ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI UN MODULO DI ACQUISIZIONE DATI AD ELEVATO THROUGHPUT DA AMPLIFICATORI PER ELETTROFISIOLOGIA

Relazione finale in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Presentata da
MATTEO MARRA

Prima Sessione di Laurea
Anno Accademico 2014 – 2015

PAROLE CHIAVE

Elements

High Throughput Data

Elettrofisiologia

Elaborazione di dati

Analisi Real Time

"And everything under the sun is in tune,
but the sun is eclipsed by the moon"

Indice

Introduzione

Lo scopo dell'elaborato di tesi è la progettazione e lo sviluppo di alcuni moduli di un software per la lettura ad elevato throughput di dati da particolari dispositivi per elettrofisiologia sviluppati dall'azienda Elements s.r.l. Elements produce amplificatori ad alta precisione per elettrofisiologia, in grado di misurare correnti a bassa intensità prodotte dai canali ionici.

Dato il grande sviluppo che l'azienda sta avendo, e vista la previsione di introdurre sul mercato nuovi dispositivi con precisione e funzionalità sempre migliori, Elements ha espresso l'esigenza di un sistema software che fosse in grado di supportare al meglio i dispositivi già prodotti, e, soprattutto, prevedere il supporto dei nuovi, con prestazioni molto migliori del software già sviluppato da loro per la lettura dei dati.

Questo, infatti, presenta vari problemi prestazionali e di perdita di dati con la versione più avanzata dei dispositivi attuali, e inoltre non è facilmente adattabile a nuovi dispositivi e moduli.

Il software richiesto deve fornire una interfaccia grafica che, comunicando con il dispositivo tramite USB per leggere dati da questo, provvede a mostrarli a schermo e permette di registrarli ed effettuare basilari operazioni di analisi.

In questa tesi verranno esposte analisi, progettazione e sviluppo dei moduli di software che si interfacciano direttamente con il dispositivo, quindi dei moduli di rilevamento, connessione, acquisizione ed elaborazione dati.

La sfida più importante era sicuramente quella di garantire la gestione di grandi throughput di dati, visto che i diversi dispositivi effettuano letture con frequenze di campionamento fino a duecento kHz e con un alto parallelismo. L'elaborazione di questa elevata quantità di dati, infatti, portava nel precedente software ad un sensibile rallentamento dell'interfaccia grafica, e quindi della resa dell'applicazione all'utente finale.

Il sistema doveva essere in grado di supportare, in normali condizioni d'uso, una bitrate di circa 12 Mbit/s per il dispositivo ad un canale, fino ai 30 Mbit/s

di quello a quattro canali e prevedere il supporto per una bitrate di almeno 96 Mbit/s, prevista nel nuovo dispositivo a sedici canali.

Il software che è stato progettato e sviluppato nell'ambito di questa tesi doveva quindi essere pronto al previsto ampliamento del numero di dispositivi prodotti, cercando di evitare la re-implementazione di parti fondamentali di software.

Fin dall'inizio, inoltre, è stato individuato che sarebbe stato molto utile aggiungere alle funzionalità già esistenti nel vecchio software la possibilità di effettuare alcuni tipi di analisi dei dati in tempo reale. Questo ha portato quindi ad una definizione modulare dell'applicazione, che permettesse in modo semplice l'aggiunta futura di altri moduli, fornendo una massima possibilità di ampliamento.

Una delle difficoltà principali è stata quella di astrarre in modo ottimale il problema in un paradigma di programmazione ad oggetti, in modo da renderlo totalmente scalabile e affidabile. Inoltre è stato problematico interfacciarsi con i diversi dispositivi, prevedendone altri non ancora disponibili. Tutta la gestione dei differenti protocolli di comunicazione con il dispositivo è stata cruciale per l'affidabilità e la performance dell'applicazione.

Bisognava limitare al minimo la perdita di dati, creando un sistema veloce ma allo stesso tempo sicuro. Inoltre bisognava gestire in modo ottimale il protocollo di comunicazione verso il dispositivo, in modo da inviare a questo diverse configurazioni in tempi brevi, rendendo così l'applicazione poco sensibile a ritardi. Per questi motivi prestazionali, il sistema è stato sviluppato con il linguaggio di programmazione C++ e attraverso l'uso delle librerie *Qt*, utilizzando il driver FTDI, produttore del chip di comunicazione presente sulle diverse versioni dei dispositivi che controlla il canale USB.

Per ovviare soprattutto ai problemi prestazionali è stato fatto uso della tecnologia multithread, ormai largamente diffusa in tutti i sistemi operativi. Questa ha permesso di eseguire ogni modulo separatamente e in modo concorrente, così da produrre ed utilizzare i dati più velocemente possibile. Sono state inoltre molto utilizzate le strutture di base del C++, come puntatori e struct, che hanno permesso di costruire un software ottimizzato nelle particolari operazioni di lettura dei dati.

Per lo scambio di informazioni tra le diverse entità sono state utilizzate le funzionalità di *signal* e *slot* delle librerie *Qt*. Non fornendo però queste ottime prestazioni, nelle parti più delicate del programma, come il trasferimento dei dati tra il modulo di lettura e quello di elaborazione, è stato deciso di

implementare dei buffer appositi, gestiti con la mutua esclusione per evitare problemi di concorrenza.

Il sistema sviluppato soddisfa tutti i requisiti richiesti dall'azienda nella fase iniziale, riuscendo a gestire senza alcun problema prestazionale tutti i dispositivi finora prodotti. È stato oltretutto previsto, attraverso delle particolari simulazioni software, che questo dovrebbe supportare senza alcun problema i nuovi dispositivi in fase di prototipazione. I dati riescono ad essere visualizzati, memorizzati ed analizzati alle bitrate previste con rallentamenti minimi ed una grande responsività dell'interfaccia grafica.

Il progetto è stato sviluppato in team con il mio collega Bajram Hushi, suddividendo attentamente i moduli tra i due. Questo elaborato di tesi tratta quindi dello sviluppo dei moduli di connessione, acquisizione ed elaborazione dati e della rilevazione del dispositivo. Il mio collega si è occupato invece di realizzare i moduli di visualizzazione, memorizzazione ed analisi.

La tesi è suddivisa in sette capitoli.

Nel primo verrà fatta una piccola introduzione ai dispositivi utilizzati da Elements.

Il secondo capitolo espone le tecnologie utilizzate dai dispositivi e nello sviluppo del software.

Il terzo capitolo include tutta l'analisi dei requisiti, dove sono espressi in termini specifici tutti i vincoli che l'applicazione deve soddisfare per essere sviluppata in modo ottimale.

Il quarto capitolo tratta l'analisi del problema, definendo le operazioni principali e dimostrando, attraverso diversi tipi di diagrammi, come questi sono stati modellati.

La progettazione è descritta nel quinto capitolo, dove viene esposta la struttura finale dei moduli sviluppati.

Il sesto capitolo contiene i più importanti dettagli implementativi dei moduli sviluppati, focalizzandosi leggermente su alcune parti di codice molto importanti nel progetto.

L'ultimo capitolo include tutte le operazioni di testing e valutazione prestazionale dell'applicazione.

Segue poi una breve conclusione.

Capitolo 1

I dispositivi Elements e l'applicazione in elettrofisiologia

I dispositivi prodotti da Elements s.r.l sono gli strumenti di misura per i quali è stata realizzata una nuova versione di software di visualizzazione. Questo capitolo ne descrive la struttura e gli utilizzi principali.

1.1 Elements s.r.l. e i suoi dispositivi

1.1.1 Elements

Elements è un'azienda che progetta, realizza e sviluppa strumentazione di misura miniaturizzata ad elevata sensibilità, basata su avanzate tecnologie microelettroniche proprietarie. Si inserisce nel panorama delle startUp italiane in continua evoluzione, e fornisce ai suoi clienti sistemi chiavi in mano nell'ambito dell'elettrofisiologia, costituiti da strumenti di misura, software di utilizzo e chip microfluidici consumabili.

I numerosi anni di ricerca nel campo dello sviluppo microelettronico applicato a sensori bio-sensibili dà al team di Elements un vantaggio unico nell'analisi di segnali a bassa ampiezza. Elements può inoltre vantare diverse collaborazioni con molti centri di ricerca universitari e aziende europee e statunitensi.

1.1.2 Descrizione dispositivi

I dispositivi prodotti da Elements s.r.l sono amplificatori di corrente per nanopori con applicazioni elettrofisiologiche. Si tratta di dispositivi molto piccoli, con una dimensione di 30x15x74mm, che al loro interno racchiudono un amplificatore a basso rumore, un digitalizzatore e dei filtri. Contengono inoltre un generatore di stimoli di tensione, ed il tutto è completamente alimentato da un unico cavo USB.

eONE ed *eFOUR*

I dispositivi sono prodotti in due versioni principali *eONE* ed *eFOUR*, che a loro volta presentano delle sotto-versioni con diverse funzionalità. Il dispositivo *eONE* misura un singolo canale alla volta, mentre *eFOUR* consente la misura simultanea di quattro canali.

(a) Un dispositivo *eONE* (b) Un dispositivo *eFOUR*

Figura 1.1: Panoramica dei dispositivi prodotti da Elements s.r.l.

eSIXTEEN

Esiste un terzo dispositivo, ancora in fase prototipale, chiamato *eSIXTEEN*, che permette la misura in parallelo su sedici canali, mantenendo struttura e dimensioni simili ai già citati *eONE* ed *eFOUR*.

1.2 Struttura dei dispositivi

Si tratta di un sistema multi-livello a tecnologia ibrida, in cui una singola piattaforma incorpora tutto il necessario per effettuare test specifici: [1]

1. Il primo livello è dedicato alla microelettronica, dove è posto un front-end di acquisizione low-noise, in grado di leggere correnti nell'ordine dei pA su bande fino a 200kHz

2. Nel secondo livello viene effettuata la computazione numerica dei dati attraverso un'architettura di elaborazione, costituita da un *ADC*, che controlla le funzionalità del sistema e organizza il trasferimento di dati al PC.

3. L'ultimo livello si occupa di acquisire, visualizzare e memorizzare i dati, ed è quello in cui si colloca il software trattato da questa tesi.

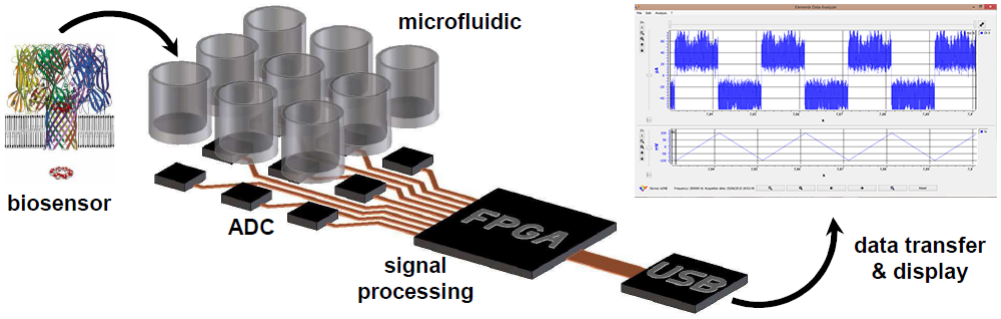

Figura 1.2: La struttura del dispositivo [1]

1.3 Caratteristiche dei dispositivi

Si tratta di dispositivi di dimensioni molto ridotte, ma che permettono un gran numero di analisi ed esperimenti. Per questo forniscono un'alta risoluzione e diverse modalità di lavoro.

Le funzionalità principali offerte sono[2]:

- Frequenza selezionabile, da un minimo di *1.25 kHz* fino ad un massimo di *200 kHz*

- Range selezionabile, scegliendo tra *20 nA* e *200 pA*

- Risoluzione ADC a 14 bit
- Range degli stimoli di tensione di $\pm$ *380 mV*

I protocolli di tensione

Attraverso il software in esecuzione sul PC è possibile inviare al dispositivo diversi protocolli di tensione, opportunamente configurabili. Il gran numero di esperimenti possibili è dovuto proprio alla presenza di questi protocolli, che vengono, caso per caso, scelti dall'elettrofisiologo che sta operando l'analisi.

Connessione al PC

È di fondamentale importanza per il funzionamento del dispositivo la connessione attraverso USB ad un computer, sia per la sua alimentazione che per l'analisi dei dati. L'interfaccia USB, inoltre, favorisce la portabilità del dispositivo poiché plug'n play ed ormai presente in tutti i calcolatori elettronici.

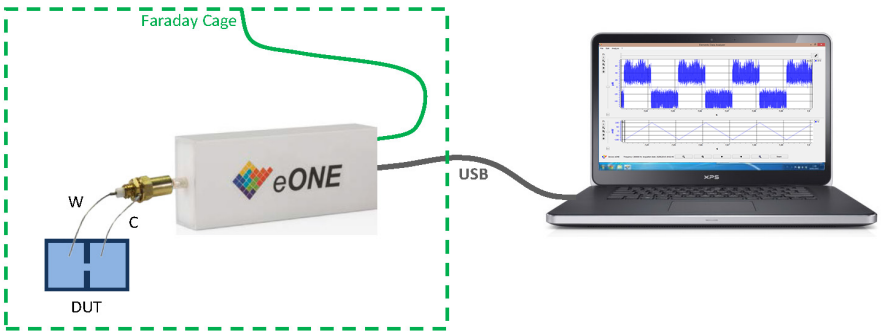

Figura 1.3: La connessione di un *eONE* ad un computer [2]

1.4 Applicazioni in elettrofisiologia

I dispositivi trattati sono utilizzati per lo studio e l'analisi dei canali ionici, settori di analisi riguardanti l'elettrofisiologia, che studia in generale il comportamento delle cellule e alcune loro componenti dal punto di vista elettrico.

I canali ionici

I canali ionici sono pori naturali formati da proteine che si inseriscono spontaneamente nelle membrane cellulari. Regolano il passaggio di sostanze nella membrana cellulare attraverso un trasporto selettivo degli ioni dato dal gradiente elettrochimico tra le due soluzioni separate dalla membrana. Ne esistono diversi tipi, distinti dalla loro selettività di ioni.

I canali ionici possono aprire o chiudere i loro pori a seconda di diversi fattori, lasciando passare ioni. Questo flusso di ioni, se connesso ad appropriati elettrodi, può essere convertito in corrente elettrica, che viene quindi acquisita dai dispositivi sopracitati.

Implicazioni

I canali ionici sono implicati in processi fisiologici, spesso fondamentali, che includono rapidi cambiamenti nelle cellule cardiache e dello scheletro, la contrazione dei muscoli e molti altri. Per il loro fondamentale ruolo biologico sono diventati molto importanti nell'analisi di medicinali, poiché la modulazione farmacologica del canale ionico permette la correzione di molte disfunzioni, soprattutto nel sistema nervoso, cardiovascolare e gastrointestinale.

L'analisi dei canali ionici permette di effettuare test in sicurezza su diverse sostanze in relazione ad altri agenti chimici e patologie, contribuendo molto nello sviluppo di queste e nel test di nuove molecole atte ad intervenire nella cura di specifiche patologie.

Capitolo 2

Tecnologie utilizzate

In questo capitolo verranno descritte le tecnologie e l'ambiente di sviluppo utilizzate nel corso dello sviluppo del progetto.

2.1 USB

L'intera comunicazione tra i dispositivi e il software sviluppato avviene attraverso un canale di comunicazione **USB**. USB, acronimo di *Universal Serial Bus*, è uno standard di comunicazione seriale che permette di collegare diverse periferiche ad un computer.

2.1.1 Descrizione di USB

Storia

USB è stato sviluppato e standardizzato da un gruppo di aziende leader nel settore informatico nel 1995. Tra i fautori di questa tecnologia troviamo *Compaq, DEC, IBM, Intel, Microsoft, NEC, HP* e molte altre, che hanno collaborato attivamente per produrre il protocollo USB e per formare una compagnia no-profit, la *USB Implementers Forum* [1] , per continuare a sostenere lo sviluppo della tecnologia e per fornire supporto al protocollo negli anni a venire.

