Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base
Area Didattica Scienze MM.FF.NN.

Corso di Laurea in Informatica

Tesi sperimentale di Laurea Magistrale

Progettazione, sviluppo ed ottimizzazione del software online per il sistema di trigger di livello 1 per gli RPC dell'esperimento ATLAS

Relatori
Prof. Guido Russo
Dr. Vincenzo Izzo
Dr. Silvio Pardi

Candidato
Vincenzo Bruscino
matr. N97/171

Correlatore
Prof. Giuliano Laccetti

Anno Accademico 2014-2015

Ai miei genitori, per la loro bontà e dolcezza, per la loro forza e disciplina

Ad Antonio, la mia più grande fonte di ispirazione

Ai veri amici, per lo starmi sempre accanto

A Federica, semplicemente il colore nella mia vita

"Color my life with the chaos of trouble!"

Indice generale

Introduzione....9

1 - Il rivelatore ATLAS....15

1.1 L'acceleratore LHC....16

1.2 I quattro rivelatori....18

1.3 ATLAS....19

1.3.1 Lo spettrometro per i muoni....20

1.3.2 Le camere di trigger RPC....22

2 - Il trigger di primo livello....24

2.1 Il sistema Pad....26

2.2 Struttura hardware della Pad....28

2.2.1 Il blocco timing....30

2.2.2 Il blocco di read-out....33

2.2.3 Il blocco di controllo ed inizializzazione....37

2.3 ELMB 128....37

3 - L'ambiente di test e monitoring....40

3.1 Hardware....41

3.1.1 Accenni al bus seriale CAN....42

3.2 Il software di inizializzazione e controllo: testpad....44

3.3 Problemi riscontrati e possibili soluzioni....49

4 – Implementazione e test....54

4.1 Connessione alla scheda Pad....56

4.2 Gestione della scheda Pad....62

4.3 Test e risultati ottenuti....69

4.4 Casi d'uso ed esempi di test....77
Conclusioni....82
A - Il protocollo CAN....84
A.1 Il Physical Layer....84
A.2 Data Link Layer....85
A.3 Application Layer....89
A.3.1 CAN Application Layer (CAL)....90
A.4 CANopen....91
A.4.1 Object Dictionary....92
A.4.2 Tipi di messaggi....92
A.4.3 Predefined Connection Set....94
B – Il protocollo I2C....96
B.1 Specifiche del protocollo....96
C - Installazione del framework TDAQ su sistemi Linux....101
C.1 Installazione dell'ambiente di programmazione TDAQ....101
C.2 Creazione di un'area di lavoro....102
C.3 Installazione e configurazione del driver Systec CAN....103
C.4 Check-out e compilazione pacchetti....104
Bibliografia....107

Indice delle figure

Figura I.1: LHC ed i suoi esperimenti....9
Figura I.2: Interfaccia grafica del software testpad....13
Figura 1.1: L'acceleratore di particelle LHC....16
Figura 1.2: Collisioni in ATLAS....17
Figura 1.3: Struttura dell'anello di LHC e disposizione dei quattro esperimenti....18
Figura 1.4: Sottosistemi del rivelatore ATLAS....20
Figura 1.5: Sezione dello spettrometro a muoni del barrel. In rosso gli strati RPC. 22
Figura 1.6: Struttura di una camera RPC con indicati tutti i componenti....22
Figura 2.1: Schema dell'intero sistema di trigger e di acquisizione dati di ATLAS. 25
Figura 2.2: Elettronica on-detector ed off-detector del trigger....27
Figura 2.3: Sistema Pad con schede Add-on-board evidenziate....28
Figura 2.4: Architettura del chip PRODE....32
Figura 2.5: Schema a blocchi del funzionamento del Phase Detector....32
Figura 2.6: Schema a blocchi del funzionamento della scheda OR....33
Figura 2.7: Struttura FPGA....36
Figura 2.8: La ELMB 128....37
Figura 2.9: Diagramma a blocchi della ELMB 128....38
Figura 2.10: Disposizione e configurazione del DIP-switch....39
Figura 3.1: Schema di collegamento tra le varie componenti hardware....41
Figura 3.2: Struttura di un frame CANopen....44
Figura 3.3: Diagramma UML del sistema di configurazione e controllo di testpad. 45
Figura 3.4: Menù principale di testpad....48

Figura 3.5: Schermata di connessione della versione in Java di testpad.....51

Figura 3.6: Schermata principale della versione in Java di testpad.....53

Figura 3.7: Esempio di report inerente ad uno Splitter spento o non funzionante....53

Figura 4.1: Classi UML della versione Java di testpad.....55

Figura 4.2: Estratto del metodo formComponentShown.....57

Figura 4.3: Estratto del metodo jButton1ActionPerformed inerente alla gestione del thread.....58

Figura 4.4: Cattura output, rilevazione Pad ed errore CAN.....59

Figura 4.5: Condizione di uscita, stampa dei risultati, sincronizzazione ed avvio del thread.....60

Figura 4.6: Memorizzazione ultima configurazione ed apertura del menù.....61

Figura 4.7: Metodo di selezione del menù PRODE.....62

Figura 4.8: Sorgente inerente al riempimento del menù a tendina delle schede Pad63

Figura 4.9: Metodo inerente al pulsante "Default init".....64

Figura 4.10: Form "Step by step".....64

Figura 4.11: Form per settare il ritardo dei quattro canali del chip PRODE.....65

Figura 4.12: Evento scaturito dal click del pulsante "Apply" del form "Set PRODE" 65

Figura 4.13: Form di lettura/scrittura dei registri della CM.....66

Figura 4.14: Form di lettura/scrittura dei registri della FPGA.....67

Figura 4.15: Voltaggio e temperatura dello Splitter collegato alla Pad.....67

Figura 4.16: Estratto del sorgente inerente alla visualizzazione dei report.....68

