A
PROTOCOLLI DI TRASMISSIONE

Questa appendice tratta ad un livello più dettagliato i protocolli di trasmissione dati citati ed utilizzati nel testo. In particolare si descrive l'architettura fisica, la logica e i pacchetti di dati del *bus* CAN e presenta le caratteristiche dell'USB.

A.1 CAN

Il *Controller Area Network* è uno standard seriale nato in ambiente *automotive* e poi diffuso in molte applicazioni industriali. Le specifiche si sono descritte in [4].

I punti di forza sono:

- la velocità, dato che il *bit rate* può raggiungere 1 Mbit/s per reti lunghe meno di 40m e 125 kbit/s per 500m
- la resistenza ai disturbi, aumentabile con vari accorgimenti. Nell'*High Speed* CAN, descritto in seguito, viene utilizzata una linea a differenza di potenziale bilanciata.
- la possibilità di trasmettere a più dispositivi contemporaneamente, detta anche *multicast*.
- la possibilità di avere più *master*. Un dispositivo è *master* se può iniziare autonomamente la trasmissione prendendo così il controllo del *bus,* cioè del mezzo di trasmissione.

Gli standard che descrivono il protocollo sono principalmente relativi ai livelli 1 e 2 del modello OSI (ISO 11898/ISO 16845), specificando il livello 3 e 4 solo per particolari applicazioni (ad esempio la diagnostica *automotive*, ISO 14229).

A.1.1 Architettura fisica

I nodi, cioè tutti i dispositivi *hardware* collegati al *bus* ed in grado di comunicare, sono connessi tramite un doppino intrecciato. Esistono ad oggi due standard ISO per il *layer* fisico:

High Speed CAN (ISO 11898-2)

La topologia è a *bus* lineare, come mostrato in Figura A.1.1, terminato da ciascun lato da una resistenza di 120 Ω. Permette velocità fino ad 1 Mbit/s.

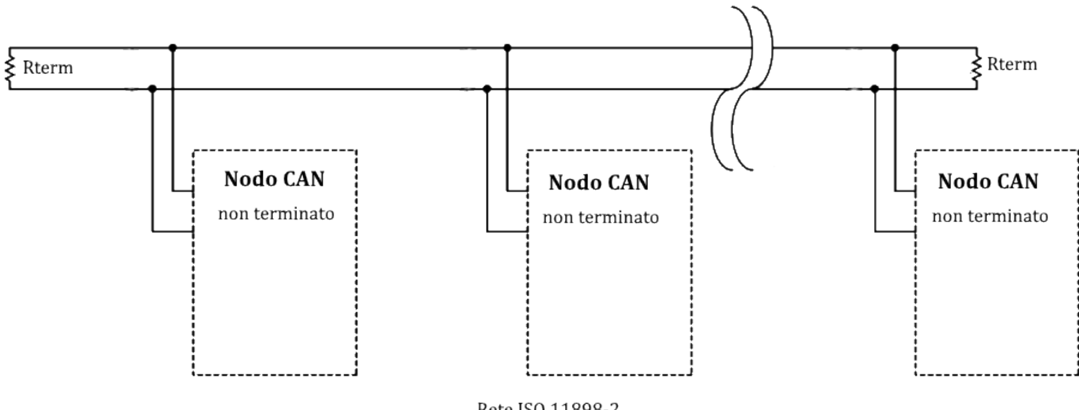


Figura A.1.1: *High Speed* CAN - topologia

Quando il *bus* trasmette il bit recessivo la tensione è 2.5V per entrambi i cavi, mentre con il bit dominante vengono imposti 5V (cavo CAN+) e 0V (cavo CAN-). Il segnale differenziale viene considerato dominante quando supera i 2V. La terminazione passiva permette di ottenere una tensione differenziale nominale di 0V, come si può notare in Figura A.1.2.

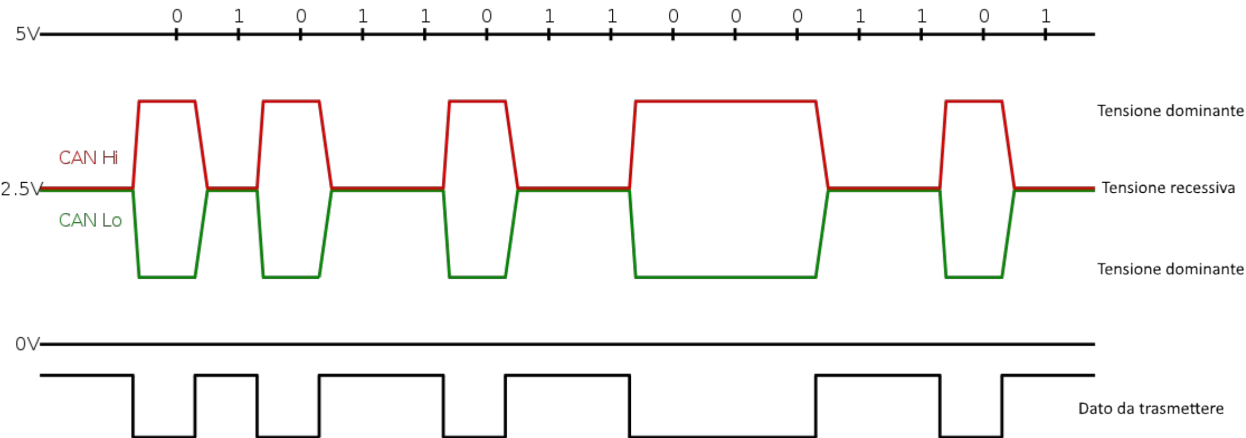


Figura A.1.2: *High Speed* CAN - esempio di trasmissione

Low Speed CAN (ISO 11898-3)

Chiamato anche *fault tolerant* CAN, permette velocità fino a 125 kbit/s e supporta sia la topologia lineare sia quelle a stella o miste. È terminata a ciascun nodo da una frazione della resistenza di terminazione totale. In generale questa non dev'essere minore di 100 Ω.