L'esigenza di questo protocollo si è palesata nei primi anni *'90*, in cui esistevano un gran numero di protocolli e connettori seriali che non erano compatibili tra di loro, spesso non plug'n play e non gestivano contemporaneamente più dispositivi connessi. [3]

[1] Solitamente abbreviato in *USB-IF*

Figura 2.1: Diverse interfacce seriali confrontate con USB [4]

Interfaccia USB

USB è stato progettato per permettere di connettere ad un PC diversi tipi di dispositivi utilizzando un'unica e standardizzata interfaccia. Fornisce un'espandibile, veloce, auto-alimentata, bi-direzionale e hot pluggable Plug and Play [2] interfaccia hardware seriale che rende più semplice la vita agli utilizzatori di PC, permettendo di connettere alla stessa interfaccia tastiere, stampanti, scanner, dispositivi di archiviazione, telefoni e, più generalmente, qualunque dispositivo implementi il protocollo di comunicazione indicato dagli standard.

Evoluzione del protocollo

Grazie agli sforzi di *USB-IF* e al continuo progredire dei dispositivi che utilizzano la tecnologia USB, negli anni il protocollo si è evoluto:

- **Versione 1.0**: Gennaio 1996, supporta collegamenti ad una velocità massima di 1,5 Mbit/s. Molti problemi dovuti alla lunghezza del filo, massima supportata 3 metri.
- **Versione 1.1**: Agosto 1998, risolve molti problemi dell'USB 1.0 e supporta una velocità di 12 Mbit/s.
- **Versione 2.0**: Aprile 2000, raggiunge una velocità teorica di 480 Mbit/s. Inoltre nei mesi successivi al rilascio sono stati introdotti numerosi tipi di connettori (dal lato dispositivo) come *USB Mini-A* e *USB Mini-B*.
- **Versione 3.0**: Primi mesi del 2010, supporta una velocità massima di 4,8 Gbit/s nella modalità *superspeed*.

2.1.2 Funzionamento di USB 2.0

Dato che i dispositivi utilizzati in questo progetto utilizzano il protocollo *USB 2.0* verrà descritto brevemente il funzionamento di questo. [4] La comunicazione USB si basa su tre componenti fondamentali:

[2] *Hot Pluggable Plug and Play*: interfaccia che permette il collegamento e lo scollegamento di un dispositivo anche a sistema avviato. Il sistema operativo si occuperà di identificare automaticamente e caricare il driver appropriato per il dispositivo

- Computer Host
- Device USB
- Bus fisico, rappresentato da un cavo USB, che collega il device all'host computer

Architettura master e slave

Il bus USB è sempre controllato dall'host. Tutti i trasferimenti di dati sono inizializzati e controllati dall'host attraverso uno scheduler. Tutti i dispositivi USB sono schiavi (*slaves*) che rispondono ai comandi dell'host (*master*).

Dato che un solo master controlla il bus, due dispositivi USB (nella versione base del protocollo) non possono comunicare direttamente tra di loro senza comunicare con il master.

Comunicazione

La comunicazione avviene con un protocollo half-duplex, con il segnale differenziale che viaggia su un doppino intrecciato. Prevede tre tipi di trasferimento:

- **Isocrono**: viene garantita una bitrate fissa. Non fornisce il sistema di controllo errori ARQ [3] ed è spesso utilizzata per dispositivi multimediali.
- **Bulk**: prevede l'invio di grandi quantità di dati, con bitrate e tempi di latenza variabile. Questa modalità viene utilizzata per invio di grandi quantità di dati a dispositivi come memorie di massa o stampanti. Viene utilizzato ARQ per garantire l'affidabilità dei dati.
- **Interrupt**: Si basa sull'utilizzo di interrupt, richieste in tempo reale, e viene utilizzato da dispositivi che dovrebbero avere priorità sugli altri, come mouse e tastiera.

[3] ARQ: Automatic Repeat re-Quest è una strategia di controllo errore, che, rilevando un errore in una comunicazione a pacchetti, richiede di ritrasmettere il pacchetto errato

2.2 Librerie Qt

Per le ragioni progettuali che verranno esplicate in seguito nella sezione 3.1.7, sono state di grande importanza nello sviluppo del progetto le librerie di **Qt**, integrate con il linguaggio di programmazione *C++*.

Qt è un framework multipiattaforma per lo sviluppo di applicazioni software che possono essere eseguite su diverse piattaforme software e hardware, con il minimo bisogno di cambiamenti nel codice sorgente.

Strumenti fondamentali

Il framework Qt si basa su 3 concetti fondamentali [5]

- **Completa astrazione della GUI**: Nella sua versione originale Qt usa un proprio motore grafico, non basandosi su componenti grafiche dei vari sistemi operativi ed ottenendo quindi lo stesso aspetto sulle diverse piattaforme.
  Nelle versioni più recenti, però, molti componenti si basano su API native per fornire una resa più performante.

- **Signals e slots**: sono un costrutto del linguaggio introdotto in *Qt* che semplifica di molto la comunicazione tra diversi oggetti, permettendo di implementare con molta facilità il pattern *observer*.
  In generale, permette la comunicazione tra due *QObject* [4]. Questi possono avere delle **signal** e delle **slot**.
  Le *signal* sono segnali che un oggetto può inviare ad un altro oggetto, eventualmente anche passando dei parametri.
  Le *slot* sono particolari metodi di un oggetto che possono essere eseguiti automaticamente in seguito alla recezione di una particolare *signal*.
  Attraverso il metodo **connect()** si può collegare la *signal* di un oggetto alla *slot* di un altro oggetto. In questo modo quando il primo emetterà una signal, il metodo del secondo verrà automaticamente eseguito senza il bisogno di altre classi e metodi intermedi.

- **Compilatore di *metaobject***: si tratta di un compilatore che viene eseguito sul codice di un programma *Qt*. Interpreta delle macro inserite nel codice C++ come annotazioni, generando del codice C++ con meta-informazioni sulle classi utilizzate nel programma.
  Queste informazioni vengono usate in *Qt* per fornire delle funzionalità che non esistono nativamente in C++, come le già citate *signal* e *slot*

[4] *QObject*: classe chiave del modello ad oggetti di *Qt*

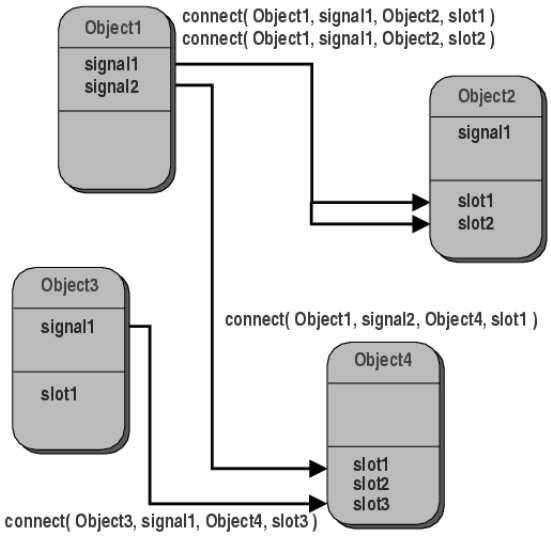

Figura 2.2: Un grafico del sistema di *signal* e *slot* in *Qt* [5]

L'ambiente Qt Creator

Una parte molto importante del *Qt SDK* è **Qt Creator**, un ambiente di sviluppo multipiattaforma in C++ per la creazione di applicazioni che utilizzano *Qt* e in particolare la sua GUI.

Include un debugger visuale ed una interfaccia grafica integrata e supporta diversi compilatori e versioni di *Qt*.

Le librerie *Qwt*

Qwt, o *Qt Widget for Technical Applications*, è un set di *Qt* widget, componenti e classi molto utili per programmi molto tecnici. Fornisce un framework per la visualizzazione 2D di grafici, con scale, slider, e molte altre componenti che permettono di controllare e visualizzare valori, array e intervalli di tipo *double*. Questa libreria è stata largamente utilizzata nella parte di visualizzazione del software sviluppato.

2.3 FTDI

In tutti i dispositivi presentati nella sezione 1.1.2 è presente un chip di comunicazione USB prodotto dall'azienda *FTDI*, *Future Technlogy Devices International*.

L'utilizzo di questo approccio migliora la facilità d'uso, poiché non è necessario implementare il protocollo di comunicazione USB, già implementato nelle librerie FTDI.

Il chip FTDI
Il chip è della serie **FT-X**, costituita da diversi modelli di circuiti integrati che forniscono una interfaccia di comunicazione seriale a basso consumo e basso numero di pin. In particolare il chip utilizzato è il **FT2232D** nell' *eONE* e il **FT2232H** nell' *eFOUR*. Entrambi forniscono una doppia porta seriale o parallela largamente configurabile e gestisce tutta la comunicazione USB internamente al chip. [6]

Il chip *FT2232D* presenta un'interfaccia USB 2.0 che raggiunge una velocità di trasferimento massima di 12 Mbit/s. Il chip *FT2232H* presenta anch'esso una interfaccia USB 2.0, ma ad alta velocità, raggiungendo velocità di trasferimento fino a 480 Mbit/s.

Il driver FTDI
Per comunicare con il dispositivo è necessario installare il corretto driver *FTDI_D2XX* che fornisce numerose funzioni di libreria per facilitare la comunicazione. Queste funzioni sono state utilizzate nel progetto attraverso una libreria C++ fornita da la stessa FTDI.

Le funzioni utilizzate Le principali funzioni utilizzate nella comunicazione con il dispositivo sono:

- **FT_OpenEx** permette, attraverso parametri come il numero seriale e il canale di comunicazione, di aprire una comunicazione su un certo canale con un dispositivo.
- **FT_Close** chiude la comunicazione precedentemente aperta
- **FT_ReadEE** permette di leggere i dati da una *EEPROM* collegata al chip FTDI
- **FT_Read** legge i dati inviati dal dispositivo nel buffer USB

- **FT_Write** scrive dei dati nel buffer USB, che provvede ad inviarli al dispositivo
- **FT_ListDevices** permette di elencare i dispositivi FTDI connessi all'host USB.

EEPROM In tutti i dispositivi descritti in 1.1.2 è presente una memoria di tipo **EEPROM**, connessa al chip FTDI È una memoria elettricamente programmabile e non volatile che permette di effettuare più volte letture e scritture di diverse celle di memoria.

Viene utilizzata per mantenere memorizzate nel dispositivo varie informazioni, tra cui dei parametri di calibrazione molto importanti in fase di lettura dei dati.

2.4 Controllo della versione del codice sorgente

Nello sviluppo del software è stato necessario utilizzare uno strumento di controllo della versione del codice sorgente. Questo perché, trattandosi di un lavoro in team, era fondamentale condividere il codice in modo corretto e tenere continuamente traccia di tutte le modifiche.

Per svolgere questa funzione è stato utilizzato il software **Mercurial**, molto versatile e performante. Tutto il codice sorgente è memorizzato su una repository Mercurial di `www.bitbucket.org`.

Bitbucket
Bitbucket è un servizio di hosting web-based che permette di memorizzare in internet progetti che utilizzano Mercurial o Git.[7] Fornisce la possibilità di creare repository pubbliche, che tutti possono vedere e nelle quali, normalmente, è possibile contribuire al codice, e repository private, come quella utilizzata in questo progetto, dove soltanto chi ha l'accesso può leggere e modificare i file.

Capitolo 3

Analisi dei requisiti

In questo capitolo verranno descritti i requisiti che l'applicazione deve rispettare e le funzionalità che questa deve implementare.

3.1 Descrizione generale

3.1.1 Prospettiva del prodotto

Il software progettato e sviluppato è un'applicazione per l'acquisizione dati ad elevato throughput dai dispositivi prodotti da Elements s.r.l., descritti nel capitolo 1.

L'azienda, che sta vivendo un momento di assoluto sviluppo, ha richiesto questo software per risolvere molti problemi di performance e scalabilità del loro software attualmente in utilizzo.

Nonostante io abbia sviluppato solo alcuni moduli del prodotto verrà effettuata una panoramica completa del sistema, prima di spiegare nel dettaglio i moduli da me progettati.

3.1.2 Funzioni del prodotto

- Lettura dei dati dal dispositivo
- Elaborazione dei dati letti
- Visualizzazione dei dati in tempo reale
- Memorizzazione su disco in diversi formati
- Possibilità di analizzare i dati in tempo reale

3.1.3 Requisiti espressi dall'azienda

Il software progettato deve rispettare dei requisiti fondamentali per consentirne la distribuzione. È di fondamentale importanza la garanzia di prestazioni elevate per evitare assolutamente perdite di dati, in modo da fornire un sistema affidabile ai clienti dell'azienda, utilizzatori finali del prodotto.

Il progetto deve essere molto scalabile, adattandosi anche ai dispositivi che verranno prodotti in futuro dei quali ancora non sono conosciute caratteristiche fisiche come la frequenza di funzionamento o il chip utilizzato per la comunicazione. Inoltre l'applicazione dev'essere costruita in modo modulare, così da permettere in futuro l'aggiunta di ulteriori moduli di elaborazione.

In sintesi, il software deve gestire, senza ritardi o errori, la comunicazione con dispositivi *eONE* ed *eFOUR*, entrambi ad una frequenza di campionamento di 200 kHz per una bitrate calcolata di circa 12 Mbit/s nel primo e circa 30 Mbit/s nel secondo. Deve inoltre essere predisposto a gestire la comunicazione con un dispositivo *eSIXTEEN* ad una frequenza di campionamento di 200 kHz per una bitrate calcolata di circa 103 Mbit/s.

3.1.4 Caratteristiche dell'utente finale

L'utente finale è normalmente l'elettrofisiologo che effettua operazioni di lettura sul dispositivo. Dato che l'utente è già abituato all'utilizzo del precedente software prodotto da Elements, è necessario implementare certe funzionalità in modo analogo alla versione precedente dell'applicazione. Normalmente l'utente utilizza tutte le funzionalità del programma, e lo fa per un tempo lungo e continuato, poiché le operazioni di analisi non sono operazioni brevi.

3.1.5 Ambiente d'utilizzo

Il software dev'essere eseguibile, con buone prestazioni, su computer di fascia media di mercato che eseguono un sistema operativo *Windows* o *Mac*.

3.1.6 Limitazioni

Il software deve essere disegnato per effettuare una lettura in tempo reale dai dispositivi specificati. Deve essere quindi compatibile con questi ed utilizzare i driver a questi correlati. Deve inoltre rispettare dei limiti temporali nella lettura, provvedendo a leggere i dati dal dispositivo con un ritardo minimo e limitando il più possibile la perdita di dati.

3.1.7 Linguaggio e piattaforma di sviluppo

La scelta della piattaforma di sviluppo e del linguaggio di programmazione è stata direttamente vincolata dai requisiti sopraelencati. Questo perché era necessario utilizzare un linguaggio di programmazione che rispondesse ai requisiti di prestazione richiesti e che fosse multi-piattaforma.

Il linguaggio di programmazione scelto per implementare il software è C++, con l'utilizzo di librerie Qt (sezione 2.2). Questa scelta è dovuta anche ad una necessaria continuità con i software già esistenti e sviluppati dall'azienda, in modo da lasciare la futura possibilità di unire in un'unica piattaforma le diverse applicazioni di Elements.

3.2 Interfacciamento con l'esterno

3.2.1 Interfaccia utente

Il software deve fornire un'interfaccia utente completa, intuitiva e funzionale e non deve differire di molto dalla versione precedente del programma, in modo da fornire continuità agli utenti. Una criticità del vecchio software da risolvere è proprio la reattività dell'interfaccia grafica, che, anche leggendo ad alte frequenze dal dispositivo, non deve bloccarsi e deve continuare a mostrare dati. La GUI dev'essere, per quanto possibile, unica, contenendo tutte le funzionalità a portata di mano dell'utente, in modo che questo non debba navigare in un gran numero di finestre.

3.2.2 Interfaccia hardware

Il software deve interfacciarsi con tutti i dispositivi descritti nel capitolo 1 e prevedere un interfacciamento con nuovi possibili dispositivi.
Deve essere in grado di utilizzare il driver FTDI(Sezione 2.3) per la comunicazione con i dispositivi.

3.2.3 Interfaccia software

Il software deve interfacciarsi con il dispositivo attraverso un determinato protocollo di comunicazione, creato e sviluppato da Elements s.r.l. e già definito prima della creazione del software. Deve permettere di memorizzare i dati acquisiti in diversi formati, che siano leggibili dal già sviluppato software di visualizzazione *Elements Data Analyzer* [1] . Inoltre deve utilizzare un efficiente e affidabile sistema di condivisione dati tra i diversi moduli dell'applicazione.

[1]Elements Data Analyzer è un software di visualizzazione e analisi di dati memorizzati da precedenti acquisizioni.