Figura 4.17: Schermata inerente alla fase di test.....71

Figura 4.18: Estratto del sorgente relativo al metodo del pulsante "Start".....72

Figura 4.19: Seconda parte del sorgente relativo al metodo del pulsante "Start".....74

Figura 4.20: Metodo inerente alla selezione di una Pad per effettuarne il test.....75

Figura 4.21: Tempi di esecuzione dell'inizializzazione standard e della nuova modalità di test.....77

Figura 4.22: Use case dell'applicativo in Java.....78

Figura 4.23: Sequence Diagram della funzionalità "Esegui Test".....79

Figura 4.24: Report degli errori sulla Pad di lowPT (ID 2).....80

Figura 4.25: Report della Pad di lowPT (ID 2) dopo l'intervento del tecnico.....80

Figura 4.26: Errore rilevato per il mancato collegamento dello Splitter.....81

Figura 4.27: Report inerente allo Splitter, alla FPGA ed al GLinkTX della Pad di highPT con ID 20.....81

Figura A.1: Struttura del Data Frame.....87

Figura B.1: Trasmissione tramite protocollo I2C con indirizzo lungo 7 bit.....97

Figura B.2: Sincronizzazione dei clock.....98

Figura B.3: Arbitraggio tra due elementi.....100

Indice delle tabelle

Tabella 2.1: Principali differenze tra la Pad low e la Pad high.....29

Tabella 4.1: Tempi di esecuzione dell'inizializzazione di default e della nuova modalità di test in relazione al numero di Pad testate.....76

Tabella A.1: Identificatori associati ai servizi offerti dal CAL.....91

Tabella A.2: Identificatori per i messaggi del CANopen.....95

Introduzione

Il CERN (*Conseil Européen pour la Recherche Nucléaire*) di Ginevra [1] è il più grande laboratorio al mondo di fisica delle particelle. Il suo scopo principale è quello di fornire ai ricercatori gli strumenti necessari per la ricerca nella fisica delle alte energie allo scopo di trovare le risposte ai quesiti ancora aperti della scienza. Questi strumenti sono principalmente gli *acceleratori di particelle*, in grado di portare nuclei subatomici ad energie molto elevate, ed i *rivelatori*, che permettono di osservare i prodotti delle collisioni tra fasci di queste particelle.

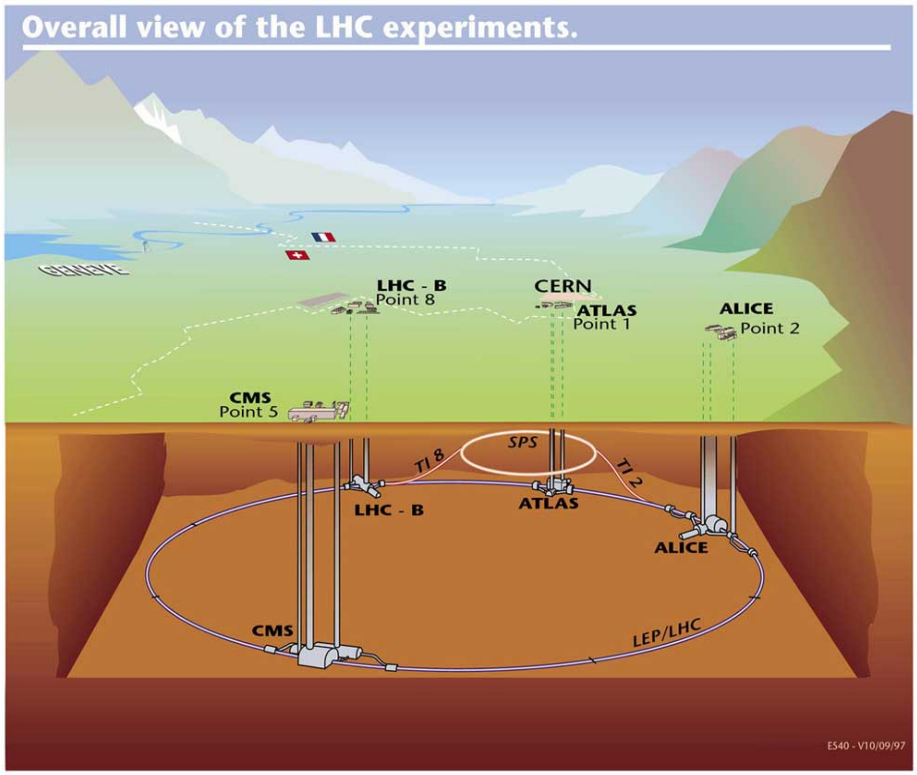

Figura I.1: LHC ed i suoi esperimenti

Il 10 settembre 2008 il CERN ha inaugurato il *Large Hadron Collider* (LHC, Fig. I.1), il più potente e grande (27 km di lunghezza ad una profondità di 100 metri) acceleratore del mondo. A causa della complessità dell'intero sistema, sono previste

costantemente sessioni di manutenzione ed aggiornamento mirate al potenziamento ed alla revisione dei dispositivi presenti in LHC ed alle loro strategie di controllo e monitoraggio. Proprio questi ultimi due aspetti creano il contesto nel quale si cala il lavoro di questa tesi.

L'obiettivo della tesi è stato quello di sviluppare un ambiente di test, nel sito BB5 presso il CERN, per l'elettronica di trigger di livello 1 dell'esperimento ATLAS [2], uno dei quattro che sono in presa dati ad LHC. Il lavoro svolto può essere riassunto dai seguenti punti:

- sono state studiate tutte le componenti dell'elettronica da testare;
- successivamente, è stato analizzato il codice esistente di interfacciamento dell'hardware in C++;
- di questo, ne è stato fatto il debug e ne sono state estese le funzionalità;
- è stato implementato un wrapper grafico, in codice Java [3], per semplificare le operazioni e l'interazione con l'elettronica da testare.