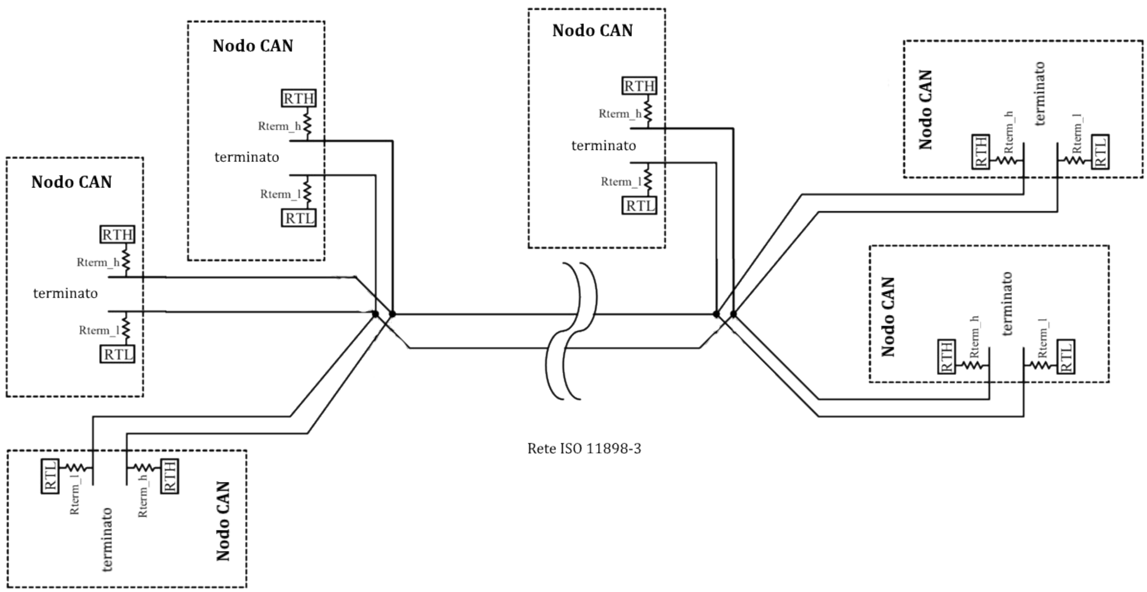


Figura A.1.3: *Low Speed* CAN - topologia

La tensione differenziale associata al segnale dominante deve superare i 2.3V, mentre per il recessivo deve essere inferiore a 0.6V. A differenza del *bus* ad alta velocità, in caso di recessivo la resistenza non è collegata a 2.5V ma al livello

opposto. I cavi ed i dispositivi collegati devono essere in grado di sopportare tensioni nell'intervallo (-27, 40)V senza subire danni.

Figura A.1.4: High Speed CAN - esempio di trasmissione

A.1.2 Trasmissione dati e arbitrato

La trasmissione dati del protocollo CAN usa un'arbitrazione bit a bit *lossless*: questo significa che non c'è ritardo nella trasmissione del messaggio prioritario, mentre il nodo con priorità minore ritenterà la trasmissione quando il *bus* sarà nuovamente libero.

Le specifiche CAN introducono i termini "recessivo" e "dominante", dove dominante è uno ‘0’ logico (imposto dal trasmettitore) ed il recessivo è un ‘1’ logico (generato passivamente da una resistenza di *pull-up* o *pull-down*). Perciò quando viene trasmesso un bit recessivo, e contemporaneamente un altro dispositivo trasmette un bit dominante, si ha una collisione, e solo il bit dominante è visibile in rete (tutte le altre collisioni sono invisibili). Durante la trasmissione, ogni nodo in trasmissione controlla lo stato del *bus* e confronta il bit ricevuto con il bit trasmesso: se un bit dominante è ricevuto mentre un bit recessivo è trasmesso il nodo interrompe la trasmissione (ossia perde l'arbitrato). Questa procedura corrisponde ad un AND logico bit a bit: l’arbitrato del CAN è dunque effettuato logicamente e non temporalmente, come altri standard. Nella Tabella A.1 è mostrato un esempio di arbitrato con tre contendenti.

Tabella A.1: Esempio di arbitrato a tre contendenti

	Start	ID Bits											Altri bit della finestra
	Bit	10	9	8	7	6	5	4	3	2	1	0	
Canale 14	0	0	0	0	0	0	0	0	1	1	1	0	
Canale 15	0	0	0	0	0	0	0	0	1	1	1	1	Interruzione trasmissione
Canale 16	0	0	0	0	0	0	0	1	Interruzione trasmissione				
CAN *bus*	0	0	0	0	0	0	0	0	1	1	1	0	

È quindi molto importante l'univocità dei canali che transitano sullo stesso *bus*, in modo da garantire che al termine della trasmissione degli ID sia rimasto soltanto un dispositivo a trasmettere.

A.1.3 Tipologie di *frame*

Tutti i frame (detti anche "messaggi") incominciano con un bit di "*start-of-frame*" (SOF).
I frame del CAN possono essere di quattro tipi:

- *Data frame*: frame contenente i dati che il nodo trasmette.
- *Remote frame*: frame che richiede la trasmissione di un determinato identificatore.
- *Error frame*: frame trasmesso da un qualsiasi nodo che ha rilevato un errore.
- *Overload frame*: frame che introduce un ritardo fra data frame e/o remote frame.

Data frame

Sono i frame che eseguono l'effettiva trasmissione dei dati. I messaggi possono avere due formati:

- *Base frame format*: con 11 bit di identificazione.
- *Extended frame format*: con 29 bit di identificazione. (introdotto con CAN 2.0b)

Lo standard CAN deve obbligatoriamente riconoscere il formato *base frame* e può opzionalmente riconoscere il formato *extended frame* anche se deve essere comunque tollerato.
Il CAN base permette 2^{11} = 2048 tipi di messaggi diversi, ma da specifiche Bosch se ne possono usare solo 2031: infatti il vincolo imposto al campo dell'identificatore è che i primi 7 bit non possano essere tutti recessivi. Nella versione extended si

possono avere fino a 2^{29} tipi di messaggi. Per la sua struttura un *base frame* ha maggiore priorità (vince l'arbitrato) rispetto ad un *extended frame*

Base frame

Il formato del *base frame* ha la struttura mostrata in Tabella A.2.