Questo software è stato sviluppato dallo stesso team di sviluppo, me incluso, nel periodo di tirocinio svolto nella stessa azienda

3.3 Funzionalità del sistema

Verranno descritte di seguito nel dettaglio le funzionalità a cui il software deve provvedere, esplicando i requisiti funzionali che queste implicano.

3.3.1 Rilevazione dispositivi

Descrizione Il software deve essere in grado di verificare periodicamente quali sono i dispositivi connessi e mostrarne una lista.

Requisiti funzionali

1. Il sistema deve provvedere automaticamente a verificare i dispositivi connessi.

2. Il sistema deve fornire all'utente una lista di dispositivi connessi

3.3.2 Selezione del dispositivo

Descrizione L'utente deve essere in grado di selezionare un dispositivo tra quelli disponibili.

Requisiti funzionali

1. Il sistema deve fornire all'utente un'interfaccia per selezionare il dispositivo desiderato

3.3.3 Connessione al dispositivo

Descrizione Il software deve connettersi al dispositivo selezionato ed iniziare uno scambio sicuro di dati con questo.

Requisiti funzionali

1. Il sistema deve aprire una comunicazione attraverso un canale affidabile

2. Il sistema deve tenere aperta la comunicazione durante lo scambio dei dati

3.3.4 Lettura dalla EEPROM del dispositivo

Descrizione Il software deve leggere dati dalla EEPROM del dispositivo per verificare la sua validità, se è scaduto il periodo di prova, qual'è la sua versione e per caricare dei parametri di calibrazione del dispositivo utilizzati in fase di lettura.

Requisiti funzionali

1. Il sistema deve essere in grado di leggere i dati da una EEPROM sul canale di comunicazione aperto con il dispositivo
2. Il sistema deve memorizzare i dati caricati dalla EEPROM come caratteristiche del dispositivo

3.3.5 Lettura dei dati dal dispositivo

Descrizione Il software deve leggere i dati dal buffer di output del dispositivo in tempi brevi e gestendo tutto il protocollo di comunicazione, sincronizzandosi e distinguendo i vari dati validi dei diversi canali. I dati letti, che sono in un formato grezzo determinato dalla specifica del protocollo, devono essere trasformati in dati floating point, correttamente riscalati secondo i parametri di calibrazione.

Requisiti funzionali

1. Il sistema deve leggere buffer di dati dal dispositivo
2. Il sistema deve estrarre i dati da questo buffer, riconoscendo la sequenza di sincronizzazione
3. Il sistema deve separare i dati validi da quelli di protocollo e separare i dati dei diversi canali
4. Il sistema deve convertire i dati e riscalarli secondo i parametri di range indicati nella configurazione
5. Il sistema deve moltiplicare i dati per dei parametri di calibrazione caricati dal dispositivo

3.3.6 Elaborazione dei dati

Descrizione I dati letti devono essere messi a disposizione dei diversi moduli di visualizzazione, memorizzazione e analisi.

Requisiti funzionali

1. Il sistema deve provvedere a trasferire i dati in modo sicuro tra i moduli dell'applicazione

2. Il trasferimento di dati dev'essere fatto in modo performante per non essere il collo di bottiglia di tutta l'applicazione

3. La condivisione dei dati in modo concorrente dev'essere gestita per non provocare errori

3.3.7 Invio dei parametri di configurazione al dispositivo

Descrizione Il software dev'essere in grado di inviare sequenze di configurazione, opportunamente scelte, al dispositivo, in modo da cambiare il suo stato di funzionamento, ad esempio cambiando la frequenza, il range d'utilizzo o inviando protocolli di tensione.

Requisiti funzionali

1. Il sistema deve essere in grado di inviare dei byte di configurazione su una connessione aperta

3.3.8 Visualizzazione in tempo reale

Descrizione Il software deve fornire un'interfaccia grafica di visualizzazione dati che mostri a schermo, in modo semplice e intuitivo in un grafico, i dati letti dal dispositivo su tutti i suoi diversi canali.

Requisiti funzionali

1. Il sistema deve leggere i dati dal modulo di acquisizione

2. Il sistema deve mostrare a schermo periodicamente tutti i dati letti

3. I ritardi devono essere minimizzati

4. La perdita di dati deve essere minimizzata

3.3.9 Selezione dei parametri di configurazione

Descrizione Il software deve fornire un'interfaccia grafica per la selezione dei parametri di configurazione come frequenza, range e protocolli di tensione. Questi parametri, una volta selezionati, devono essere elaborati attraverso maschere predefinite ed inviati al dispositivo.

Requisiti funzionali

1. L'interfaccia grafica di selezione dev'essere intuitiva e reattiva
2. L'invio dei dati dev'essere fatto con il minor ritardo possibile

3.3.10 Apertura del file di configurazione

Descrizione Il software deve leggere, decriptare e analizzare un file di configurazione, diverso per ogni modello di dispositivo, per analizzare tutti i parametri di configurazione, la composizione del protocollo di comunicazione e il tipo di maschere da applicare per navigare tra le diverse configurazioni possibili.

Requisiti funzionali

1. Il file dev'essere completamente analizzato caricando tutti i dati in modo corretto

3.3.11 Selezione automatica del file di configurazione

Descrizione Il file di configurazione è differente per ogni versione del dispositivo. È quindi necessario, per semplificare le operazioni dell'utente, caricare in modo automatico il file di configurazione giusto.

Requisiti funzionali

1. Distinguere le diverse versioni del dispositivo
2. Scegliere tra una lista di file di configurazione quello corretto

3.3.12 Memorizzazione dei dati

Descrizione Il software deve fornire la possibilità di salvare i dati per eseguire analisi successive. Questo salvataggio dev'essere possibile nei formati *.dat*, definito da Elements, *.abf*, formato proprietario dell'azienda *Axon*, e *.csv*.

Requisiti funzionali

1. Salvare i dati in modo affidabile, senza variarli o perderli
2. Salvare i dati rispettando i requisiti temporali, senza variare le prestazioni dell'applicazione

3.3.13 Analisi dei dati

Descrizione Dev'essere fornita la possibilità di effettuare, in modo automatico, alcune operazione di analisi dei dati come la creazione dell'istogramma, mostrandone i risultati a schermo.

Requisiti funzionali

1. Effettuare un'analisi affidabile dei dati
2. L'analisi deve essere effettuata senza gravare sulle prestazioni dell'applicazione
3. Dev'essere fornita un'interfaccia grafica che mostri a schermo i risultati delle analisi

Capitolo 4

Analisi

In questo capitolo verrà spiegata brevemente la suddivisone in moduli dell'applicazione nel suo complesso.

Verranno quindi analizzati più nel dettaglio i moduli da me sviluppati. Infine verranno introdotti il file di configurazione, la configurazione del dispositivo e il protocollo di comunicazione.

4.1 Moduli del progetto

Come già espresso nell'analisi dei requisiti (3) l'applicazione è stata sviluppata in modo modulare, per permetterne la massima scalabilità. I moduli individuati sono sei:

- **Rilevamento**
- **Connessione e acquisizione**
- **Elaborazione**
- **Visualizzazione**
- **Memorizzazione**
- **Analisi**

Modulo di rilevamento

Lo scopo di questo modulo è di gestire tutte le funzionalità di rilevamento dei dispositivi. Si occupa quindi di analizzare lo stato dell'USB, di inizializzare i dispositivi e fornisce al modulo di visualizzazione una lista di dispositivi tra i quali scegliere quello desiderato.

Modulo di connessione e acquisizione

Questo modulo si occupa di gestire tutta la connessione e la comunicazione con il dispositivo. Qui sono implementate tutte le funzionalità di lettura dalla EEPROM, lettura di dati dal dispositivo, invio dei byte di configurazione al dispositivo.

Si occupa quindi di tutta la gestione del dispositivo e della sua configurazione, comprendendo il caricamento e l'analisi del config file.

Modulo di elaborazione

È in questo modulo che avviene la distribuzione dei dati presi dal modulo di acquisizione e trasferiti verso i vari moduli di visualizzazione, memorizzazione e analisi.

Modulo di visualizzazione

Questo modulo si occupa della gestione di tutta l'interfaccia grafica, comprendendo quindi la gestione dei comandi da eseguire sugli altri moduli e la visualizzazione dei dati in arrivo dal modulo di elaborazione. Da questo modulo, inoltre, sono abilitate le funzionalità dei moduli di memorizzazione e analisi.

Modulo di memorizzazione

In questo modulo avviene il salvataggio su disco dei dati provenienti dal modulo di acquisizione. Deve gestire diversi formati di file e soprattutto la criticità della lentezza del salvataggio su disco.

Modulo di analisi

Questo è il modulo che permette l'analisi in tempo reale dei dati acquisiti. È collegato al modulo di elaborazione ed effettua vari calcoli, come ad esempio la costruzione in tempo reale di un istogramma.

4.2 Suddivisione dei moduli

Trattandosi di un lavoro in team, è stato deciso di suddividere i moduli tra i due componenti del gruppo, in modo da favorire uno sviluppo più consistente del sistema. I moduli sviluppati da me sono:

- Rilevamento
- Connessione e acquisizione
- Elaborazione

I restanti moduli sono stati sviluppati dall'altro componente del gruppo, Hushi Bajram.

4.3 Panoramica del progetto

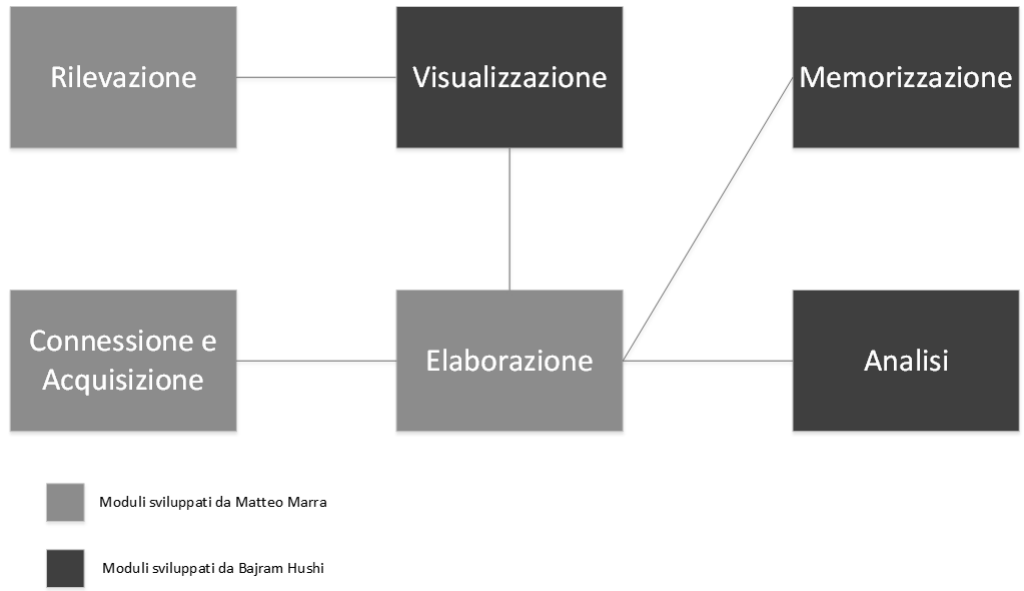

Figura 4.1: Diagramma che mostra i moduli di cui è composto il software sviluppato. I diversi colori indicano i diversi sviluppatori

4.3.1 Comunicazione tra i moduli

Un punto cruciale del software è la comunicazione tra i diversi moduli, soprattutto tra quelli sviluppati da due persone diverse. Lo sviluppo di queste parti, soprattutto nei dettagli delle scelte implementative, è stato effettuato congiuntamente da entrambi i componenti del team, in modo da ottenere una comunicazione ottimale tramite la definizione di specifiche strutture dati.

Nota Da questo punto in poi verranno descritte e relazionate solo le componenti progettate, sviluppate e implementate da me. Verrà descritto prima il modulo di connessione e acquisizione, poiché in questo si trovano delle componenti collegate ad entrambi i moduli di rilevamento e di elaborazione.

4.4 Analisi del modulo di connessione e acquisizione

Il modulo di connessione e acquisizione è probabilmente il modulo più importante del software sviluppato. Il funzionamento dell'intero software, infatti, si basa sulla corretta riuscita delle operazioni che si svolgono qui.

È il modulo che si interfaccia direttamente con il dispositivo, e, attraverso le funzionalità fornite dal driver, provvede a leggere i dati da questo, fornendoli ai moduli di livello superiore. In questo modulo è inoltre presente la gestione della configurazione del dispositivo, con la possibilità di variarne il comportamento.

4.4.1 Casi d'uso

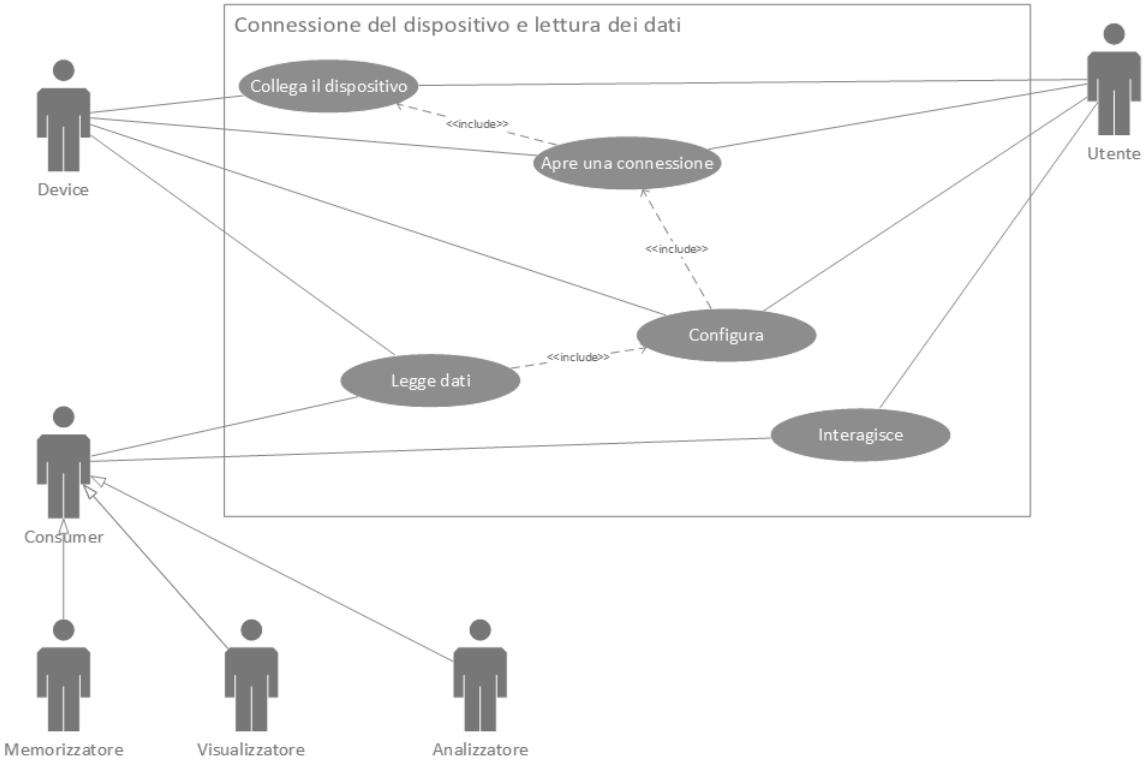

Figura 4.2: Diagramma dei casi d'uso dello scenario di connessione e acquisizione

Come mostrato nella figura 4.2 la fase di connessione e acquisizione presenta uno scenario principale con diversi casi d'uso all'interno di questo. Verrà ora descritto lo scenario nei diversi casi.

4.4.2 Descrizione dello scenario

Caso d'uso: collegamento del dispositivo

Scenario base

1. L'utente collega, tramite un cavo USB, il dispositivo al PC
2. Il software riconosce il dispositivo collegato
3. L'utente seleziona il dispositivo da connettere

Varianti

2.a Il riconoscimento non avviene per errori nel collegamento, resta in attesa fino a quando non rileva un dispositivo.