L'attività è di notevole impatto sulle operazioni del livello 1 in quanto, il rinnovato sito di test di BB5, consentirà il collaudo dell'elettronica da installare in esperimento in caso di necessità e la verifica ed il debug di quella estratta dal rivelatore perché malfunzionante.
Dopo una prima fase, di verifica dell'hardware a disposizione e del cablaggio dell'elettronica, è stato necessario installare l'ambiente software che replica quello dell'esperimento ed è stato progettato ed ottimizzato, attraverso operazioni di accesso al driver scritto in codice C/C++, il nuovo software di test per l'elettronica. Uno dei risultati più eclatanti è che questo ambiente di test, tramite accessi semplificati, ha permesso di velocizzare l'interazione con tutta l'elettronica dell'esperimento da controllare e validare.
Le principali problematiche affrontate durante la fase di progettazione sono state:

1. fornire tutte le funzionalità di un ambiente di test dedicato ad un componente hardware;
2. fornire un ambiente di test di facile utilizzo, anche a chi non pratico del sistema Linux;
3. garantire la manutenibilità del sistema;
4. garantire la ripetibilità del test;
5. garantire la sicurezza del sistema;
6. rendere il software compatibile con l'attuale versione del framework TDAQ (05-05-00) [4];
7. rendere disponibile un output del test semplificato per una più facile comprensione;
8. ottimizzare la procedura di inizializzazione dell'elettronica sotto test semplificandone le comunicazioni con i vari moduli da testare.

Di seguito verranno illustrate brevemente le varie fasi del lavoro svolto nell'ambito di questa tesi.

Dapprima c'è stata una fase di comprensione del problema e di studio preliminare presso l'Università Federico II e la sezione di Napoli dell'INFN [5]. Si è studiato il principio base di funzionamento del rivelatore ATLAS, porgendo maggiore attenzione all'elettronica di trigger di primo livello. In particolare, le schede elettroniche denominate Pad, con le relative schede *Add-on-board* (*Coincidence Matrix*, *GLinkTX, TTC*, ecc.), sono state al centro di questa prima fase di studio.

Successivamente c'è stato un periodo di lavoro al CERN (di circa tre mesi) dove sono state svolte le seguenti attività.

- *Approntamento dell'ambiente di test*: assemblaggio e cablaggio di tutto l'hardware da testare, con successiva installazione dell'ambiente software necessaria all'esecuzione del framework TDAQ. Sul server *Dell XPS 8700* è stato montato il sistema operativo *Scientific Linux CERN release 6.7* [6], una

particolare distribuzione (di derivazione *Red Hat*) che contiene tutto il necessario per poter lanciare i software utilizzati al CERN.

- *Comunicazione con i device*: il software comunica con i dispositivi fisici attraverso un'architettura centralizzata, utilizzando un'interfaccia *Systec USB-CANmodul16* [7] con il relativo bus. Tale architettura viene utilizzata sia al CERN, sia, in generale, nel campo dell'automotica. Nell'ambito del lavoro di tesi, ho dovuto studiare tale sistema di comunicazione, integrandone i driver all'interno dei moduli del kernel nel sistema operativo (anche mediante la scrittura di script in Python).
- *Ottimizzazione e rifinitura del driver/software testpad*: per permettere l'interazione tra la stazione di test e l'hardware da controllare, c'è bisogno di uno strato software che vada ad implementare il protocollo di comunicazione. Nell'esperimento ATLAS tale compito viene svolto dal software *testpad*; durante il periodo di tesi, si è ritenuto necessario apportare alcune modifiche al driver dato che sono stati trovati dei bug inerenti alla parte di inizializzazione dei registri interni di alcune componenti della scheda Pad.
- *Implementazione di un'interfaccia grafica per il software testpad*: per rendere più veloce ed intuitivo l'utilizzo del driver da parte degli addetti ai lavori, si è pensato di implementare un'interfaccia grafica (Fig. I.2) per le operazioni principali da svolgere nell'ambito del test. Inizializzazione dei componenti, visualizzazioni dei report, scrittura e lettura dei registri vengono ora lanciati con il semplice click del mouse.
- *Semplificare il processo di comunicazione con la Pad per velocizzarne i test*: è stato snellito il procedimento di controllo delle schede *Add-on-board* della Pad eliminando comunicazioni eccessive o ridondanti. Così facendo, si è arrivati ad un miglioramento delle prestazioni temporali dell'ordine di un fattore 8. Infine, sono stati accorpati tutti i test interessanti, al fine dell'esperimento, per rendere più veloce il “check” completo della Pad.