Tabella A.2: Struttura del frame di base

Nome del campo	Lunghezza (numero di bit)	Funzione
Start-of-frame	1	Indica l'avvio della sequenza di trasmissione
Identificatore	11	Identificatore (unico) dei dati
Richiesta remota di trasmissione (RTR)	1	Deve essere un bit dominante
Bit aggiuntivo di identificazione (IDE)	1	Deve essere un bit dominante
Bit riservato (r0)	1	Riservato
Codice di lunghezza dati (DLC)	4	Numero di Byte per codificare ciascun dato (0-8 Byte)
Campo dati	0-8 Byte	Dati da trasmettere (la lunghezza è specificata dal campo DLC)
CRC	15	Controllo di parità a ridondanza
delimitatore CRC	1	Deve essere un bit recessivo
Slot ACK	1	Il trasmettitore invia un bit recessivo e ogni ricevitore può confermare la ricezione con un bit dominante
Delimitatore ACK	1	Deve essere un bit recessivo
End-of-frame (EOF)	7	Devono essere bit recessivi

Nel 2012 è stato pubblicato lo *standard* CAN FD [5] che, pur mantenendo la retrocompatibilità con gli standard 2.0, introduce minor latenza per la trasmissione di dati e la possibilità di trasmettere fino a 64 Byte di dati in ogni frame dati, agevolando così la trasmissione di grandi quantità di dati (ad esempio flussi video). Dato che in questo studio lo standard CAN FD non introdurrebbe vantaggi apprezzabili non è stato considerato, anche se è ugualmente inserito negli sviluppi futuri.

Extended frame

Il formato dell'*extended frame* ha la struttura mostrata in Tabella A.3:

Tabella A.3: Struttura del frame esteso

Nome del campo	Lunghezza (numero di bit)	Funzione
Start-of-frame	1	Indica l'avvio della sequenza di trasmissione
Identificatore A	11	Prima parte dell'identificatore (unico) dei dati
Richiesta remota sostitutiva (SRR)	1	Deve essere un bit recessivo
Bit aggiuntivo di identificazione (IDE)	1	Deve essere un bit recessivo
Identificatore B	18	Seconda parte dell'identificatore (unico) dei dati
Richiesta remota di trasmissione (RTR)	1	Deve essere un bit dominante
Bit riservati (r1 & r0)	2	Riservati
Codice di lunghezza dati (DLC)	4	Numero di Byte del dato (0-8 Byte)
Campo dati	0-8 Byte	Dati da trasmettere (lunghezza specificata dal campo DLC)
CRC	15	Controllo di parità a ridondanza
Delimitatore CRC	1	Deve essere un bit recessivo
Slot ACK	1	Il trasmettitore invia un bit recessivo e ogni ricevitore può confermare la ricezione con un bit dominante
Delimitatore ACK	1	Deve essere un bit recessivo
End-of-frame (EOF)	7	Devono essere bit recessivi

I due identificatori (A e B) combinati formano un unico identificatore di 29 bit.

Remote Frame

Il *Remote Frame* è identico al *Data Frame*, eccetto che:

- il bit RTR è posto allo stato di bit recessivo
- il campo lunghezza dati contiene il numero di Byte (relativi al payload del pacchetto) richiesti al data frame.

Error Frame

Il *frame* di errore è composto da due campi:

- il primo è formato dalla combinazione dei *flag* di errore attivati da uno dei nodi collegati alla rete
- il secondo è il cosiddetto “delimitatore di errore” (*Error Delimiter*)

Esistono due tipi di *Error Flag*:

- *Active Error Flag*: trasmessi da un nodo che ha rilevato un errore di rete, e che si trova nello stato di "error active"

- *Passive Error Flag*: trasmessi da un nodo che ha rilevato la presenza sulla rete di un *Active Error Flag*, e che si trova nello stato di "*error passive*"

Overload frame

Il *frame* di *Overload* contiene due campi: *Overload Flag* e *Overload Delimiter*. Esistono due condizioni di *overload* che possono determinare la trasmissione di un *overload flag*:

- Stato del ricevitore, che richiede un ritardo di trasmissione dal successivo *data frame* o *remote frame*
- Viene rilevato un bit dominante durante un intervallo nella trasmissione

Un *overload frame* dovuto al primo caso è consentito solo per essere avviato al momento del primo bit di un intervallo previsto, mentre un *overload frame* dovuto al secondo caso incomincia un bit dopo aver rilevato il bit dominante. L'*Overload Flag* è costituito da sei bit dominanti (tutti pari a zero). La forma complessiva è come quella di un *active error flag*, che azzera i campi intervalli. Conseguentemente, anche gli altri nodi della rete rilevano una condizione di *overload* e trasmettono un *overload flag*. L'*overload Delimiter* è costituito da 8 bit recessivi (tutti pari a uno) e ha la stessa forma di un *Error Delimiter*.

A.1.4 Temporizzazione e *bit stuffing*

Il *bus* CAN ha una comunicazione seriale con codifica di linea NRZ (*non return to zero*) e non dispone, a differenza di altri protocolli di comunicazione, di una linea di trasmissione del *clock* (è quindi un protocollo asincrono). Inizialmente viene effettuata una sincronizzazione tramite il bit di *Start* (transizione recessivo-dominante), mentre successivamente viene utilizzata una continua risincronizzazione (ad ogni transizione recessivo-dominante) per poter aumentare la flessibilità, la resistenza ai disturbi e alla deriva o alla variazione della temporizzazione tra nodi. Se la transizione non avviene nell'istante esatto previsto dal ricevitore, questo aggiusta di conseguenza il tempo di bit nominale. Questo aggiustamento viene effettuato dividendo ogni bit in quanti temporali (il numero varia a seconda del controllore del *bus*) ed assegnando un numero di quanti ad ognuno dei quattro segmenti temporali del bit: sincronizzazione, propagazione e due segmenti di fase.