Caso d'uso: connessione ad un dispositivo

Scenario di base

1. L'utente preme il pulsante per l'apertura della connessione
2. Viene aperto un canale di comunicazione verso il dispositivo selezionato
3. Viene inviata una sequenza iniziale di configurazione

Varianti

2.a Errore nell'apertura del dispositivo. Lo scenario termina

3.a Errore nel caricamento della configurazione. Lo scenario termina

Caso d'uso: configurazione del dispositivo

Scenario di base

1. L'utente seleziona una particolare configurazione attraverso l'interfaccia grafica
2. La sequenza di configurazione viene inviata al dispositivo

Varianti

2.a C'è un errore di trasmissione della configurazione al dispositivo. I dati non sono coerenti

Caso d'uso: lettura dei dati

Scenario di base

1. Il dispositivo produce dati
2. I dati vengono caricati e mandati ai diversi consumer
3. I dati vengono mostrati a schermo dal consumer di visualizzazione
4. I dati vengono memorizzati su disco dal consumer di memorizzazione
5. I dati vengono analizzati e mostrati a schermo dal consumer di analisi

Varianti

4.a Non è stata iniziata la registrazione dei dati, il dispositivo non memorizza. Si può procedere con lo scenario

4.b C'è un errore nella scrittura su disco. La scrittura viene terminata e si può procedere con lo scenario

5.a Non è stata richiesta l'analisi dei dati. Lo scenario può proseguire

4.4.3 Analisi dello scenario

Lo scenario sopra descritto mostra le operazioni di connessione, configurazione e lettura dal dispositivo. Come si può facilmente notare dal diagramma dei casi d'uso tutte queste operazioni sono effettuate in modo sequenziale, dato che se un dispositivo non risulta connesso non ha senso procedere con la lettura e quindi la visualizzazione dei dati da esso prodotti.

Tutti i vari moduli esterni a quello di acquisizione (visualizzazione, memorizzazione, analisi) sono stati rappresentati in una gerarchia come un generico *Consumer*. Questo perché si vuole rendere i moduli di acquisizione ed elaborazione totalmente indipendenti dagli altri. Trattandoli come un generico consumer viene definito un unico modo di scambio dati, che verrà descritto nel capitolo 5. La sequenza descritta viene mostrata nella figura 4.3.

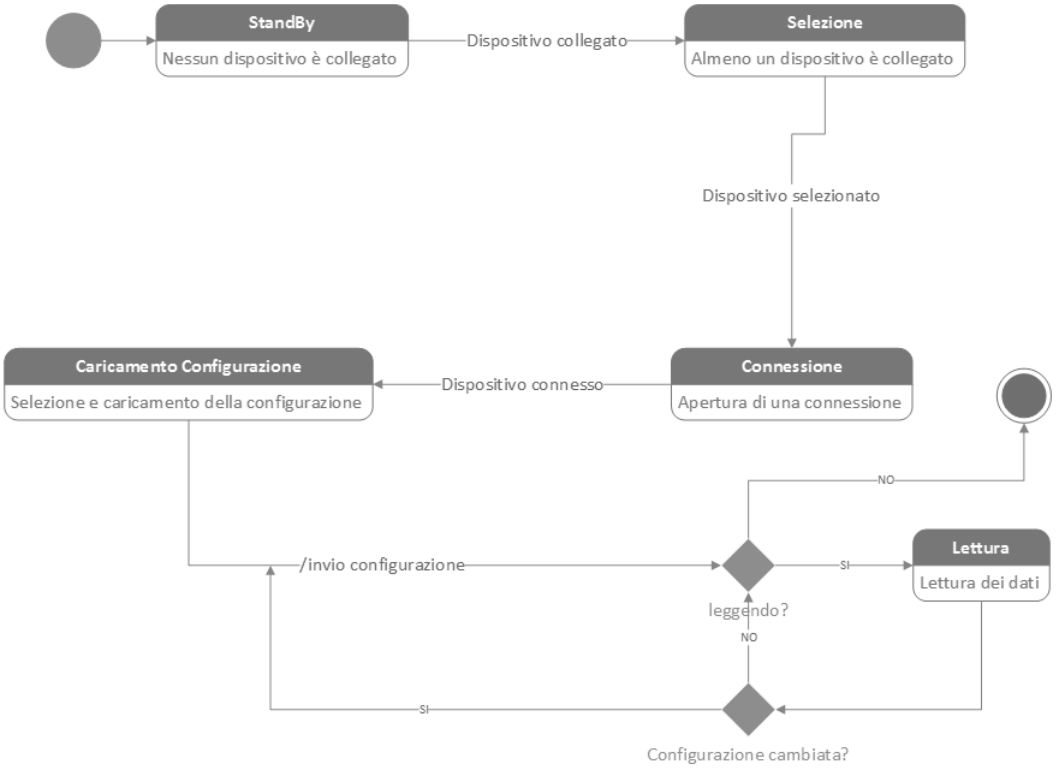

Figura 4.3: Diagramma che mostra la sequenza di stati dell'applicazione in fase di connessione e acquisizione

4.4.4 Diagramma delle classi

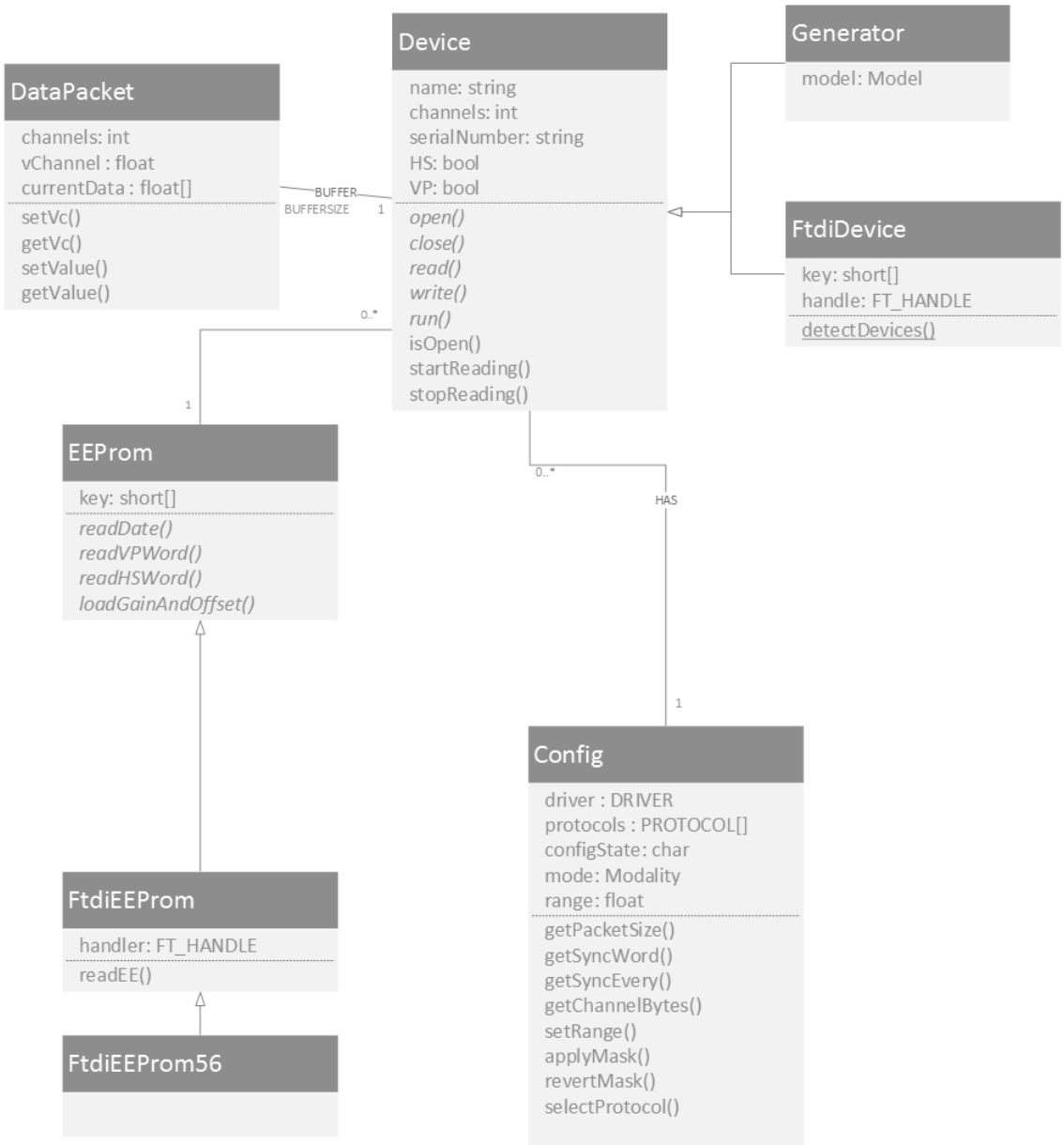

Figura 4.4: Diagramma che mostra le principali classi del modulo di acquisizione

4.5 Analisi del modulo di elaborazione

Anche se di dimensioni limitate, il modulo di elaborazione è il nodo centrale dell'applicazione. È qui che avviene la distribuzione di tutti i dati caricati dal Device ai consumer associati.

4.5.1 Casi d'uso

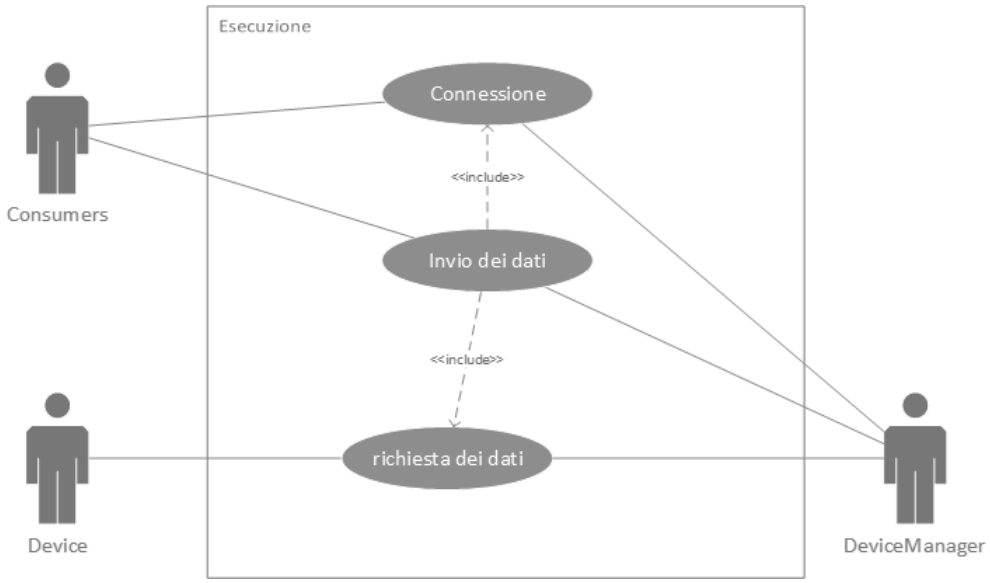

Figura 4.5: Diagramma dei casi d'uso nello scenario di utilizzo del modulo di elaborazione

Come mostrato in figura 4.5, il modulo di elaborazione dei dati provvede a richiedere i dati al dispositivo e li ridistribuisce ai vari consumer.

4.5.2 Descrizione dello scenario

Caso d'uso: richiesta dei dati

Scenario di base

1. Il Device Manager richiede periodicamente un pacchetto di dati al Device
2. Il Device fornisce un pacchetto di dati al DeviceManager

Varianti

2.a Il Device non ha dati da fornire al Manager, il Manager resta in attesa di dati

2.b Il device non è più disponibile, la sequenza termina

Caso d'uso: connessione al Device Manager

Scenario di base

1. Il consumer richiede al Device Manager la connessione
2. Il Device Manager aggiunge il consumer a quelli da notificare in caso di nuovi dati

Caso d'uso: invio dei dati

Scenario di base

1. Il Manager riceve i dati
2. Il Manager invia a tutti i suoi consumer i dati ricevuti

4.5.3 Analisi dello scenario

Anche in questo modulo tutte le operazioni avvengono in modo sequenziale. La distribuzione dei dati non può avere luogo se prima non è stata effettuata una connessione tra il Consumer e il Device Manager. Deve inoltre avvenire prima la lettura dei dati dal dispositivo, che verranno poi inviati ai diversi Consumer connessi.

Questa struttura è quella che permette di tenere contemporaneamente in esecuzione i diversi moduli di visualizzazione, memorizzazione e analisi, inviando a tutti loro gli stessi dati non appena ricevuti. Fornisce inoltre una grande scalabilità, poiché, in caso venga implementato un nuovo modulo che necessita di dati, basterà collegarlo al DeviceManager. Nella figura 4.6 vengono espressi graficamente i diversi stati del modulo di elaborazione.

Figura 4.6: Diagramma degli stati del modulo di elaborazione

4.5.4 Diagramma delle classi

Figura 4.7: Diagramma delle classi principali del modulo di elaborazione

4.6 Analisi del modulo di rilevamento

Il modulo di rilevamento svolge il compito di verificare periodicamente la presenza di nuovi dispositivi e di notificarne l'avvenuta connessione al modulo di visualizzazione, che provvederà poi ad attivare la procedura di connessione già descritta nella sezione 4.4

4.6.1 Casi d'uso

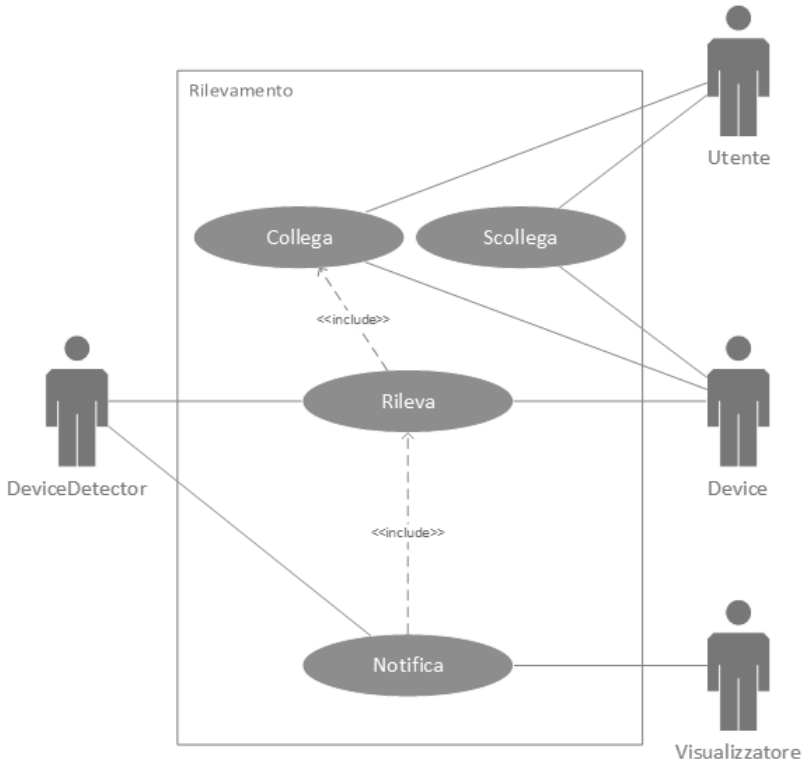

Figura 4.8: Diagramma dei casi d'uso del modulo di rilevamento

La figura 4.8 mostra come il *DeviceDetector*, una volta collegato un Device, lo rileva e notifica al visualizzatore la presenza di un nuovo dispositivo. Analogamente il DeviceDetector deve gestire la rimozione di un dispositivo.

4.6.2 Descrizione dello scenario

Caso d'uso: collegamento

Scenario di base

1. L'utente collega il dispositivo al computer
2. Il dispositivo viene aggiunto all'elenco dei dispositivi USB connessi

Caso d'uso: rilevazione

Scenario di base

1. Vengono rilevati i dispositivi connessi via USB
2. Viene aggiunto, se non presente, il nuovo dispositivo all'elenco dei dispositivi

Varianti

2.a Il dispositivo è già presente, non viene effettuata nessuna operazione

Caso d'uso: notifica

Scenario di base

1. Viene notato un cambiamento nella lista dei dispositivi connessi
2. Viene notificato il visualizzatore del nuovo dispositivo o del dispositivo non più presente

Caso d'uso: scollegamento

Scenario di base

1. Il cavo USB del dispositivo viene scollegato
2. Viene individuato che un dispositivo non è più collegato all'USB
3. Viene rimosso il dispositivo dalla lista di quelli collegati

4.6.3 Analisi dello scenario

Questo modulo è costantemente in ascolto sul bus USB, ricercando i dispositivi connessi. Al momento del collegamento di un dispositivo, il modulo lo rileva e lo aggiunge ad una lista, notificando il modulo di visualizzazione dell'avvenuto collegamento.