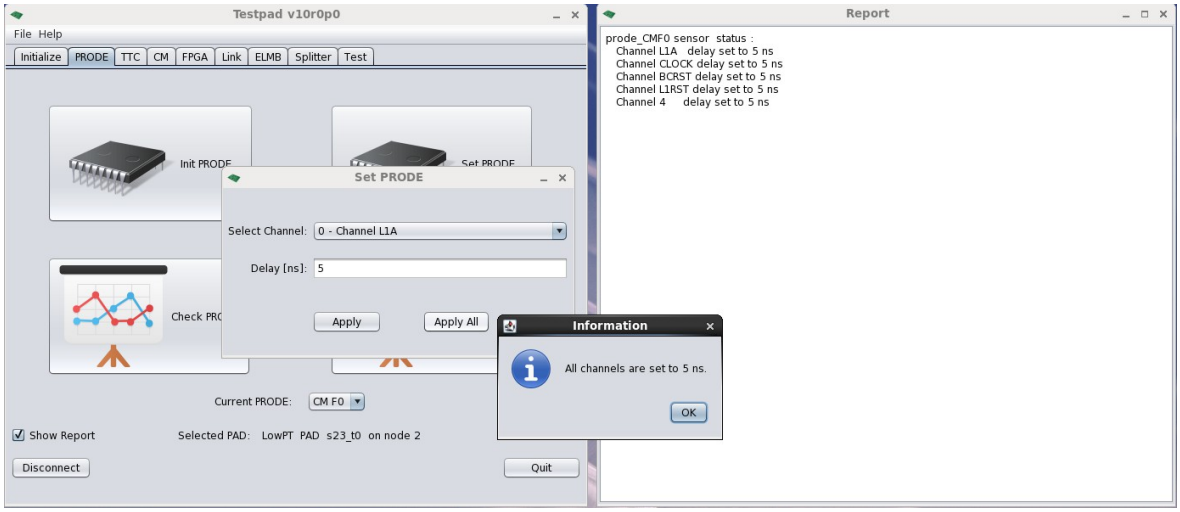

Figura I.2: Interfaccia grafica del software testpad

Infine c'è stata una fase di rifinitura del lavoro a Napoli. Da remoto si è continuato a lavorare sulle problematiche riscontrate in un secondo momento, riguardanti per lo più la configurazione della stazione di test e la limatura di qualche aspetto implementativo.

La tesi, organizzata in quattro capitoli, è così strutturata:

- *Capitolo 1*: viene esposta una panoramica sul rivelatore ATLAS, introducendo brevemente l'acceleratore LHC con i suoi quattro esperimenti e ponendo maggior attenzione su aspetti quali lo spettrometro per i muoni e le camere di trigger RPC.
- *Capitolo 2*: si entra nel dettaglio dell'esperimento ATLAS analizzando il trigger di primo livello. In particolare, si descrive il sistema Pad con la relativa struttura hardware divisa in tre blocchi (*blocco di timing*, *blocco di read-out* e *blocco di controllo ed inizializzazione*). Il capitolo si conclude con un accenno di funzionamento della scheda *Add-on-board ELMB 128*.
- *Capitolo 3*: si descrive la stazione di test messa in piedi, sia dal lato hardware (accennando anche al bus seriale CAN) che da quello software. Di quest'ultimo aspetto si analizza il driver di inizializzazione e controllo (*testpad*) e si valutano le possibili soluzioni ai problemi riscontrati.

- *Capitolo 4*: si analizzano i casi d'uso, i dettagli implementativi ed i test svolti sull'interfaccia grafica sviluppata. Si descrive come avviene la connessione alla scheda Pad, come la si gestisce e quali sono stati i risultati raggiunti dalla nuova modalità di test delle schede *Add-on-board*.

Capitolo 1

1 - Il rivelatore ATLAS

Al fine di cooperare nell'ambito della ricerca della fisica delle particelle, il "*Conseil Européen pour la Recherche Nucléaire*", conosciuto come CERN, fu fondato nel 1954 a Parigi da 12 stati membri (oggi ne sono 20). Lo scopo dell'organizzazione è lo studio dei costituenti fondamentali della materia e delle forze che avvengono tra essa. Il CERN ha due siti principali, uno in Francia, nella zona di Prevessin, e l'altro nella zona nord-ovest di Ginevra, nel sito di Meyrin.
Il programma di ricerca al CERN copre differenti campi scientifici come la fisica, l'ingegneria e l'informatica. Proprio nell'ambito informatico, il CERN è stato il luogo di nascita del "*World Wide Web*" dove, grazie allo scienziato Tim Berners-Lee, si è potuto comunicare a distanza, per poter scambiare efficientemente dati tra chi lavorava a diversi esperimenti, tramite il concetto di "ipertesto" (*Hypertext*). Sempre nello stesso ambito, al CERN va dato anche il merito della creazione del linguaggio di markup HTML (*HyperText Markup Language*), del protocollo HTTP (*HyperText Transfer Protocol*), del primo web server, del primo browser e del primo editor di pagine web.

Il complesso del CERN possiede sette *acceleratori di particelle*, costruiti in differenti periodi, dal primo *Synchrocyclotron* (SC, costruito nel 1957) all'ultimo *Large Hadron Collider* (LHC, avviato il 10 settembre 2008). Fin dall'inizio, fu stabilito che ogni nuovo acceleratore doveva usare gli altri come iniettori, creando così una catena di acceleratori che porta gradualmente un fascio di particelle ad energie sempre più elevate. Tutte le funzionalità degli acceleratori sono gestite da un singolo segnale atomico di clock, distribuito lungo tutta l'installazione, in modo da assicurare il funzionamento corretto della catena.

1.1 L'acceleratore LHC

Il *Large Hadron Collider* (LHC, Fig. 1.1) è il più grande acceleratore di particelle che sia mai stato costruito. È situato presso il CERN di Ginevra ed è utilizzato per studiare le caratteristiche delle particelle subatomiche (anche dette "elementari"). Al progetto partecipano approssimativamente 2000 scienziati, ingegneri ed informatici, divisi tra 165 istituti di 35 nazioni. L'entrata in funzione del complesso, originariamente prevista per la fine del 2007, è avvenuta il 10 settembre 2008.

LHC è un collisore protone-protone composto da due anelli circolari concentrici di diametro 8,4 km (circa 27 km di circonferenza). Queste condizioni fisiche hanno portato ad un'energia teorica di progetto nel centro di massa pari a 14 TeV. L'*elettronvolt*, eV, è l'unità di misura naturale dei fenomeni elettromagnetici; in unità di misura meccaniche è una quantità estremamente piccola: 1 eV = $1.6*10^{-19}$ joule = $3.8*10^{-18}$ calorie. Ad esempio, l'energia che possiede un protone di 2000 GeV equivale a ~$3*10^{-7}$ joule; per confronto, una zanzara con la massa di circa 1 mg, che voli alla velocità di 1 m/s, ha un'energia cinetica di ~ $5*10^{-7}$ joule. Però il volume di una zanzara è ~1 mm^3 mentre un protone ha un volume di 10^{-36} mm^3, quindi un grande acceleratore può essere visto come un concentratore di energia che realizza, in uno spazio piccolissimo, un'eccezionale densità di energia. La massima energia finora raggiunta è di 13 TeV, avvenuta nel maggio 2015.