Un esempio è fornito in Figura A.1.5

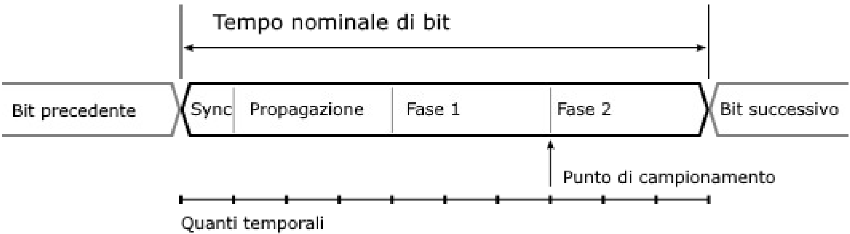


Figura A.1.5: Esempio di divisione temporale del bit in 10 quanti

Il controllore può dunque modificare la durata della prima o della seconda fase . Questo permette al dispositivo che perde l'arbitrato di ottenere la sincronizzazione con il dispositivo vincitore. Per assicurare transizioni sufficienti per mantenere la sincronizzazione viene utilizzata una tecnica chiamata *Bit Stuffing*, mostrata in Figura A.1.6 e consiste nell'introduzione di un bit di sincronizzazione (di polarità opposta) dopo cinque bit della stessa polarità, contando anche eventuali precedenti bit di *stuffing*. Questa pratica è utilizzata in tutti i campi ad eccezione del CRC, ACK e EOF. In ogni caso il ricevitore decodifica e rimuove automaticamente i bit aggiuntivi. Il ricevimento di sei bit della stessa polarità nei campi dove non è permesso è considerato un errore.

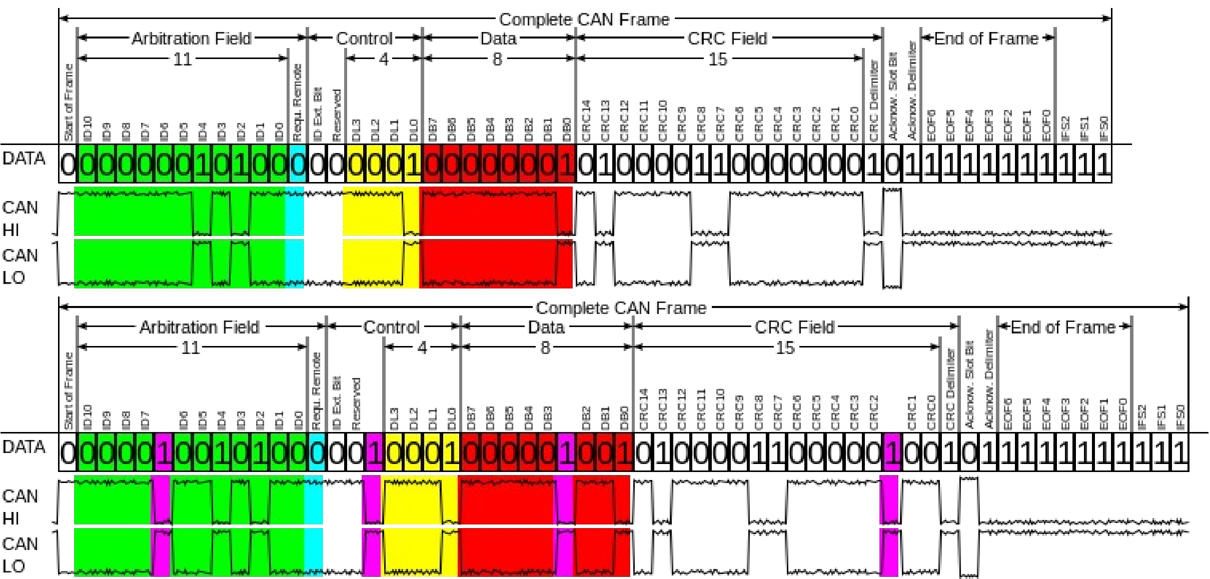


Figura A.1.6: Esempio di messaggio prima e dopo il bit *stuffing*

Il bit *stuffing*, necessario a causa della codifica NRZ, aumenta l'overhead di comunicazione. In particolare nel caso peggiore è inserito un bit ogni quattro

(contando il primo dei cinque bit come un precedente bit di stuffing) e, data la particolare struttura dei bit dell'header, solo

34 su 44 ne risultano sottoponibili. Perciò, per un messaggio originale di $8n+44bit$ dove n è il numero di Byte dati trasmessi e 44 è il numero di bit di overhead, si possono calcolare i bit totali usando la

$$8n+44+\left[\frac{34+8n-1}{4}\right]bit$$

Formula 3: Lunghezza di un frame CAN con bit stuffing

Una discussione analoga si applica nel caso di indirizzamento a 29 bit.
In Tabella A.4 si possono osservare le velocità massime, che rappresentano il caso peggiore in caso di acquisizione, e le velocità minime garantite, da considerare in caso di trasmissione dati.

Tabella A.4: Velocità del *bus* CAN 2.0A

Byte Dati/ frame	bit/ frame (min)	bit/ frame (max)	frame/s (max)	frame/s (min)	Byte/s (max)	Byte/s (min)	frame/s (max)	frame/s (min)	Byte/s (max)	Byte/s (min)
1	55	65	18182	15385	18182	15385	2273	1923	2273	1923
2	63	75	15873	13333	31746	26667	1984	1667	3968	3333
3	71	85	14085	11765	42254	35294	1761	1471	5282	4412
4	79	95	12658	10526	50633	42105	1582	1316	6329	5263
5	87	105	11494	9524	57471	47619	1437	1190	7184	5952
6	95	115	10526	8696	63158	52174	1316	1087	7895	6522
7	103	125	9709	8000	67961	56000	1214	1000	8495	7000
8	111	135	9009	7407	72072	59259	1126	926	9009	7407

Nel calcolo è stato considerato anche l'*interframe* (3 bit recessivi) posto prima di ogni finestra dati.