Quando un dispositivo viene scollegato, non rilevandolo più tra quelli connessi, lo rimuove dalla sua lista e notifica il visualizzatore dell'avvenuta rimozione. La sequenza delle operazioni è mostrata nella figura 4.9

Figura 4.9: Diagramma degli stati del modulo di rilevamento

4.6.4 Diagramma delle classi

Figura 4.10: Diagramma delle classi del modulo di rilevamento

4.7 Il file di configurazione

Il file di configurazione, chiamato anche *config file*, è un file contenente tutti i parametri necessari a configurare il software e il dispositivo per la lettura dei dati. È un file criptato, differente per ogni versione e sotto-versione dei dispositivi, ereditato dalle precedenti versioni del software e leggermente ottimizzato per adattarlo al software da noi sviluppato.

Verranno ora esposte le principali informazioni contenute nel file, non potendo però scendere nei dettagli sulla sua composizione specifica perché parte integrante della forza strategica dell'azienda. Il *config file* contiene:

- Informazioni sul dispositivo, come ad esempio il numero di canali, che non possono essere memorizzate in questo per dimensione limitata della memoria EEPROM.
- Definizione del protocollo di comunicazione utilizzato
- Sequenza iniziale di configurazione del dispositivo
- Definizione delle diverse maschere di configurazione
- Elenco e struttura dei controlli da inserire nell'interfaccia grafica di configurazione

I dati contenuti nel file, propriamente decriptati, vengono utilizzati nelle fasi di connessione, lettura, costruzione dell'interfaccia grafica e cambiamento della configurazione.

4.8 Configurazione del dispositivo

La configurazione del dispositivo avviene ogni qualvolta viene utilizzato un comando diverso nell'interfaccia grafica per variarne il funzionamento. Consiste in un buffer di bit di configurazione che, propriamente mascherati attraverso i comandi da eseguire, viene inviato attraverso il canale di comunicazione. Anche in questo caso non possono essere forniti dettagli maggiori sulla composizione di questo buffer.

4.9 Protocollo di comunicazione

Il protocollo di comunicazione definisce come avviene la comunicazione tra il dispositivo e il computer. Il dispositivo, infatti, invia continuamente una sequenza di dati ordinati, organizzati in pacchetti definiti nel file di configurazione. Questi pacchetti sono composti da una certa sequenza di sincronizzazione, in modo da poter riconoscere una eventuale perdita della sincronizzazione e quindi incoerenza dei dati, seguita dai dati prodotti dal dispositivo, con il valore di tutti i canali e del canale di tensione.

Il buffer in uscita dal dispositivo contiene quindi una serie di questi pacchetti, che vengono poi elaborati nel modulo di acquisizione. Esiste inoltre un protocollo di comunicazione da PC a dispositivo, attraverso il quale vengono inviati i byte di configurazione. Anche questo è descritto nel config-file, in modo da essere totalmente dinamico rispetto ai dispositivi.

Il software prevede la possibilità di gestire più protocolli di comunicazione nello stesso dispositivo, in modo da poter cambiare alcune proprietà di questo anche a runtime se necessario, ad esempio per mostrare meno canali o per produrre dati in particolari modi.

La possibilità di gestire più protocolli dipende comunque dal dispositivo hardware e da ciò che è indicato nel file di configurazione.

Capitolo 5

Progettazione

In questo capitolo verranno descritte le classi dei vari moduli, indicando le scelte progettuali che hanno portato alla loro definizione.

5.1 L'uso del multithreading

Per riuscire a rispettare il vincolo prestazionale descritto nella sezione 3.1.3, dopo un'attenta analisi del precedente software e delle alternative possibili, è stato deciso di utilizzare una tecnologia multithreading, ormai comune in molte applicazioni. Questo permette di eseguire contemporaneamente [1] i diversi moduli dell'applicazione migliorando significativamente le prestazioni. Inoltre l'approccio multithreading permette di avere una GUI reattiva, che non si blocca eseguendo le varie operazioni di connessione, acquisizione, visualizzazione, memorizzazione, ecc...

Concettualmente *Device*, *DeviceManager* e i vari *ContentConsumer* lavorano su thread diversi, e, oltre a questi, anche *DeviceDetector* si trova in un thread, per provvedere ad un ascolto continuo sul canale USB. Come si può notare, quindi, ogni diverso modulo è eseguito su un thread diverso.

Concorrenza
Nella navigazione dei dati tra i diversi moduli, lavorando questi su thread diversi che possono accedere in momenti diversi alle variabili comuni, si crea un problema di concorrenza e di coerenza dei dati scritti e letti. Nelle parti di comunicazione tra i modulo di acquisizione ed elaborazione, dove vengono

[1] L'esecuzione contemporanea è relativa, dato che, come noto, questa dipende sempre dal sistema operativo e dal suo modo di schedulare i thread nei diversi istanti di tempo in modo completamente trasparente al programmatore e non modificabile.

scambiati i dati tramite un buffer comune attraverso una implementazione del problema *Readers and Writers* [2], lo scambio di dati avviene con l'utilizzo di costrutti di mutua esclusione forniti da Qt, come sarà descritto nel capitolo 6.

5.2 Modulo di connessione e acquisizione

5.2.1 Progetto delle associazioni

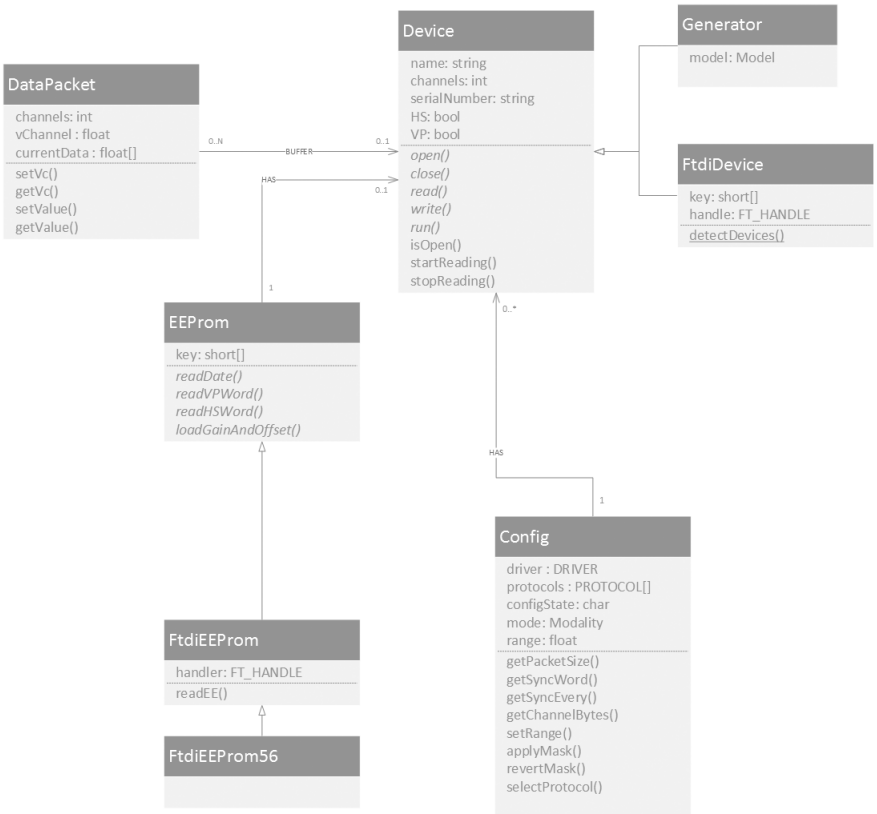

Figura 5.1: Diagramma delle classi e delle associazioni del modulo di connessione e acquisizione

[2]Readers and Writers è un comune problema nella programmazione concorrente. Si crea quando vi è un produttore che scrive su un buffer, che deve poi essere letto da un lettore, creando quindi una situazione di concorrenza sulle diverse variabili del buffer.

5.2.2 Descrizione delle classi principali

Device

Device è la classe chiave di questo modulo. Astrae un dispositivo fornendo alle classi esterne tutte le modalità per comunicare con esso. Presenta vari campi contenenti le diverse informazioni del dispositivo (nome, numero seriale, versione, ...), che vengono caricati dalla EEPROM del dispositivo in fase di apertura e da un'analisi del file di configurazione.

È una classe astratta, che necessita quindi di essere implementata. Le implementazioni di *Device* devono fornire una implementazione dei metodi *run(), open(), close(), read(), write()*. Questo perché, prevedendo l'aggiunta futura di dispositivi che possano utilizzare un chip diverso da quello FTDI, sarà in seguito necessario re-implementare soltanto quei metodi con l'utilizzo dei differenti dati, rendendo questo cambio trasparente al resto dell'applicazione.

Le due implementazioni per ora fornite nel software sono *FtdiDevice* e *Generator*.

FtdiDevice

FtdiDevice rappresenta un dispositivo di tipo FTDI, includendo quindi tutti i dispositivi finora prodotti dall'azienda. Ha come campo, oltre quelli ereditati da Device, una chiave utilizzata nel decriptare varie informazioni, e un handler di tipo *FT_HANDLE*, che è una classe del driver FTDI, rappresentante lo stato della connessione e utilizzato in tutte le fasi di comunicazione con il dispositivo.

Generator

Generator è una implementazione software del dispositivo creata per simulare la generazione dei dati in assenza del dispositivo fisico, utilizzata soprattutto nella prima fase di sviluppo e nella fase di testing. Implementa quindi il metodo run, generando dei dati più o meno casuali, a seconda della sua configurazione, che sono poi trattati allo stesso modo dei dati reali negli altri moduli.

DataPacket

Datapacket rappresenta un pacchetto di dati: sono presenti infatti i campi rappresentanti il numero di canali, il valore della tensione e un array con i diversi valori in corrente. I metodi permettono di settare e recuperare questi valori, e il pacchetto è utilizzato in un *buffer*, contenuto in Device, che verrà poi utilizzato nello scambio di dati con il modulo di elaborazione.

Config

Config rappresenta l'astrazione del file config descritto nella sezione 4.7. Ha quindi come campi tutti i parametri di configurazione ed è associato e presente in ogni Device, in modo da fornire sempre una lettura coerente con la corretta configurazione. Senza la presenza di un config file il device non può essere aperto, ed è per questo che è stato implementato un sistema di selezione e caricamento automatico del config.

EEProm

EEProm rappresenta l'astrazione della memoria EEPROM di un dispositivo. È sempre associata e presente in un dispositivo, altrimenti non sarebbe possibile leggere da questa le informazioni e i parametri di calibrazione. È stato necessario astrarla dal dispositivo per fornire la possibilità di inserire EEPROM diverse in dispositivi, anche di tipo FTDI, differenti. Fornisce quindi i metodi *readDate(), readVPWord(), readHSWord(), loadGainAndOffset()* per recuperare da questa le informazioni fondamentali alla lettura dei dati.

FtdiEEProm e FtdiEEProm56

FtdiEEProm è una estensione di EEProm. Non rappresenta però una EEProm in versione finale utilizzabile. Questo perché tutte le EEPROM che comunicano con un chip FTDI, hanno bisogno di un riferimento all'handler del collegamento, inserito appunto nel generico FtdiEEProm. Fornisce una implementazione del metodo readEE() che, dato un indirizzo di memoria, ritorna il valore presente a questo.

Una implementazione funzionante di FtdiEEProm è FtdiEEProm56, dove il numero è dato dalla versione di EEPROM attualmente presente nei dispositivi *eONE* ed *eFOUR*.
In questa sono definiti tramite costanti tutti gli indirizzi di memoria da utilizzare e sono implementati i metodi di lettura, avvalendosi del metodo *readEE()* precedentemente definito.

5.2.3 Operazioni svolte

Le operazioni del modulo di connessione ed acquisizione possono essere riassunte in un diagramma di sequenza, mostrato nella figura 5.2

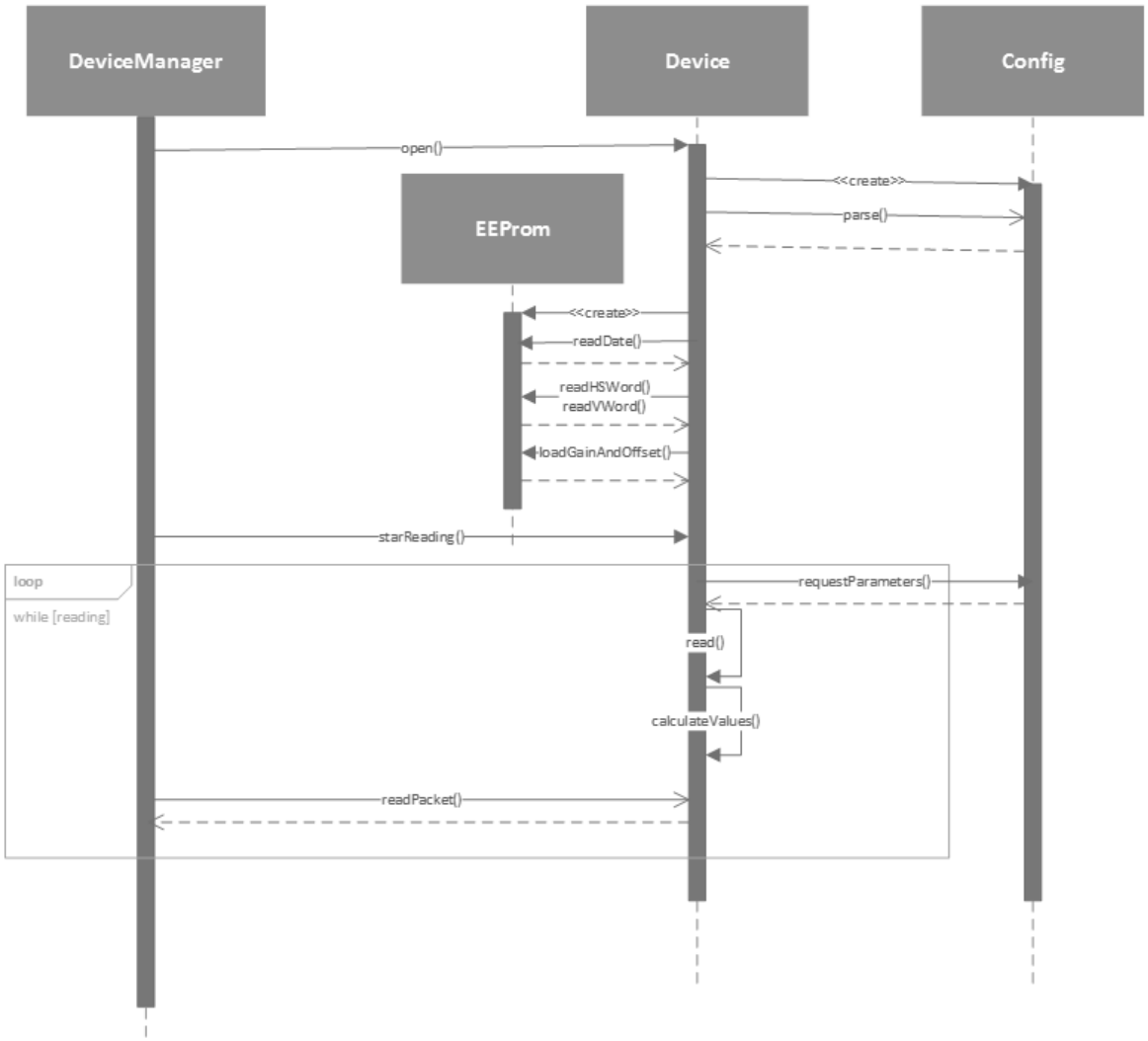

Figura 5.2: Diagramma che mostra la sequenza delle operazioni di acquisizione tra gli oggetti del modulo

5.3 Modulo di elaborazione

5.3.1 Progetto delle associazioni

Figura 5.3: Diagramma che mostra le classi principali del modulo di elaborazione e le loro associazioni

5.3.2 Descrizione delle classi

DeviceManager

DeviceManager è la classe costituente il modulo di elaborazione. Viene eseguita in un thread e si occupa di chiedere periodicamente dei dati al device collegato e di redistribuirli ai diversi ContentConsumer connessi. Ha come campi un elenco dei ContentConsumer connessi e il Device che gestisce. Presenta il metodo run(), dove viene effettuata la richiesta di dati dal dispositivo e l'invio dei dati ai ContentConsumer associati.