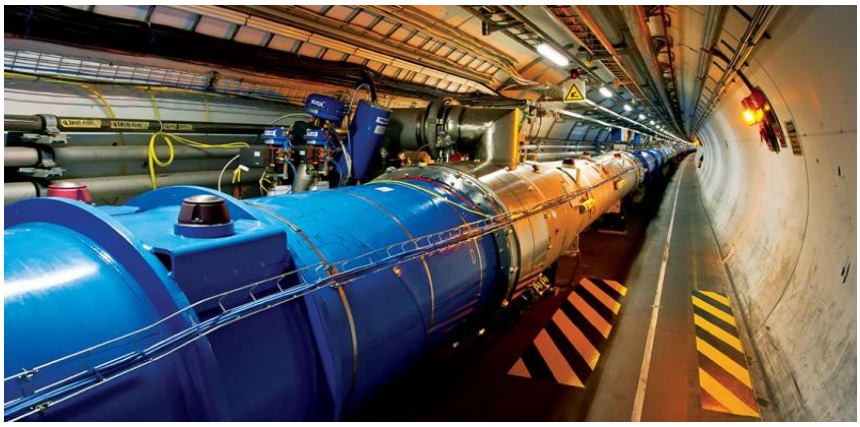

Figura 1.1: L'acceleratore di particelle LHC

Dal primo avvio ha fornito dati a quattro esperimenti: ATLAS (*A Toroidal LHC ApparatuS*), CMS (*Compact Muon Solenoid*) [8], LHCb (*LHC-beauty*) [9] ed ALICE (*A Large Ion Collider Experiment*) [10]. Si tratta di enormi apparati costituiti da numerosi rivelatori che utilizzano tecnologie diverse ed operano intorno al punto in cui i fasci collidono.

La macchina accelera due fasci di particelle che circolano in direzioni opposte, ciascuno contenuto in un tubo a vuoto. Questi collidono (Fig. 1.2) in quattro punti lungo l'orbita, in corrispondenza di caverne nelle quali il tunnel si allarga per lasciare spazio a grandi sale sperimentali. Tra gli scopi principali degli studi vi è la ricerca di tracce dell'esistenza di nuove particelle: il Run 1 di LHC, avvenuto tra il 2009 ed il 2013, ha evidenziato l'esistenza di una nuova particella, compatibile con il bosone Higgs previsto dalla teoria del Modello Standard, responsabile della massa delle particelle elementari. Questo è stato un risultato entusiasmante che ha portato l'8 ottobre 2013 Peter Higgs e François Englert ad essere stati insigniti del premio Nobel per la Fisica per la scoperta del Bosone di Higgs.

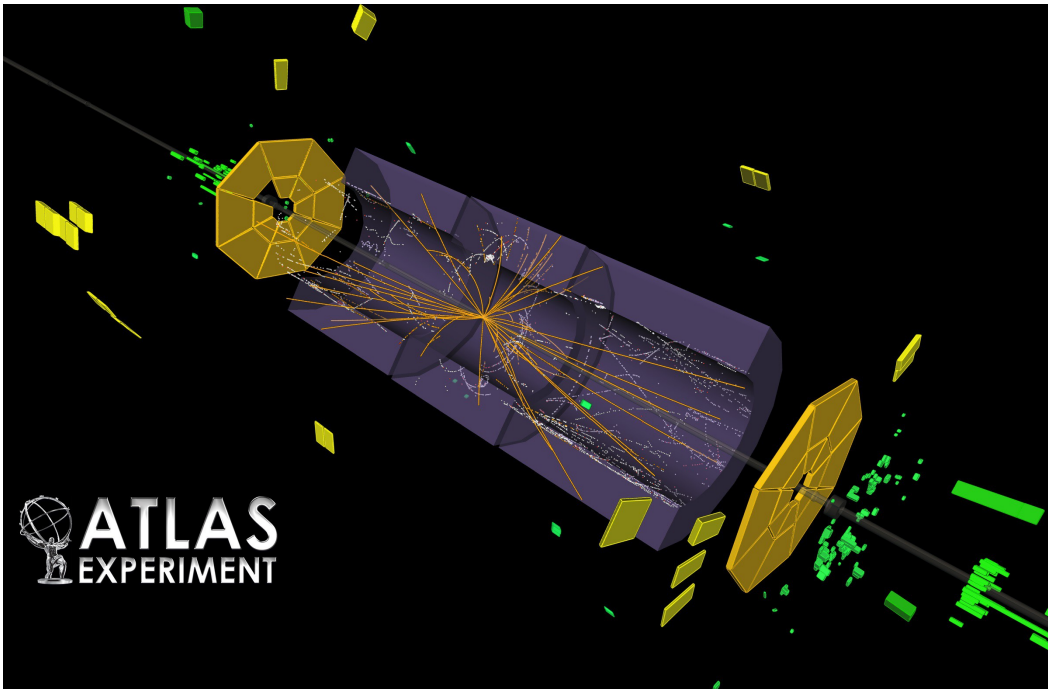

Figura 1.2: Collisioni in ATLAS

1.2 I quattro rivelatori

Come già accennato, LHC fornisce collisioni a quattro rivelatori, situati in altrettante caverne lungo l'anello. I quattro esperimenti sono, riportati in Fig. 1.3:

- **ATLAS**: il suo scopo è una ricerca su tutta la finestra di energia fornita da LHC. Gli obiettivi primari sono lo studio del Bosone di Higgs, la fisica della teoria del Modello Standard e la ricerca di eventuali nuovi fenomeni non previsti da questo modello.
- **CMS**: ha gli stessi obiettivi di ATLAS ma punta su strategie diverse di raggiungimento.
- **LHCb**: progettato per lo studio del decadimento dei mesoni B e la violazione di CP.
- **ALICE**: studia l'interazione forte e il quark-gluon plasma, stato presente, secondo le teorie più accreditate, nei primi istanti successivi al Big Bang.