A.2 USB

Acronimo di *Universal Serial bus,* è un'interfaccia di comunicazione seriale, altamente diffusa ed utilizzata, sviluppata per creare uno standard unico di comunicazione fra periferiche elettroniche. Rispetto alle interfacce esistenti in

precedenza, che sono state praticamente rimpiazzate dallo standard, è compatto, ro*bus*to e permette alte velocità di comunicazione. In più supporta la funzionalità *Plug and Play* consentendo così di collegare e scollegare periferiche senza dover riavviare l'*host* (tipicamente un computer).
Essendo un'interfaccia estremamente flessibile è anche molto complessa ed articolata. In seguito verranno descritte solamente le funzionalità relative alle finalità di questo elaborato, in particolare ciò che riguarda il trasferimento dei dati da un microcontrollore ad una memoria di massa. Per informazioni dettagliate e ufficiali si rimanda invece al sito dell'organizzazione che sviluppa lo standard [6].

A.2.1 Architettura fisica

Il sistema USB è asimmetrico: consiste in un singolo gestore e molte periferiche collegate ad albero, attraverso dei dispositivi chiamati *hub*. Supporta fino a un massimo di 127 periferiche per gestore, ma nel computo vanno inclusi anche gli *hub* e il gestore stesso, quindi, in realtà, il numero totale di dispositivi collegabili è inferiore. Per la comunicazione sono utilizzati due cavi, indentificati con le sigle D+ e D-, sui quali vengono trasmessi segnali differenziali. Inoltre sono previsti due segnali di modo comune, il livello alto SE1 ed il livello basso SE0. Mentre il primo non dovrebbe mai verificarsi, il secondo è utilizzato per segnalare la disconnessione, il reset, ed il termine dei pacchetti.
Un cavo USB può essere lungo al massimo 5 m: per lunghezze superiori è necessario ricorrere a uno o più *hub* attivi che amplifichino il segnale, con il limite di cinque in cascata.

A.2.2 Trasmissione dei dati ed *endpoint*

Il trasferimento di dati avviene attraverso una serie di eventi chiamati transazioni. Le transazioni avvengono all'interno di intervalli temporali (chiamati frame) gestiti dall'host e sono divise in tre parti:

- Token: inviato dall'host, contiene tutti i parametri della transazione
- Dati: è gestito dall'host per un trasferimento OUT, dallo slave per un trasferimento IN. È una parte opzionale
- Handshake: Stato della richiesta, è la risposta di chi ha ricevuto i dati

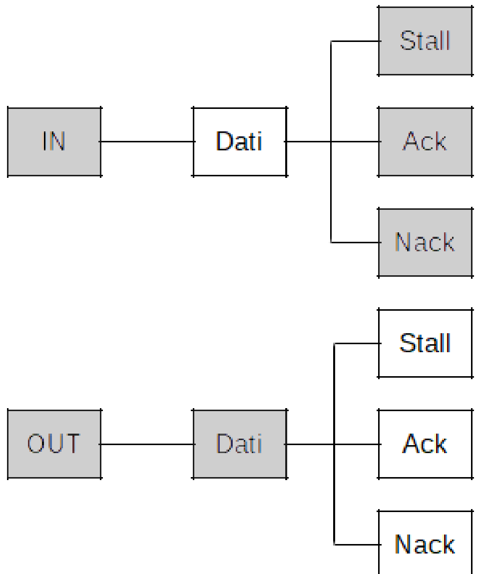


Figura A.2.1: Transazioni USB, IN ed OUT

Nella Figura A.2.1 sono mostrate separatamente le varie parti di una transazione, con sfondo grigio se controllate dall'*host*, bianco viceversa. Si evidenza che, in una transazione avvenuta correttamente, l'*handshake* contiene il messaggio ACK (*acknowledgement*). Esistono tre tipi di transazioni, utilizzate per scopi diversi:

- *Setup*: l'*host* invia le richieste al *device*.
- *Data*: contiene le informazioni relative alla richiesta precedente. Possono susseguirsi più transazioni IN o OUT di questo tipo
- *Status*: Fornisce il risultato della richiesta fatta dall'host

Ogni *frame*, descritto in seguito, utilizza diverse combinazioni e numero di queste transazioni. In Figura A.2.2 è mostrata la struttura di un trasferimento di controllo.

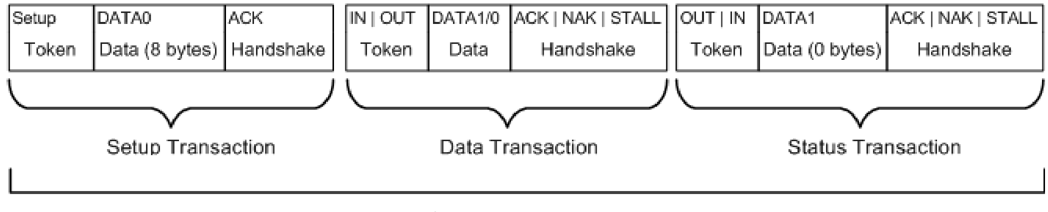


Figura A.2.2: Esempio di un trasferimento di controllo

Su un *bus* USB possono essere collegati fisicamente solo due dispositivi, un *host* ed un *device*. Nonostante ciò possono esistere più canali virtuali di comunicazione, chiamati *pipe*. Ogni *pipe* collega due *endpoint,* uno per dispositivo. Ad ogni *endpoint* è associato un tipo di trasferimento ed una direzione (IN o OUT). Ogni dispositivo può implementare un numero arbitrario di *endpoint*, per un massimo di 32 (16 IN e 16 OUT). In aggiunta a questi dev'essere sempre presente l'*endpoin*t 0, che ha la particolarità di permettere entrambe le direzioni di trasferimento. Per le comunicazioni di controllo viene creato un *pipe* tra i rispettivi *endpoint* 0 (che sono implementati obbligatoriamente) in cui vengono inviati i relativi *frame,* di controllo.

Nel protocollo USB è presente il *bit stuffing*, essendo anch'esso come il CAN un protocollo seriale asincrono. La sua codifica deriva dal NRZ, il NRZI (*non return to zero inversion*). L'unica differenza è che nel secondo uno '0' è rappresentato dal cambio del livello della linea, e viceversa. Il bit stuffing avviene inserendo un cambio di livello (logicamente sarebbe uno '0') dopo il sesto bit consecutivo dello stesso livello (cioè dopo sei '1').

A.2.3 Tipologie trasferimento e *frame*

I *frame* sono divisioni temporali gestite dall'*host*. Esistono diverse tipologie di trasferimento.