ContentConsumer

ContentConsumer è una classe astratta che rappresenta un generale consumatore di dati. Presenta un metodo *recieveData()* che viene chiamato dal DeviceManager non appena sono caricati nuovi dati. Viene implementato dal Visualzzatore (MainAppWindow), dal Memorizzatore (DataWriter) e dal modulo di analisi. Anche questo è mantenuto in un thread separato, in modo da ottimizzare i tempi di visualizzazione a schermo, scrittura sul file e analisi.

5.3.3 Operazioni svolte

Le operazioni di lettura dal Device e distribuzione ai vari consumer possono essere riassunti nel diagramma di sequenza in figura 5.4. Le operazioni si riferiscono ad un generico ContentConsumer, ma sono analoghe nel caso vengano connessi diversi ContentConsumer

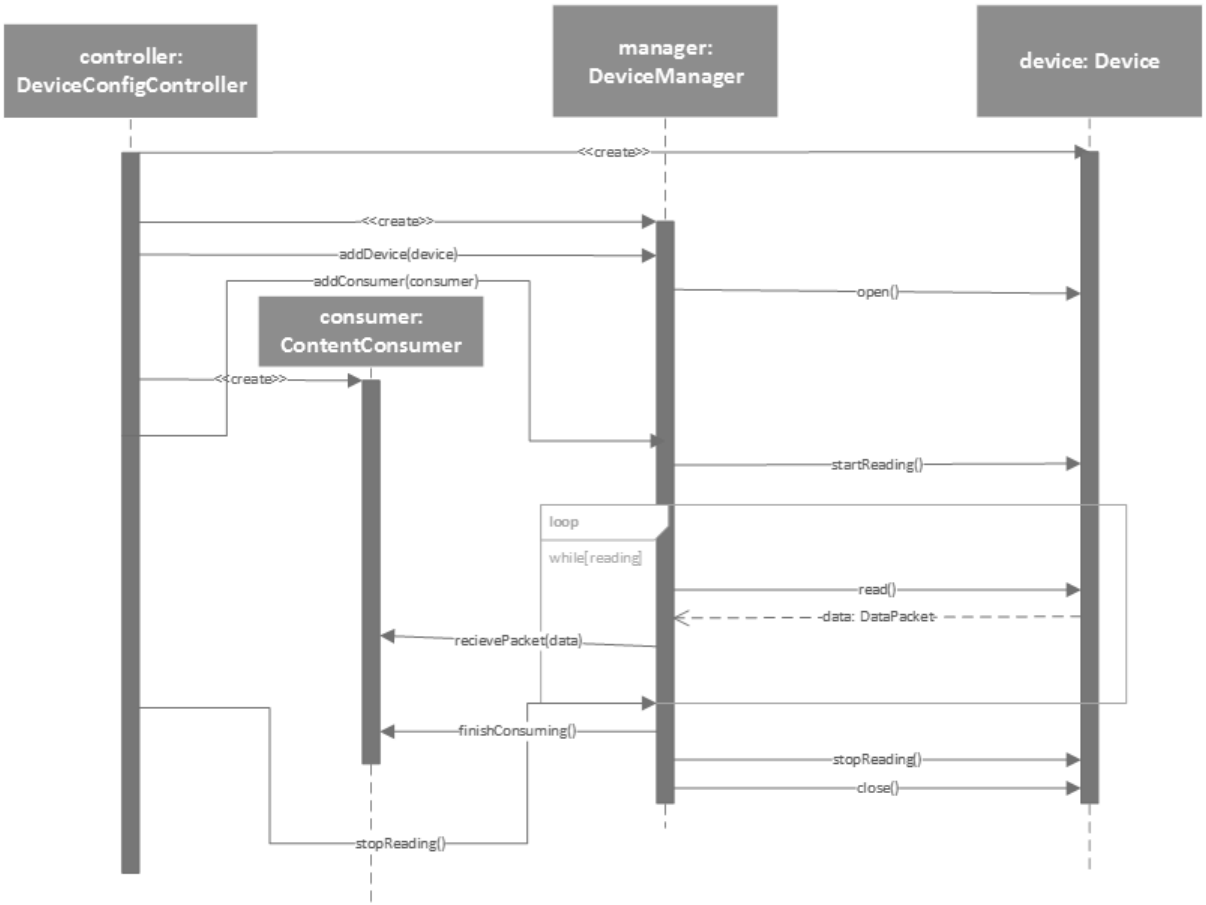

Figura 5.4: Diagramma che mostra la sequenza delle operazioni di elaborazione dati tra gli oggetti del modulo

5.4 Modulo di rilevamento

5.4.1 Progetto delle associazioni

Figura 5.5: Diagramma che mostra le classi e le associazioni del modulo di rilevamento

5.4.2 Descrizione delle classi

DeviceDetector

Device detector è la classe che compone il modulo di rilevazione. Ha come obiettivo quello di stare continuamente in ascolto sul canale USB per notificare al modulo di visualizzazione la presenza di nuovi dispositivi.

Anche questo viene eseguito su un thread separato, implementando tutte le operazioni nel metodo *Run()*. Presenta inoltre un metodo *detectFtdiDevice()* che provvede a verificare la presenza di dispositivi FTDI nel canale USB tramite l'utilizzo del driver.

Nel caso dovessero essere prodotti nuovi dispositivi con caratteristiche e chip diversi, basterà aggiungere qui un metodo e chiamarlo nel metodo *Run()* per effettuare una verifica sul canale USB attraverso lo specifico driver del dispositivo.

La classe è disegnata per implementare il pattern *Observer* [3] in modo da notificare gli altri moduli dell'avvenuta connessione o disconnessione di un dispositivo.

[3]Observer è un design pattern utilizzato per tenere sotto controllo lo stato dei diversi oggetti.

5.4.3 Operazioni svolte

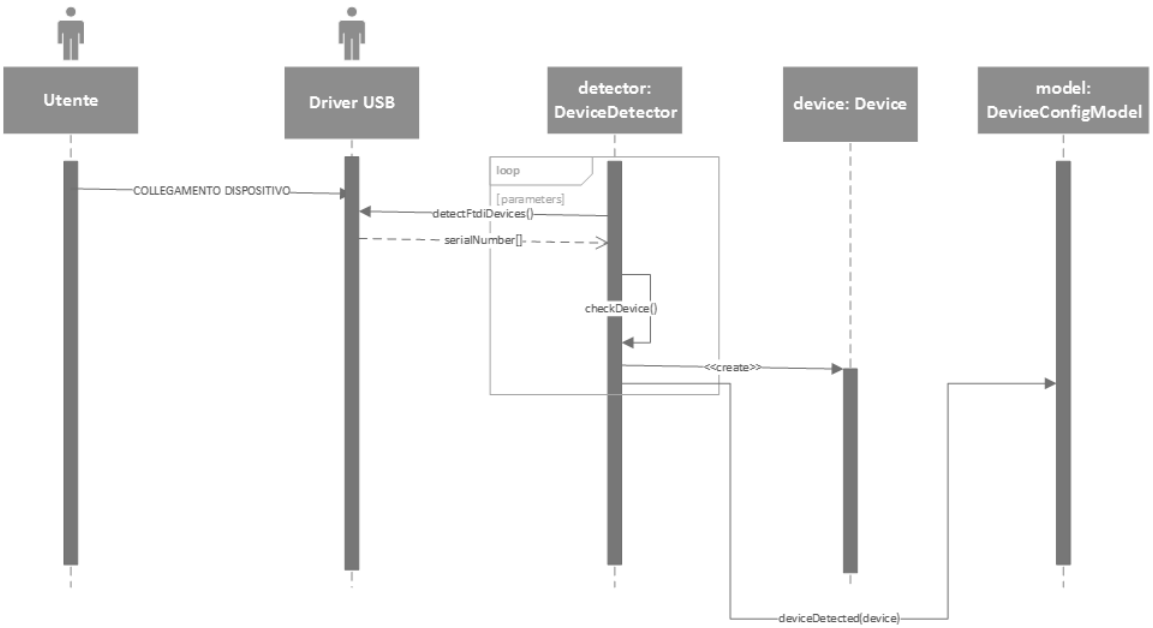

Figura 5.6: Diagramma che la sequenza di operazioni svolte all'interno del modulo

5.5 Componenti di utilità

Esternamente ai moduli sono presenti delle componenti di utilità utilizzate in diverse fasi. Queste sono rappresentate nella classe *Utility*, dove sono forniti dei metodi per decriptare i dati e dove vi sono due metodi per il caricamento automatico del file di configurazione e per la inizializzazione automatica della di EEPROM del dispositivo. Per questo è stato progettato un file che tiene traccia dei diversi file di configurazione e delle memorie EEPROM associate ai vari modelli del dispositivo.

5.5.1 Tabella delle configurazioni

Il suddetto file è stato costruito e codificato in modalità *.csv* [4] , in modo da fornire uno strumento rapido e leggero per il caricamento e per l'editing del file.

Questo file tiene quindi traccia, in quattro colonne di:

1. Modello del dispositivo
2. Versione del dispositivo
3. File di configurazione associato
4. Memoria EEPROM associata

5.5.2 Caricamento automatico del file di configurazione

Questa operazione è stata progettata per facilitare l'operazione di connessione del dispositivo all'utente finale. Questo perché nella versione precedente del software la connessione avveniva con una selezione manuale del file di configurazione da una lista.

Questa operazione è stata automatizzata, facendo in modo che il software riconosca da solo la versione e la sotto-versione del dispositivo per caricare correttamente e in modo automatico un file di configurazione. Attraverso l'uso della tabella delle configurazioni il file viene automaticamente selezionato e successivamente caricato.

[4]Comma Separated Values, è un formato di file basato su file di testo per l'importazione e l'esportazione di una tabella di dati. Ogni riga del file rappresenta una riga della tabella, mentre le colonne sono separate da virgole

5.5.3 Inizializzazione automatica della EEPROM

Dato che ogni dispositivo può avere un modello di EEPROM diversa, con una disposizione dei dati in indirizzi di memoria diversi, anche il tipo di EEPROM viene caricato dalla tabella delle configurazioni sopra descritta.

In questo caso, però, la EEPROM non è, come il config, un semplice file di testo da scorrere e inizializzare. Si tratta bensì di una classe astratta, da inizializzare con la giusta implementazione a seconda del modello di EEPROM letto. Per questo scopo è stata introdotta una classe **MemoryLoader** che utilizza il pattern *Factory Method* [5].

Questa classe si occupa quindi di inizializzare un oggetto della giusta classe , basandosi sul nome della EEPROM caricato dalla tabella di configurazione, e di ritornarlo al richiedente. Il tutto dev'essere implementato in una classe statica, in modo che sia sempre disponibile per la creazione di una EEPROM.

[5]Il factory method viene utilizzato per la creazione di oggetti senza specificarne l'esatta classe.

Capitolo 6

Implementazione

In questo capitolo verranno descritte alcune implementazioni di parti importanti dell'applicazione.

6.1 Uso dei thread

Diverse classi implementate fanno uso di thread, per migliorare le prestazioni e l'aspetto dell'applicazione. Si è scelto di implementare queste classi utilizzando, a seconda delle situazioni, le due classi di libreria Qt: *QThread* e *QRunnable*.

DeviceManager
DeviceManager, parte del modulo di Elaborazione, viene associato sempre e solo ad un unico Device. Dato che viene inizializzato al momento della connessione del dispositivo ed eliminato quando questo viene disconnesso, durante la sua istanza rimane sempre in esecuzione, senza la necessità di dover essere fermato e fatto ripartire. Per questa ragione è stato sufficiente implementare DeviceManager estendendo *QThread*, del quale è stato implementato il metodo *run()*.

DeviceDetector
DeviceDetector, parte del modulo di rilevamento, è anch'esso implementato come *QThread*. In questo caso però vi è, normalmente, una unica istanza di DeviceDetector che è in ascolto sui driver USB e notifica gli altri moduli dell'avvenuto collegamento di un dispositivo. Il thread viene eseguito all'apertura dell'applicazione e terminato alla chiusura.

Device

Device, che rappresenta il dispositivo, è invece implementato in modo differente. Dato che questo viene inizializzato per il suo numero seriale, anche quando un dispositivo viene disconnesso [1] la sua istanza rimane in memoria, pronta ad essere riaperta al momento di una nuova connessione.

Per questo motivo non può essere utilizzato *QThread*, poiché questo accetta una unica esecuzione e non può essere riavviato facilmente al termine di questa. È stato quindi implementato estendendo la classe *QRunnable*, che fornisce sempre il metodo *run()*, ma può essere eseguita diverse volte in varie istanze di *QThread* appositamente creati.

ContentConsumer

ContentConsumer non è stato implementato direttamente come thread. Questo perché non ha un vero e proprio metodo *run()* da eseguire, ed è sempre in attesa che venga effettuata una chiamata al suo metodo *recievePacket()*. Per fare questo l'oggetto deve comunque rimanere in esecuzione e, quando riceve i dati, deve effettuare le sue operazioni senza gravare sul thread del DeviceManager. Per questo motivo è stata introdotta una classe **ConsumerThread**, nella quale viene spostato il consumer da eseguire al momento della necessità, attraverso il metodo *moveToThread()* fornito dalla libreria *Qt*.

[1] Per disconnesso si intende la pressione del pulsante Disconnect, non lo scollegamento del dispositivo

6.2 Operazioni principali

Verranno ora mostrati e spiegati alcuni frammenti di codice delle operazioni principali.

6.2.1 Rilevamento dispositivo

```
void DeviceDetector::detectFtdiDevices(){
    vector<string> serials = FtdiDevice::detectDevice();
    string serial;
    QList<string> currentDevices;
    //check for new devices and emit deviceDetected if there is a new one
    for(unsigned i=0;i<serials.size();i++){
        serial = serials[i];
        serial = serial.substr(0,serial.length()-1);
        if(serial!="" && !currentDevices.contains(serial)){
            currentDevices.push_back(serial);
            if (!devices.contains(serial)){
                devices.push_back(serial);
                Device *d = new FtdiDevice(serial);
                emit deviceDetected(d);
            }
        }
    }

    foreach(string s,devices){
        if(!currentDevices.contains(s)){
            devices.removeOne(s);
            emit deviceRemoved(QString::fromStdString(s));
        }
    }

}
```

Listato 6.1: Estratto del codice di DeviceDetector: metodo detectFtdiDevices()

Il codice mostrato nel frammento 6.1 descrive l'operazione di rilevamento del dispositivo. Dopo aver chiamato il metodo statico *detectDevice()* della classe FtdiDevice, che restituisce un *vector* di stringhe con il numero seriale, viene iterata questa lista, prendendo il numero seriale senza l'ultimo carattere.

Questo perché quando un dispositivo viene connesso, avendo il chip Ftdi due canali, questo viene rilevato due volte come `numeroserialeA` e `numeroserialeB`. Viene verificata la presenza di questo dispositivo nella lista dei dispositivi at-

tuali e, se non presente, viene aggiunta la stringa agli attuali dispositivi, viene inizializzato il dispositivo e viene emessa una *signal* con il puntatore a questo.

Nella seconda iterazione viene invece verificato se nella lista dei dispositivi attuali ne è presente uno che non è più collegato all'USB. Se questo viene rilevato viene emessa una *signal* con il numero seriale del dispositivo, in modo che il modulo superiore si occupi di de-allocarlo.

6.2.2 Apertura del dispositivo

```
[...]
//select channel A or B and update the serial num
int channel = config->getDriver().channel;
char channelWord;
switch(channel){
case 0:
   channelWord = 'A';
   break;
case 1:
   channelWord = 'B';
   break;
default:
   channelWord = 'B';
}

this->serialNumber = this->serialNumber + channelWord;
//calculates the key to be used in the decrypting of eeprom data
calculateKey();

// open the device
result = FT_OpenEx((PVOID)serialNumber.c_str(), FT_OPEN_BY_SERIAL_NUMBER,
    handle);
if(result != FT_OK){
   DeviceOpeningException e(result);
   throw e;
}
this->opened = true;
```

Listato 6.2: Estratto del codice di FtdiDevice: apertura del dispositivo(1)

```
this->setEEProm(
MemoryLoader::loadFromEEPromName(utils::selectEEProm(qSerial),
key,handle));


//Try a command to see if the device is opened
result = FT_GetQueueStatus(*handle, &FTQBytes);
if(result != FT_OK){
   DeviceOpeningException e(result);
   throw e;
}

[Other methods of FTDI driver are called to initialize the device]

checkDate();
checkVersion();
eeprom->loadGainAndOffset(channelDetails,MODE_NUMBER,RANGE_NUMBER);
initializeBuffer();
[...]
```

Listato 6.3: Estratto del codice di FtdiDevice: apertura del dispositivo(2)

Dopo una prima fase di selezione e apertura del corretto file di configurazione, che è stata omessa dal codice mostrato per motivi di spazio, viene selezionato, a seconda di come indicato nel config file, il canale da aprire (tra i canali A e B in uscita dal chip FTDI). Viene quindi aggiornato il numero seriale e chiamato il metodo che calcola la chiave di decodifica dei dati.