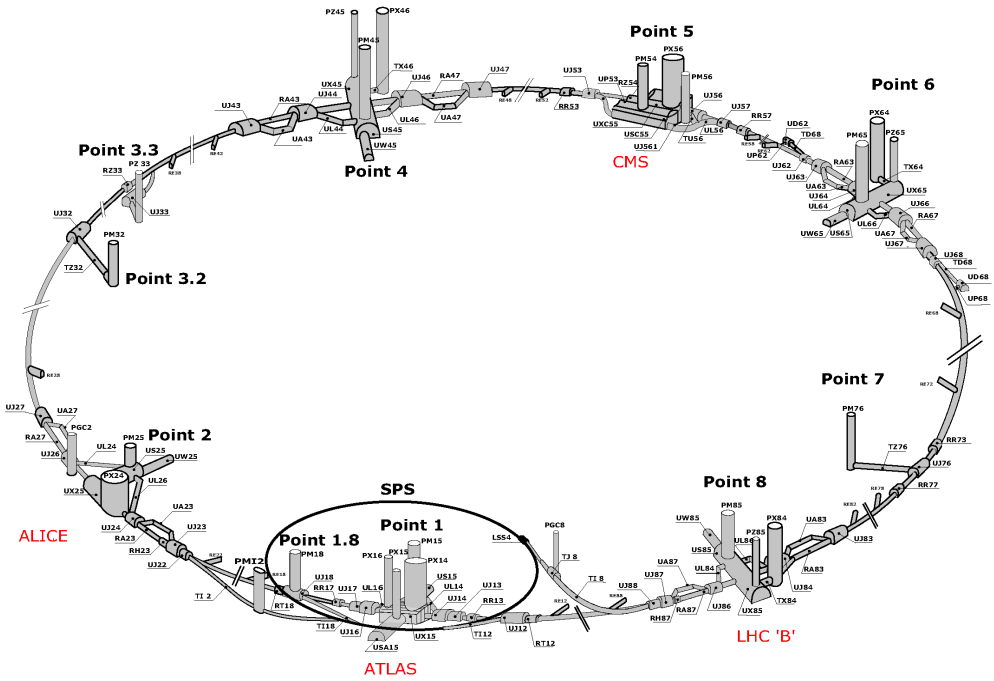

Figura 1.3: Struttura dell'anello di LHC e disposizione dei quattro esperimenti

1.3 ATLAS

Il rivelatore ATLAS è un rivelatore cilindrico (45 metri di lunghezza per 24 di altezza) disposto in modo tale che il suo asse principale coincida con la direzione di volo dei fasci di protoni che orbitano nel tunnel.
L'obiettivo principale di ATLAS, oltre alla ricerca di eventuali segnali di nuova fisica presenti ad alte energie, è stato la scoperta del bosone di Higgs. All'esperimento fanno parte fisici, tecnici ed ingegneri di tutti gli stati membri ed un grosso contributo è stato dato dalle università e dai centri di ricerca italiani (tra cui l'Università degli Studi di Napoli "Federico II" e l'INFN). L'INFN ha contribuito con 19 sedi ed, in particolare, la sezione INFN di Napoli si è occupata della ricerca sui muoni, della ricerca sul bosone di Higgs, dello spettrometro a muoni e dei rivelatori RPC. Inoltre parte del sistema di computing e di storage distribuito di ReCAS [11] è stato messo a disposizione per l'analisi dei dati acquisiti nel rivelatore.

Alla luce di questi obiettivi, il progetto ATLAS ha sviluppato le seguenti caratteristiche:

- un *inner detector* per la misura molto precisa delle tracce lasciate dalle particelle;
- un sistema di *calorimetri* per la misura dell'energia delle particelle;
- uno *spettrometro*, situato nella regione più esterna, utilizzato per rivelare i muoni (particelle con carica elettrica negativa, fondamentali per lo studio della fisica che interessa l'esperimento).

L'intero esperimento sfrutta gli effetti dei campi magnetici, sulle particelle cariche, per misurare la velocità e l'energia delle particelle stesse. In particolare, l'inner detector è contenuto in un magnete che genera un campo magnetico, parallelo alla direzione dei fasci, mentre lo spettrometro a muoni utilizza un campo toroidale generato dalle bobine esterne.