- *Controllo*: viene usato per trasferire informazioni relative alla configurazione dello *slave* e avviene sempre attraverso l'*endpoint* 0. È supportato in tutte le versioni (Tabella A.6).
- *Bulk*: viene usato per trasferire grandi quantitativi di dati sequenzialmente. Non ha una banda minima garantita, ma utilizza il *bus* nei *frame* non utilizzati da altri traferimenti. È unidirezionale e non viene supportato nella modalità *Low-Speed* .
- *Isocronous*: viene usato per trasferire dati con cadenza costante, come audio o video. È garantita una banda minima ma non un controllo di correttezza, quindi alcuni dati potrebbero essere persi o corrotti. Per risparmiare banda non viene trasmesso l'handshake. È unidirezionale e non viene supportato nella modalità *Low-Speed* .
- *Interrupt:*Nei *frame* di questo tipo l'*host* interroga i vari dispositivi per verificare se qualcuno di essi deve traferire dei dati. Queste richieste vengono

effettuate regolarmente nel tempo. Questo trasferimento è usato anche per trasferire dati in modo periodico: se il dispositivo non è pronto risponderà con un handshake NACK

Nella Tabella A.5 vengono comparate queste tipologie nel caso dell'*High-Speed* USB, che ha *frame* di durata 125 μs, mentre nelle versioni più lente è di 1 ms.

Tabella A.5: Comparazione trasferimenti USB *High-Speed*

Tipologie di trasferimento	Dimensione massima del trasferimento	Trasferimenti per *frame*	Velocità massima di trasferimento (teorica)
Controllo	64 Byte	1	64 kByte/s
Interrupt	1024 Byte	Fino a 3	24 MByte/s
Bulk	512 Byte	Fino a 13	53 MByte/s
Isochronous	1024 Byte	Fino a 3	24 MByte/s

A.2.4 OTG

USB *On-The-Go* è una specifica che permette a un qualsiasi dispositivo che la implementa di agire come *host* (*master*) e quindi di comunicare con altre periferiche USB, oltre che come *peripheral* (*slave*). A differenza dell'USB standard possono essere collegati solo dispositivi che non richiedono *driver*, gestitbili tramite il protocollo *Host Negotiation Protocol* (HNP): infatti non sono supportati alcuni protocolli come *Enhanced Host Controller Interface* (EHCI), *Open Host Controller Interface* (OHCI), *Universal Host Controller Interface* (UHCI).

In particolare le funzionalità da aggiungere rispetto ad un normale dispositivo *slave* sono:

- invio di pacchetti SOF (*Start of Frame*)
- invio di pacchetti SETUP, IN, and OUT
- Pianificazione dei trasferimenti nell'arco temporale di una finestra
- Segnale di *reset*
- Fornire l'alimentazione al dispositivo *slave*

La massima velocità di trasferimento di ogni versione è la stessa dell'USB tradizionale. In Tabella A.6 è inserita a titolo comparativo anche la velocità reale, che tiene in considerazione sia l'*overhead* dei vari segnali di controllo sia il *bit stuffing*.

Tabella A.6: Velocità di trasmissione delle versioni USB

Nome	Versione	Velocità teorica	Velocità reale
Low-Speed	USB 1.0	1,5 Mbit/s (187,5 kByte/s)	1 Mbit/s (125 kByte/s)
Full-Speed	USB 1.1	12 Mbit/s (1,5 MByte/s)	7 Mbit/s (875 kByte/s)
High-Speed	USB 2.0	480 Mbit/s (60 MByte/s)	280 Mbit/s (35 MByte/s)
Super-Speed	USB 3.0	4,8 Gbit/s (600 MByte/s)	3,2 Gbit/s (400 MByte/s)

Nell'elaborato sono stati considerati solamente gli standard fino all'USB 2.0, che è largamente diffuso e più che sufficiente per trasferire le informazioni provenienti dal *bus* CAN, oltre ad alcuni dati aggiuntivi. Inoltre, tipicamente, la velocità di trasferimento non è limitata dallo standard ma dalla memoria di massa.

APPENDICE B
FILE SYSTEM FAT

Un *file system* è una struttura di organizzazione e posizionamento dei *file* su un dispositivo di archiviazione. In questa appendice viene descritto il FAT (*file Allocation Table*) che è un *file system* largamente diffuso, anche grazie al fatto di essere stato adottato ufficialmente dai sistemi operativi Microsoft®, azienda che ne ha curato lo sviluppo.

Nel tempo è stato espanso per supportare dimensioni sempre maggiori. Esistono ad oggi quattro versioni: FAT12, FAT16, FAT32, ExFAT (conosciuto anche come FAT64).

La principale differenza fra le versioni è la dimensione, in bit, delle voci presenti nella struttura del FAT. Questa dimensione, collegata al numero massimo di *cluster*, corrisponde al numero presente nel nome e, nelle ultime due versioni, è anche il numero massimo di indirizzamento al Byte per ogni *file*. Dunque un *file* nel *file system* FAT32 può avere al massimo 2^{32} Byte mentre nel ExFat 2^{64}, valori riportati anche in Tabella B.1.

Tabella B.1: Confronto dei *file system* FAT

	Dimensione partizione	**Numero di *file***	**Dimensione *file***	**Numero di Cluster**	**Lunghezza del nome**
exFAT	128PByte	2796202	16 EByte	4294967295	255
FAT32	32GByte	4194304	4 GByte	4177918	255
FAT16	2GByte	65536	2GByte	65520	11-255
FAT12	16MByte	-	16MByte	4080	11-254

B.1 Organizzazione

Generalmente il primo settore di un dispositivo di memorizzazione contiene una tabella delle partizioni, in cui è possibile trovare le informazioni relative alle partizioni presenti. Il più diffuso è il *Master Boot Record* (MBR), che è diviso in due sezioni. La prima, *Master Boot Program* (MBP) consiste in 446 Byte di codice eseguibile mentre la seconda, *Master Boot Table* (MBT), è una tabella di 64 Byte che può contenere fino a quattro voci di partizione. In particolare viene indicata la loro tipologia, il settore di inizio e di fine, la dimensione ed il numero di *cluster* contenuti. Negli ultimi 2 Byte è presente un valore fisso, usato come delimitatore.
A sua volta una partizione FAT è divisa in tre sezioni: la prima contiene i parametri relativi al proprio funzionamento, la seconda la tabella di allocazione vera e propria, la terza i *file*.