Dopodiché viene aperta la connessione al dispositivo attraverso l'utilizzo della funzione *FT_OPEN* del driver FTDI. Viene caricata la EEPROM utilizzando la funzione di utility come factory, e successivamente vengono chiamati i metodi di *checkDate()* per verificare la data di scadenza dell'eventuale periodo di prova del dispositivo, *checkVersion()* per caricare le informazioni sulla sottoversione, *loadGainAndOffset()* per caricare i dati di calibrazione del dispositivo e *initializeBuffer()* per inizializzare il buffer utilizzato nella comunicazione con il DeviceManager.

6.2.3 Lettura dei dati

```
[...]
//Reads if there is something in the queue
result = FT_GetQueueStatus(*handle, &FTQBytes/*FT_Q_Bytes*/);
if (result != FT_OK){
   cout << "error " << result << endl;
   emit deviceNotFound();
   break ;
}
//calculates the packet number, if there are no packets go back to the
    while condition
packNum = FTQBytes / config->getPacketSize();
[...]
bufferSize = packNum * config->getPacketSize();
//Reads the data from the FTDI chip
result = FT_Read(*handle, recieveBuffer, bufferSize, &readResult);
if (result != FT_OK){
   [...]
}
data = recieveBuffer;
int *sync;
//Calculates the sync
scartati = 0;
sync = (int*) recieveBuffer;
while( *sync != config->getSyncWord()){
   data++;
   sync = (int*) data;
   scartati++;
}
//sync if needed
if(scartati>0){
   emit outOfSync();
   result = FT_Read(*handle, recieveBuffer+bufferSize, scartati,
       &readResult);
}
```

Listato 6.4: Estratto del codice di FtdiDevice: lettura dal dispositivo (1)

```
index = 0;
for(int i=0;i<packNum;i++){
   /*
    * skipping the sync in the beginning of every packet
    * */
   index += config->getSyncBytes();
   for(int j=0;j<config->getSyncEvery();j++){
      c=0;
      for(int ch=0;ch<config->getChannelsCount();ch++){
         unsigned char* p = recieveBuffer + index;
         value = 0;
         for(int b=0;b<config->getChannelBytes();b++){
               value <<= 8;
               value += *(p+b);
         }
         index += config->getChannelBytes();
         /*set the values in the packet calling the mapping method
             calculateValue
            with the channel number as parameter.
            If is the Vc channel, use -1 as channel number*/
         if (ch==vChannel.channelIndex){
            packet->setVc(calculateValue(value,-1));
         } else {
            packet->setValue(c,calculateValue(value,c));
            c++;
         }
         //if vchannel is not present
         if(vChannel.channelIndex==-1){
            packet->setVc(0);
         }
      }
      this->writePacket(packet);
   }
}
[...]
```

Listato 6.5: Estratto del codice di FtdiDevice: lettura dal dispositivo (2)

Gli estratti di codice 6.4 6.5 rappresentano una parte del metodo chiave del software, ossia la comunicazione tra FtdiDevice e il driver FTDI per la lettura dei dati, con l'estrazione di questi dai pacchetti ricevuti.

Viene letto lo stato del buffer USB e viene contato il numero di pacchetti presenti. È poi eseguita una lettura di dati per quell'esatto numero di byte, lasciando nel buffer dati di pacchetti non completi eventualmente in eccesso.

Una parte molto importante del codice è la sincronizzazione: se il primo dato corrisponde alla word di sincronizzazione si prosegue nella lettura; nel caso

contrario vengono contati i byte che ci sono prima del sync seguente, e vengono ricaricati i dati mancanti per completare l'ultimo pacchetto, segnalando attraverso una *signal* che c'è stata una perdita di sincronismo.

Dopodichè per ogni pacchetto, attraverso l'uso di puntatori e operazioni di base come degli shift per massimizzare le prestazioni, vengono scorsi tutti i dati e aggiunti al pacchetto da scrivere nel buffer in uscita. A questo punto viene chiamato il metodo *writePacket()* che provvede ad inserire nel buffer il pacchetto.

6.2.4 Invio di una configurazione al dispositivo

```
void FtdiDevice::doWrite(){
    // Clean TX buffer
    FT_STATUS r = FT_Purge(*handle, FT_PURGE_TX);
    unsigned long result = 0;
    int size = config->getConfigBytesNumber();
    unsigned char* buffer = config->getConfigState();
    // Transmit
    r = FT_Write(*handle, buffer, size, &result);
    writing = false;
}
```

Listato 6.6: Estratto del codice di FtdiDevice: invio di una configurazione

L'invio di una configurazione al device viene effettuato recuperando lo stato dal Config e trasmettendolo, attraverso il driver FTDI, al dispositivo.

6.2.5 Distribuzione dei dati

```
void DeviceManager::run(){
    device->open();
    device->startReading();
    Thread *t = new Thread(device);
    t->start();
    while(!stopped){
        DataPacket value = device->read();
        for(unsigned i = 0; i < consumers->size(); i++){
            consumers->at(i)->receiveData(value);
        }
    }
    this->device->stopReading();
    this->device->close();
    delete t;
    this->deleteLater();
}
```

Listato 6.7: Estratto del codice di DeviceManager: lettura e ridistribuzione dei dati

Al momento dell'esecuzione, il DeviceManager apre il dispositivo a lui associato e gli comunica di iniziare a leggere e fa partire la lettura attraverso l'uso di un Thread.

Fino a quando la variabile *stopped* è falsa, viene chiamato il metodo *read()* sul dispositivo e, ricevuto un DataPacket, viene mandato a tutti i consumer associati. All'uscita del ciclo viene detto al dispositivo di smettere di leggere e viene chiuso, oltre a deallocare il thread utilizzato e lo stesso DeviceManager.

6.3 Comunicazione tra i moduli di acquisizione ed elaborazione

Come visto nell'implementazione della distribuzione dei dati, il modulo di elaborazione, rappresentato dalla classe DeviceManager, richiede al modulo di acquisizione, Device, i pacchetti di dati. L'intera comunicazione è quindi gestita dal modulo di acquisizione, che elabora quale pacchetto inviare. Il Device ha un buffer di DataPacket, di dimensioni definite, sul quale mantiene i dati prima di inviarli a chi li richiede.

Questo è un buffer circolare, con due indici: uno della lettura e uno della scrittura. Nel listato 6.8 è mostrato il codice del metodo *read()* in Device.

```
DataPacket Device::read(){
    while(readPointer==writePointer && isEmpty){
        QThread::msleep(1);
    }
    this->mutex.lock();
    isFull = false;
    DataPacket res = buffer->at(readPointer);
    readPointer = (++readPointer)%DATA_BUFFER_SIZE;
    if(readPointer==writePointer && !isEmpty){
        isEmpty = true;
    }
    this->mutex.unlock();
    return res;
}
```

Listato 6.8: Estratto del codice di Device: richiesta di dati

Viene innanzitutto verificato che i due puntatori di lettura e scrittura non coincidano e che il buffer non sia vuoto. Viene poi richiesto il dato alla posizione del puntatore di lettura, che viene incrementato. In seguito viene verificato se il dato letto è l'ultimo del buffer attuale, e eventualmente impostato il flag *isEmpty* a true.

Gestione della mutua esclusione Come si può vedere, nel metodo di lettura è presente una variabile mutex, di tipo QMutex, per la gestione della mutua esclusione della sezione critica di lettura.

Il mutex viene bloccato all'inizio delle operazioni, in modo da assicurarsi che, nello stesso momento, il metodo *run()* non stia scrivendo sul buffer, e viene sbloccato al termine delle operazioni in modo da lasciare libera la scrittura.

Scrittura sul buffer

```
void Device::writePacket(DataPacket *packet){
    mutex.lock();
    buffer->at(writePointer) = *packet;
    writePointer = (++writePointer)%DATA_BUFFER_SIZE;
    mutex.unlock();
}
```

Listato 6.9: Estratto del codice di Device: scrittura sul buffer

Utilizzando lo stesso costrutto di mutua esclusione della lettura, i dati vengono scritti al valore del puntatore di scrittura che viene poi incrementato.

Capitolo 7

Testing

In questo capitolo verranno descritti i vari test effettuati con l'applicazione sui diversi dispositivi di elements. Si procederà descrivendo la valutazione leggendo da un dispositivo *eONE* memorizzando e non, e in modo analogo con il dispositivo *eFOUR*. I test verranno inoltre effettuati variando il carico di lavoro del computer in uso. Non essendo presente un prototipo funzionante di un dispositivo *eSIXTEEN* verrà dimostrato il test attraverso l'uso della classe *Generator*, rappresentante un simulatore software del dispositivo.

Il computer in uso

Il PC su cui è stato effettuato il test è un pc portatile *Asus* prodotto nell'anno 2012, con processore *Intel core i7-2670QM, 2.2 to 3.0 GHz*, con 8 GB di memoria RAM DDR3, su cui è eseguito il sistema operativo *Microsoft Windows 8.1*, connesso ad alimentazione elettrica e con il profilo di risparmio energia impostato a *"bilanciato"*.
Questa configurazione rispetta la richiesta dell'azienda di fornire buone prestazione su PC di fascia media di mercato.

7.1 eONE

Il dispositivo *eONE* effettua la lettura di dati su un solo canale. I dati sono in formato *float*, con una dimensione di quattro byte per dato. Ogni campionatura vengono ricevuti due dati: quello del canale in corrente e quello del canale in tensione. Nella versione *HS*, quella che presenta tutte le funzionalità, raggiunge una frequenza di campionamento massima di 200 kHz. La bitrate da mantenere richiesta è quindi di circa 12,20 Mbit/s.

7.1.1 Condizioni di base

Per condizioni di base si intende l'applicazione *EDR* in esecuzione, con soltanto altri programmi di base come un browser, un editor di testo ed un programma di riproduzione musicale aperti contemporaneamente. Senza l'applicazione in esecuzione l'utilizzo di CPU si attesta intorno all'8%, consumo di RAM intorno a 4 GB.

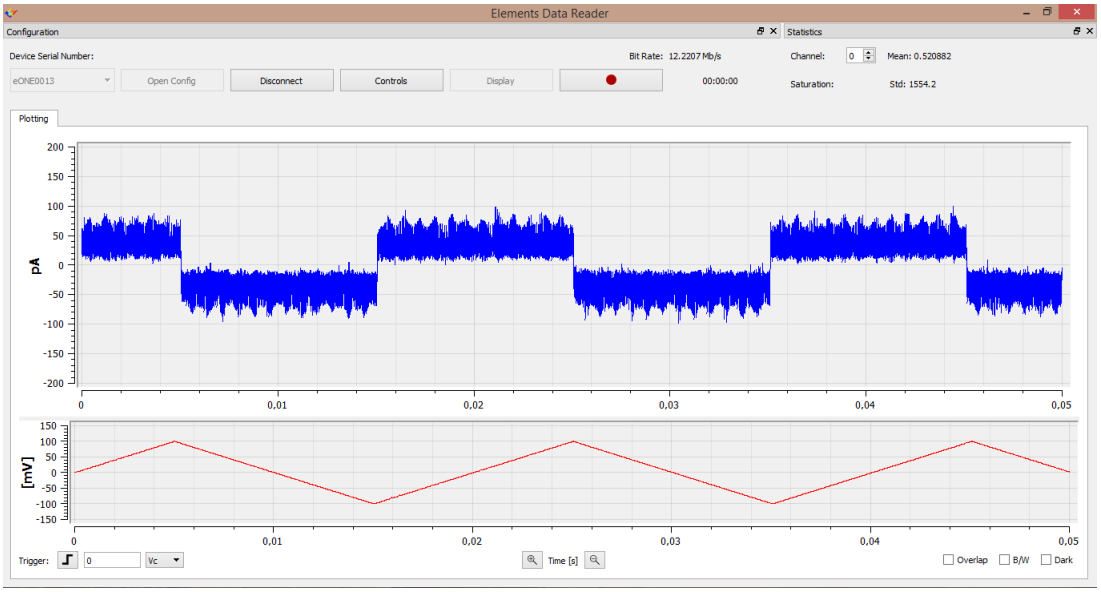

Figura 7.1: Schermata dell'applicazione: lettura da eONE in condizioni di base

Leggendo dati

Nello scenario in cui non si stiano né memorizzando né analizzando dati, mostrato in figura 7.1, la bitrate mantenuta è circa uguale a quella richiesta, ossia 12,207 Mbit/s, rispondendo completamente ai requisiti. L'utilizzo percentuale della CPU dell'applicazione EDR si attesta intorno al 1%, con una occupazione della memoria RAM intorno ai 95 MB.

Leggendo e memorizzando dati

Leggendo e memorizzando dati, la bitrate mantenuta continua ad attestarsi intorno ai richiesti 12.2 Mbit/s. L'utilizzo percentuale di CPU si attesta intorno al 2%, quello di memoria RAM intorno ai 100 MB.

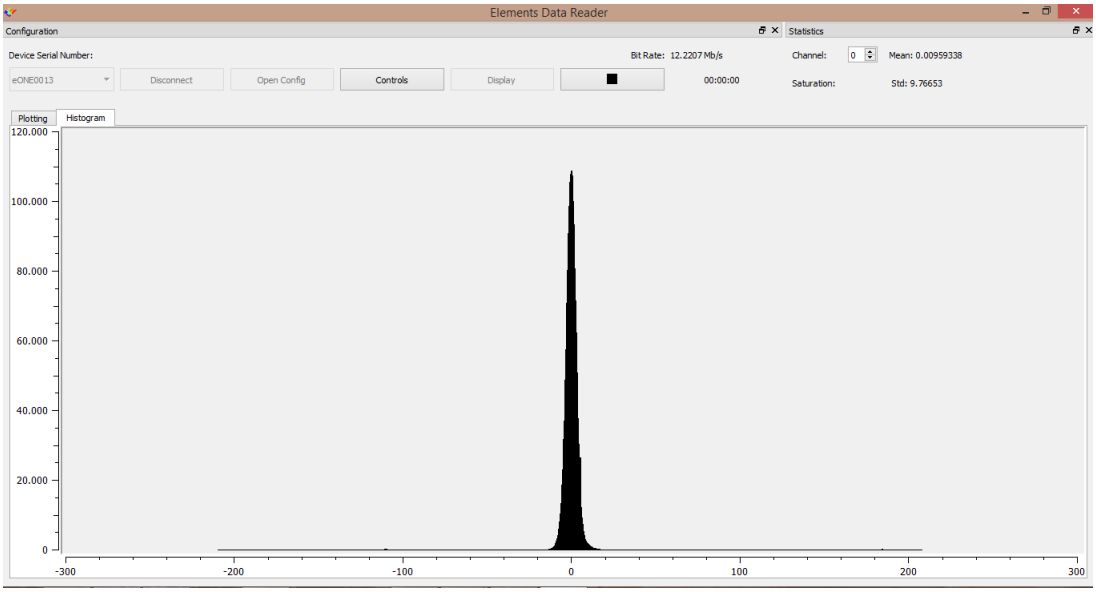

Figura 7.2: Schermata dell'applicazione: lettura, memorizzazione e costruzione istogramma da eONE in condizioni di base

Leggendo, memorizzando e analizzando dati

Leggendo, memorizzando e creando un istogramma in tempo reale dei dati, come mostrato in figura 7.2, l'applicazione risulta ancora stabile intorno ai 12.2 Mbit/s. L'utilizzo della CPU si attesta intorno al 3% ed il consumo di memoria RAM intorno ai 100 MB.

7.1.2 Condizioni di alto carico di lavoro

Per condizione di alto carico di lavoro si intende una esecuzione contemporanea del programma con le applicazioni dello scenario base, più un programma di editing di immagini, un riproduttore video ad alta definizione attivo ed una macchina virtuale accesa.