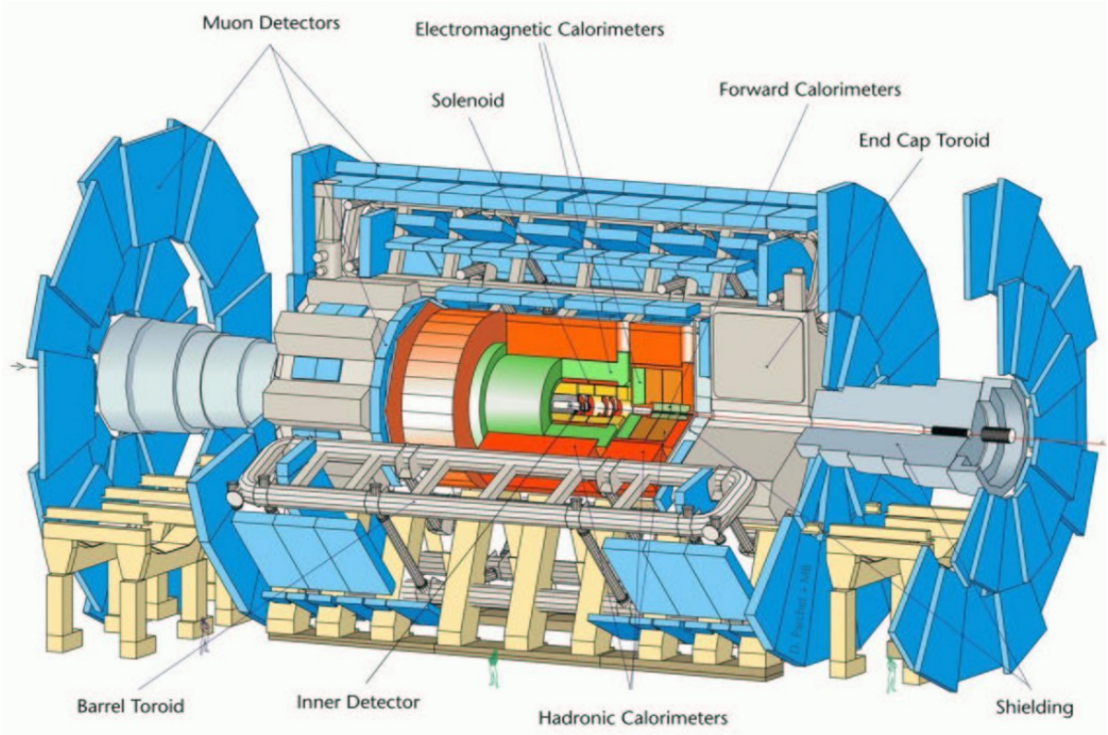

Figura 1.4: Sottosistemi del rivelatore ATLAS

Il rivelatore ATLAS può essere diviso in tre macro-parti (Fig. 1.4): la parte centrale del cilindro è detta *barrel*, la zona delle due basi è chiamata *end-cap* mentre quella intermedia tra le due è detta di *transizione*.

1.3.1 Lo spettrometro per i muoni

Pur essendo ogni sottorivelatore molto complicato, e meritevole di attenzione, in quanto progettato e realizzato appositamente per ottimizzare la rivelazione e l'acquisizione dei segnali interessanti per la fisica da studiare in ATLAS, ci concentriamo solo sullo spettrometro in quanto la presente tesi riguarda questa specifica parte di ATLAS.

Lo spettrometro a muoni è la parte più esterna di ATLAS: si basa sulla curvatura della traiettoria dei muoni (particelle cariche molto penetranti che riescono ad attraversare l'intero apparato) generata dal campo magnetico toroidale (*Barrel*

Toroid ed *End Cap Toroid*) ed è dotato di un trigger e di camere tracciantiindipendenti dal resto del rivelatore.

La struttura dello spettrometro per muoni (Fig. 1.5) indica le differenti regioni in cui sono collocati i rivelatori che costituiscono lo spettrometro stesso. Le camere del *barrel* formano tre cilindri concentrici con l'asse di direzione del fascio, i cui raggi sono 5, 7.5 e 10 *m*. Le camere dell'*end-cap* sono collocate su quattro dischi distanti 7, 10, 14 e 23 *m* dal punto d'interazione dei fasci. Viste le enormi dimensioni dello spettrometro (l'area coperta dal rivelatore è superiore ai 3000 m^2), si è scelto di utilizzare rivelatori a gas più economici e più facili da realizzare rispetto ai rivelatori interni.

Il sistema è composto da quattro tipi di camere, due che misurano la posizione (camere di precisione) e due che rivelano il passaggio di particelle e la loro velocità (camere di trigger):

- **Camere di precisione** che si dividono in:
  - **Monitored Drift Chambers (MDT)**: nella regione del barrel;
  - **Cathode Strip Chambers (CSC)**: nella regione dell'end-cap.
- **Camere di Trigger**: hanno, come detto, un duplice scopo: identificare il passaggio di particelle penetranti che hanno attraversato i calorimetri e misurare grossolanamente la curvatura (e quindi l'impulso) per rigettare particelle non interessanti. Le camere di trigger si dividono in:
  - **Resistive Plate Chambers (RPC)**: sono rivelatori a gas, presenti nella regione del barrel, di cui si parlerà nel prossimo paragrafo;
  - **Thin Gap Chambers (TGC)**: sono camere a filo molto resistenti alle radiazioni e per questo motivo sono state collocate nella regione dell'end-cap.

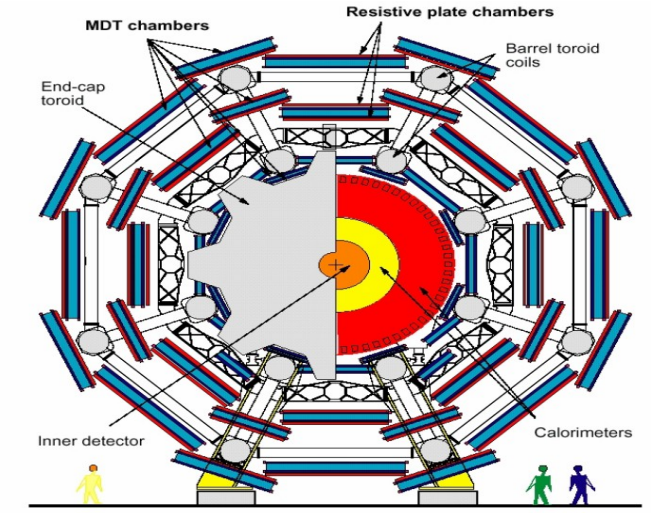

Figura 1.5: Sezione dello spettrometro a muoni del barrel.
In rosso gli strati RPC

1.3.2 Le camere di trigger RPC

Gli RPC (Fig. 1.6) sono rivelatori a gas (tipicamente viene usata una miscela di tetrafluoretano, isobutano e freon) nei quali il gas è confinato tra due piani metallici (elettrodi) distanti pochi millimetri tra loro. I rivelatori utilizzano un elevato campo elettrico (circa 4.5 kV/mm) ed hanno risoluzione spazio-temporale dell'ordine di 1 $cm * 1\ ns$.