Boot sector

È la prima struttura di una partizione FAT e ne occupa sempre il primo settore. È anche chiamata *BIOS Parameter Block* (BPB). Per una descrizione completa dei campi si rimanda al riferimento bibliografico [9], mentre in Tabella A.5 sono mostrati i campi di nostro interesse.

Tabella B.2: Campi di interesse - Boot Sector

Campo	Nome	Offset	Dimensione	Valore
Bytes Per Settore	BPB_BytsPerSec	0x0B	16 Bit	Solitamente 512 (Byte)
Settori Per Cluster	BPB_SecPerClus	0x0D	8 Bit	1,2,4,8,16,32,64,128
Numero di Settori Riservati	BPB_RsvdSecCnt	0x0E	16 Bit	Solitamente 0x20
Numero di tabelle FAT	BPB_NumFATs	0x10	8 Bit	Solitamente 2
Settori Per FAT	BPB_FATSz32	0x24	32 Bit	Dipendente dalle dimensioni del disco
Primo Cluster di Radice degli Indici	BPB_RootClus	0x2C	32 Bit	Solitamente 0x00000002
Delimitatore	(none)	0x1FE	16 Bit	Sempre 0xAA55

Tabella di Allocazione file

Posta dopo il *Boot Sector*, la FAT contiene in forma di lista concatenata le locazioni dei *cluster* relative ai vari *file*. Un cluster è un'unità di allocazione multipla del settore, esattamente del valore indicato nel campo *BPB_SecPerClus* del *Boot Sector*. Ogni voce della tabella contiene il valore '0' se il relativo *cluster* è disponibile, il valore esadecimale '0xFFFFFFFF' se il relativo *cluster* è l'ultimo di un *file*, oppure il valore del *cluster* successivo relativo al *file* stesso.
Viene riportato in Tabella B.3 un esempio completo di allocazione di FAT32, con due *file* (A e B) , un *file* di indice, spazio disponibile e frammentazione.

Tabella B.3: Esempio di *file* Allocation Table

Indice	Valore	Significato	*file*	Progressivo
0x00000000	XXXXXXXX	Valore riservato	-	-
0x00000001	XXXXXXXX	Valore riservato	-	-
0x00000002	0x00000013	Prossimo cluster del *file*	indice	1
0x00000003	0x00000004	Prossimo cluster del *file*	A	1
0x00000004	0x00000005	Prossimo cluster del *file*	A	2
0x00000005	0x00000006	Prossimo cluster del *file*	A	3
0x00000006	0x0000000C	Prossimo cluster del *file*	A	4
0x00000007	0x0000000B	Prossimo cluster del *file*	B	1
0x00000008	0xFFFFFFFF	Fine del *file*	B	5
0x00000009	0x00000008	Prossimo cluster del *file*	B	4
0x0000000A	0x00000009	Prossimo cluster del *file*	B	3
0x0000000B	0x0000000A	Prossimo cluster del *file*	B	2
0x0000000C	0x0000000D	Prossimo cluster del *file*	A	5
0x0000000D	0xFFFFFFFF	Fine del *file*	A	6
0x0000000E	0x00000000	Cluster libero	-	-
0x0000000F	0x00000000	Cluster libero	-	-
0x00000010	0x00000000	Cluster libero	-	-
0x00000011	0x00000000	Cluster libero	-	-
0x00000012	0x00000000	Cluster libero	-	-
0x00000013	0xFFFFFFFF	Fine del *file*	indice	2
……….	…………	…………	…………	…………

In particolare il *file* A inizia nel quarto *cluster*: nella posizione 4 della tabella è indicato il *cluster* successivo, cioè il secondo del *file* (posizione 5). Nella posizione 5

della tabella è presente l'indirizzo del terzo *cluster* del *file* e così via. Nell'esempio i *cluster* del *file* A sono consecutivi fino all'indice 6, che poi salta in posizione 13. Questa non sequenzialità del *file* viene chiamata frammentazione. Il *file* B è estremamente frammentato pur essendo contiguo, in quanto per recuperarlo vengono richiesti indirizzi in ordine non sequenziale.

Si ricorda che ogni voce della tabella ha dimensione in bit pari al numero presente nel nome. Quindi nell'esempio riportato, riferito al FAT 32, l'indice riporta posizioni distanti l'una dall'altre 4 Byte.
Solitamente sono presenti, per sicurezza, due FAT con contenuto identico. Questo è indispensabile perché in esse sono contenute le informazioni sulla struttura dei dati memorizzati nell'ultima sezione della partizione.

Tabella di Allocazione file

È stato spiegato quindi come recuperare il contenuto di un *file* partendo dalla conoscenza del primo *cluster*. Esiste una struttura, chiamata radice degli indici, che contiene questa ed altre informazioni come nome del *file*, dimensione, data di creazione, modifica, ultimo utilizzo ecc. Allo stesso modo sono memorizzate anche le informazioni sulle cartelle.
Questa struttura dalla versione FAT32 è collocata nello spazio dati e segue le regole di un normale *file*. In particolare questa scelta permette di non avere limiti sulla quantità di *file* e cartelle memorizzabili.
Questa sezione contiene i dati veri e propri. In particolare, come indicato in precedenza, può essere presente anche la radice degli indici. Questa ha la struttura riportata in Tabella B.4, in cui ogni voce occupa uno spazio di 32 Byte. Perciò ogni settore può contenere informazioni di 16 tra *file* e cartelle.