In questo caso il consumo di base, applicazione sviluppata esclusa, è di circa il 20% per la CPU con circa 6,5 GB di RAM occupati. Le prestazioni si dimostrano, in tutti i casi visti sopra, esattamente comparabili a quelle dello scenario base

7.2 eFOUR

Il dispositivo *eFOUR* effettua la lettura di dati quattro canali in parallelo. I dati sono in formato *float*, con una dimensione di quattro byte per dato. Ogni campionatura vengono ricevuti cinque dati: i quattro dei canali in corrente e quello del canale in tensione.

Nella versione *HS*, quella che presenta tutte le funzionalità, raggiunge una frequenza di campionamento massima di 200 kHz. La bitrate da mantenere richiesta è quindi di circa 30,4 Mbit/s.

7.2.1 Condizioni di base

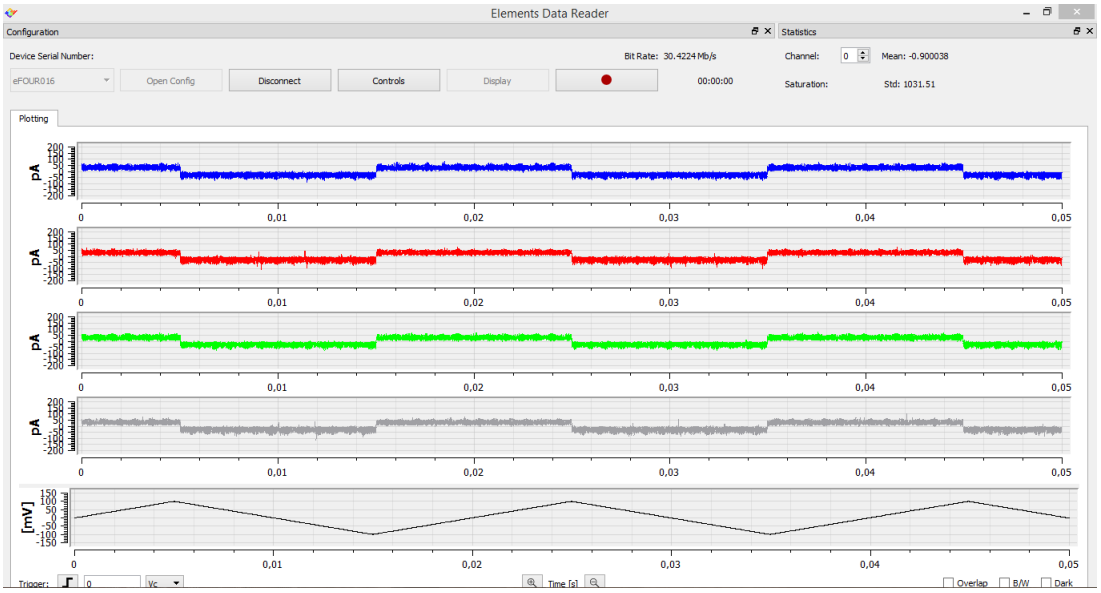

Figura 7.3: Schermata dell'applicazione: lettura da eFOUR

Per condizioni di base si intendono le stesse descritte sopra.

Leggendo dati

Effettuando solo una lettura dei dati, il bitrate si attesta intorno ai 30.4 Mbit/s previsti.
L'utilizzo della CPU è circa 1 %, l'occupazione della memoria RAM è circa 80 MB.

Leggendo e memorizzando dati

Nell'operazione di lettura e memorizzazione dati la bitrate rimane del valore di 30.4 Mbit/s.
L'utilizzo della CPU è di circa il 3%, la memoria RAM occupata è di media 110 MB.

Leggendo, memorizzando e analizzando i dati Leggendo, memorizzando e creando un istogramma in tempo reale dei dati il bitrate risulta ancora stabile intorno ai 30.4 Mbit/s. L'utilizzo della CPU si attesta intorno al 6% ed il consumo di memoria RAM intorno ai 120 MB.

7.2.2 Condizioni di alto carico di lavoro

Nelle condizioni di alto carico di lavoro spiegate nel paragrafo precedente, le prestazioni dell'applicazione non subiscono importanti variazioni, rimanendo costanti nel bitrate.

7.3 Tabella riassuntiva

Device	**Cond.**	**Freq.**	**M***	**A****	**Bitrate**	**CPU**	**RAM**
eONE	Base/Carico	200 kHz	NO	NO	12.20 Mbit/s	1 %	75 MB
eONE	Base/Carico	200 kHz	SI	NO	12.20 Mbit/s	2 %	100 MB
eONE	Base/Carico	200 kHz	SI	SI	12.20 Mbit/s	3 %	100 MB
eFOUR	Base/Carico	200 kHz	NO	NO	30.40 Mbit/s	1 %	80 MB
eFOUR	Base/Carico	200 kHz	SI	NO	30.40 Mbit/s	3 %	110 MB
eFOUR	Base/Carico	200 kHz	SI	SI	30.40 Mbit/s	5 %	120 MB

* = Memorizzando, ** = Analizzando, CPU, Bitrate e RAM si intendono come valori medi

7.4 eSIXTEEN

Non essendo disponibile un dispositivo eSIXTEEN, è stata effettuata una prova con un dispositivo simulato.
Bisogna però considerare che le prestazioni del programma sono influenzate ora anche dal processo di generazione dei dati.

Si prevede che il dispositivo eSIXTEEN, alla massima frequenza di 200 kHz abbia, trasmettendo sedici canali di corrente ed uno di tensione, una bitrate di circa 103 Mbit/s.

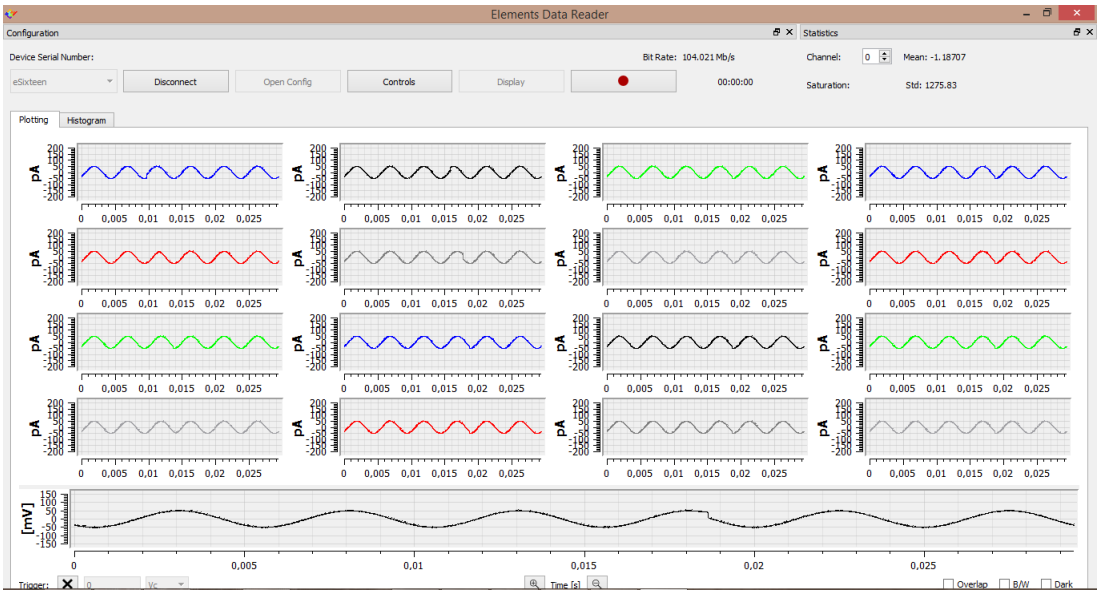

Figura 7.4: Schermata dell'applicazione: lettura, memorizzazione e costruzione istogramma da eSIXTEEN simulato

Come si nota nella figura 7.4, la bitrate mantenuta è intorno ai 103 Mbit/s, con memorizzazione inclusa. L'uso della CPU è intorno al 10% e quello della memoria RAM di circa 100 MB. Si prevede quindi che, leggendo da un eSIXTEEN reale, queste prestazioni vengono mantenute, probabilmente con un utilizzo leggermente minore della CPU e con una maggiore occupazione di memoria RAM.

7.5 Ulteriori simulazioni

Aumentando la frequenza del simulatore, è stato testato che il software riesce a memorizzare dati, sempre lavorando su 16 canali, fino a bitrate maggiori di 300 Mbit/s.

7.6 Stabilità dell'applicazione

Nel corso dei vari test è stata verificata anche la stabilità dell'applicazione, attraverso sessioni prolungate sia di semplice visualizzazione dati che di scrittura su disco dei dati letti. L'applicazione è stata in grado di visualizzare e memorizzare dati per più di 2 ore senza generare alcun errore, rendendo superfluo il proseguire con i test. È stato quindi verificato che l'applicazione si presta a lunghe sessioni di registrazione dati.

7.7 Consistenza dei dati

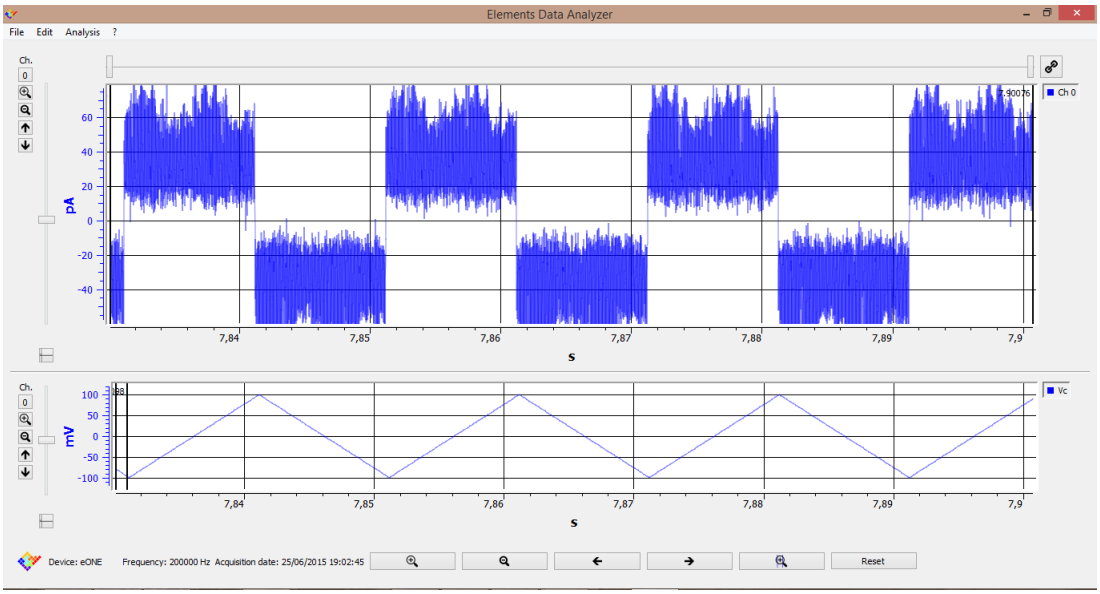

Figura 7.5: Schermata di Elements Data Analyzer con il risultato di una registrazione su eONE

Avendo come modello la precedente versione del software si è tentato di verificare la coerenza dei dati. Purtroppo però, per le limitazioni imposte da USB, non è stato possibile verificare contemporaneamente il comportamento delle due applicazioni leggendo dallo stesso dispositivo.

È stato però notato, da dati registrati da entrambi, che i dati letti dal dispositivo risultano, per valori e forme d'onda, identici a quelli del software precedente, indicando quindi che i dati letti sono assolutamente corretti. Inoltre è stata utilizzata come verifica della consistenza dei dati la lettura attraverso il software *Elements Data Analyzer*. Il risultato, mostrato nella figura 7.5, è assolutamente compatibile con il tipo di registrazione effettuata.

7.8 Soddisfazione dei requisiti

Come dimostrato dai test, i requisiti prestazionali dell'applicazione sono stati rispettati a pieno sia in condizioni di basso carico di lavoro per il computer, sia in condizioni di alto carico, con tutti i dispositivi prodotti alle diverse frequenze, soprattutto non si presenta nessun rallentamento dell'interfaccia grafica leggendo anche a frequenze più grandi.

In sintesi il software è in grado di verificare periodicamente quali sono i dispositivi connessi e mostrarne una lista dalla quale l'utente può selezionare il dispositivo selezionato. È possibile aprire una connessione al dispositivo, la cui apertura provvede a caricare i dati di calibrazione dalla EEPROM.

La lettura dei dati è completamente funzionante, così come l'elaborazione dei dati che provvede a distribuirli ai diversi moduli che si occupano di visualizzare i dati a schermo in tempo reale, memorizzarli e effettuarne l'analisi.

Conclusioni

È stato veramente interessante e motivante sviluppare questo progetto. Il lavoro si è svolto con un'ottima sinergia di gruppo, collaborando sulle parti in comune e trovando una grande sintonia quando si trattava di risolvere i problemi più importanti.

Sono stati di fondamentale importanza l'appoggio e la fiducia totale che il team dell'azienda ci ha concesso, aiutandoci quando ne avevamo bisogno e, soprattutto, valutando attentamente le nostre idee, concedendoci molta libertà in tutte le fasi dello sviluppo del software.

Il processo di realizzazione è stato svolto come avevamo previsto nei primi colloqui, procedendo passo per passo nello sviluppo dei moduli e analizzando ogni volta il buon esito delle nuove implementazioni. Al termine dei lavori il software sviluppato soddisfa la maggior parte dei requisiti richiesti, ma, per motivi temporali, non è stato ancora possibile effettuare i numerosi test di debug, necessari prima di distribuire il software ai clienti.

I due punti di forza dell'applicazione finale sono sicuramente le prestazioni e la scalabilità. Infatti le prestazioni esibite nei test effettuati sono state completamente soddisfacenti, anche di più di quello che ci aspettavamo.

In confronto con la vecchia versione del software, la nostra è molto più responsiva e meno macchinosa in quasi tutte le operazioni, soprattutto leggendo ad alte frequenze. Con la sua struttura modulare risulta inoltre totalmente scalabile, consentendo l'aggiunta di ulteriori moduli modificando minimamente il codice di quelli esistenti e fornendo un'alta manutenibilità del codice.

Sviluppi Futuri

Il software sviluppato è soltanto una prima versione di base, che comunque riesce a soddisfare tutte le operazioni fondamentali. È stato creato con l'intenzione di aggiungere in futuro numerose funzionalità, una di queste è sicuramente estendere il modulo di visualizzazione, aggiungendo funzioni che rendono l'utilizzo del software più facile all'utente finale.

Vi è poi la necessità di sviluppare tutta la parte di connessione a dispositivi di tipo non FTDI che verranno creati da Elements in futuro, cosa per la quale comunque il software è totalmente predisposto. Ulteriormente è prevista l'aggiunta di altri tipi di analisi in tempo reale, che renderebbero il software all'avanguardia nel settore.

In compatibilità con lo sviluppo dell'hardware, inoltre, è prevista l'aggiunta, in sistemi con numerosi canali, la possibilità di deselezionare alcuni canali nel protocollo di comunicazione e nella visualizzazione per migliorare le prestazioni ed evitare la lettura di dati che potrebbero essere ridondanti o inutili.

È previsto, allo stato attuale, un periodo di collaborazione con l'azienda per rendere il software commercializzabile ed estenderne le funzionalità come elencato.

Ringraziamenti

Ringrazio innanzitutto i miei genitori, senza i quali non sarebbe stato possibile arrivare fino a questo punto.
Un grazie particolare va a Bajram per aver fatto con me questo progetto, lavorando senza sosta, giorno dopo giorno, sempre con il massimo impegno per provare a raggiungere il massimo.
Ci tengo a ringraziare Federico, Marco e Michele per la fantastica opportunità che mi hanno offerto e per tutto l'aiuto.
Ringrazio Francesca per i preziosi consigli e Andrea, Alex e Sophie per avermi sopportato durante gli ultimi, intensi, giorni di lavoro su questa tesi.

Elenco delle figure

Bibliografia

[1] Thei Federico, *Phd thesis : A HYBRID TECHNOLOGY FOR PARALLEL RECORDING OF SINGLE ION CHANNELS*, Università di Bologna, 2011.

[2] `http://www.elements-ic.com/`

[3] `http://www.usb-if.com/`

[4] `http://www.usblyzer.com/`

[5] `http://www.qt.io/`

[6] `http://www.ftdichip.com/`

[7] `https://bitbucket.org`