Tabella B.4: Struttura di una voce della radice degli indici

Campo	Nome	Offset	Dimensione
Nome	DIR_Name	0x00	11 Byte
Attributi	DIR_Attr	0x0B	1 Byte
Riservato	DIR_NTRes	0x0C	1 Byte
Creazione	DIR_CrtTimeTenth	0x0D	1 Byte
Primo cluster (Prima parte)	DIR_FstClusHI	0x14	2 Byte

Ultima modifica	DIR_WrtTime	0x16	2 Byte
	DIR_WrtDate	0x18	2 Byte
Primo cluster (Seconda parte)	DIR_FstClusLO	0x1A	2 Byte
Dimensione	DIR_*file*Size	0x1C	4 Byte

B.2 Utilizzo

In questo paragrafo presentate le procedure da implementare per alcuni utilizzi tipici del *file system*. Dove significativo verrà fatto tiferimento ai dati della Tabella B.3: per brevità verrà utilizzato il termine *indice* per indicare una voce alla rispettiva posizione della FAT ed il termine *cluster* per riferirsi al relativo contenuto localizzato nello spazio dei dati. Si supponga inoltre di avere cluster di dimensione 2048 Byte, cioè 4 settori ciascuno.

Inizializzazione

Anche detta montaggio, dall'inglese *mount*, è la procedura di preparazione di un dispositivo che si rende necessaria all'avvio. Successivamente il *file system* è disponibile per ulteriori operazioni.

1. Lettura del *Master Boot Table*
2. Scelta della partizione (se più di una)
3. Lettura del *BIOS Parameter Block*
4. Verifica di congruenza della dimensione della partizione
5. Verifica di congruenza delle copie del FAT

Lettura di un file

Si supponga di voler leggere il *file* "B" fino al terzo *cluster*, avendo a disposizione solamente il percorso, cioè una stringa letterale che indichi il nome del *file* e le cartelle in cui è contenuto. Inoltre, in modo arbitrario, si ipotizzi che la voce relativa al *file* sia nel secondo *cluster* della radice degli indici.

1. Lettura del cluster 2 (radice degli indici)
2. Ricerca del *file* "B" nelle 64 voci di indice contenute nel cluster
3. Lettura dell'indice 0x00000002 (si ottiene 0x00000013)
4. Lettura del cluster 13
5. Ricerca del *file* "B" nelle voci del cluster (si ottiene 0x00000007)
6. Lettura del cluster 7

```
 7. Lettura dell’indice 0x00000007 (si ottiene 0x0000000B)
 8. Lettura del cluster 11
 9. Lettura dell’indice 0x0000000B (si ottiene 0x0000000A)
10. Lettura del cluster 10
```

In totale si possono contare 8 operazioni di lettura distinte e non ottimizzabili: infatti si ricorda che il *file* è frammentato, per cui non può riutilizzare più volte una stessa lettura.

Creazione di un file

Si supponga di voler creare un *file* vuoto “C” nella cartella principale.

```
1. Lettura dell’indice 0x00000002 (si ottiene 0x00000013)
2. Lettura dell’indice 0x00000013 (si ottiene 0xFFFFFFFF)
3. Ricerca di un indice libero (il primo è 0x0000000E)
4. Scrittura di 0xFFFFFFFF nell’indice 0x0000000E
5. Aggiunta di una nuova di file voce nel cluster 19 (0x00000013)
   Se pieno si prosegue col punto successivo
6. Ricerca di un indice libero (il primo è 0x0000000F)
7. Scrittura di 0x0000000F nell’indice 0x000000013
8. Scrittura di 0xFFFFFFFF nell’indice 0x0000000F
9. Aggiunta di una nuova di file voce nel cluster 15
```

Espansione di un file

Si supponga di voler espandere il *file* “A” e che il relativo indice sia nel primo *cluster* degli indici.

```
1. Lettura del cluster 2 (radice degli indici)
2. Ricerca del file “A” nelle 64 voci di indice contenute nel cluster
3. Lettura del’indice 0x00000003 (si ottiene 0x00000004)
4. Ripetere il 3. fino al raggiungimento del valore 0xFFFFFFFF (fine)
5. Ricerca di un indice libero (il primo è 0x0000000E)
6. Scrittura di 0xFFFFFFFF nell’indice 0x0000000E
7. Scrittura di 0x0000000E nell’indice 0x0000000D, precedente fine del file
```

		0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
MBR	0x00000000	codice d'avvio															
	....																
	0x0000001A																
	0x0000001B																
	0x0000001C	Voce di partizione 1															
	0x0000001D	Voce di partizione 2															
	0x0000001E	Voce di partizione 3															
	0x0000001F	Voce di partizione 4														55	AA
BPB	0x00000020												BPS		SPC	RES	
	0x00000021	n°FAT															
	....																
	0x0000003F															55	AA
FAT	0x00000040	Indice 1				Indice 2				Indice 3				Indice 4			
	....	FAT 1															
	0x0000430F																
	0x00004040	Indice 1				Indice 2				Indice 3				Indice 4			
	....	FAT 2															
	0x0000830F																
Dati	0x00008040	primo cluster (2048 Byte)															
	....																
	0x00008839																
	0x00008840																
	....																

Figura B.2.1: Riepilogo grafico della struttura di una memoria con *file system* FAT32

Riferimenti bibliografici

[1] Bonnick A. "*automotive* Computer Controlled *system*s",Butterworth-Heinemann, 2000, ISBN: 978-0-7506508-9-2, pag. 130-137; 179-186

[2] Boatright R. · Correa C. · Kozierok C. · Quesnelle J. "*automotive* Ethernet - The Definitive Guide", 2014, ISBN: 978-0-9905388-0-6

[3] Lee W. · Han I. "Development and test of a motor vehicle event data recorder", 2004

[4] Robert Bosch GmbH "CAN Specification Version 2.0", 1991

[5] Robert Bosch GmbH "CAN FD Specification Version 1.0", 2012

[6] Documentazione USB, https://www.usb.org/documents (11 dicembre 2018)

[7] Sear J. "The ARL 'Black Box' Flight Recorder - Invention and Memory", The University of Melbourne, 2001

[8] Davis R. · Burns A.· Bril R. · Lukkien J. "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised" Springer Science + *bus*iness Media, 2007

[9] "Microsoft Extensible Firmware Initiative FAT32 *file system* Specification Version 1.03", Microsoft Corporation, 2000

[10]Axelson J. "USB Mass Storage: Designing and Programming Devices and Embedded Hosts", lakeview research llc, 2006, ISBN: 978-1-931448-04-8