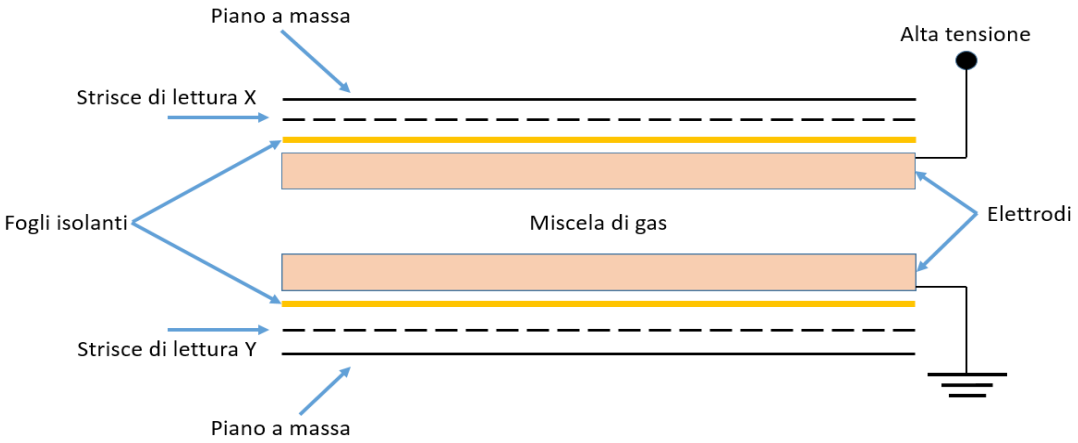

Figura 1.6: Struttura di una camera RPC con indicati tutti i componenti

Il principio di funzionamento degli RPC in ATLAS è il seguente: gli elettroni, prodotti nel gas dal passaggio di una particella, vengono accelerati dal campo elettrico ed urtano altri atomi, liberando altri elettroni. La cascata di elettroni che si crea genera un segnale elettrico indotto sulle strip di lettura. Il fatto di avere confinato il gas tra due piani di materiale isolante permette di localizzare nello spazio il deposito di carica e quindi il passaggio della particella, cosa impossibile con l'uso di un materiale conduttore.

I segnali prodotti dalla ionizzazione del gas vengono indotti su due set di strisce di lettura (*strip*, in rame con larghezza 3 *cm*) ortogonali tra loro poste sulle superfici esterne dei piani. In questo modo, si ottengono informazioni bidimensionali sul passaggio della particella. Le principali caratteristiche degli RPC usati in ATLAS sono le seguenti:

- durata del segnale < 10 *ns*;
- tempo di risposta < 3 *ns*;
- risoluzione temporale ~ 1 *ns*;
- campo elettrico ~ 4,5 *kV/mm*.

Ogni stazione RPC è formata da due piani di rivelatori, ognuno con due pannelli di *read-out*. Ogni stazione di trigger è composta da due camere RPC adiacenti; le zone attive adiacenti sono sovrapposte allo scopo di eliminare le zone morte. Tutti gli elementi sono tenuti rigidamente uniti da due pannelli di supporto che rendono compatta l'intera struttura meccanica delle camere. Il numero di RPC nella zona del barrel è 1052 con circa 430000 canali di read-out.

Capitolo 2

2 - Il trigger di primo livello

L'obiettivo primario del sistema di *Trigger/DAQ* (TDAQ) è quello di separare con elevata efficienza i processi di fisica interessanti dalla maggioranza degli eventi (anche detti *di fondo*) e memorizzare i dati su supporto permanente. In un esperimento di fisica delle alte energie il trigger è costituito da un insieme di dispositivi sia hardware che software aventi, quindi, il compito di decidere in tempo reale se un dato evento rientra nella gamma di fenomeni di fisica in studio. Nel caso di risposta positiva, tutte le informazioni relative all'evento vengono salvate per poter essere analizzate off-line, altrimenti l'evento viene ignorato. Per questo motivo solitamente il trigger è un "OR" logico tra varie condizioni che dipendono dall'obiettivo dell'esperimento. L'effetto dell'azione già detto sopra del sistema di trigger è quello di abbassare il rate di eventi da quello iniziale, dovuto dalle interazioni, fino ad un rate tale da poter essere scritto su nastro in maniera tale da mantenere un'efficienza alta per gli eventi interessanti.

In ATLAS il trigger è organizzato in tre livelli successivi (primo livello – LVL1, secondo livello – LVL2 ed Event Filter – EF), in maniera tale che al terzo livello, ossia dove la selezione viene fatta in maniera più accurata ma con maggior tempo per l'analisi, arrivino solo quegli eventi che hanno passato le richieste dei livelli precedenti (Fig. 2.1). Ognuno dei tre livelli riduce il rate di eventi da 40 MHz a circa 200 Hz, in modo tale da rendere possibili le operazioni dei livelli successivi.

Il trigger di primo livello (LVL1) è un sistema hardware costituito da processori dedicati, alcuni dei quali posti direttamente sui rivelatori, che operano una scelta rapida e mirata degli eventi, analizzando i dati provenienti dal calorimetro e dalle camere di trigger dello spettrometro. La latenza del LVL1, cioè il tempo impiegato per prendere le decisioni e spostare i dati verso il LVL2, è di circa 2 μs. La

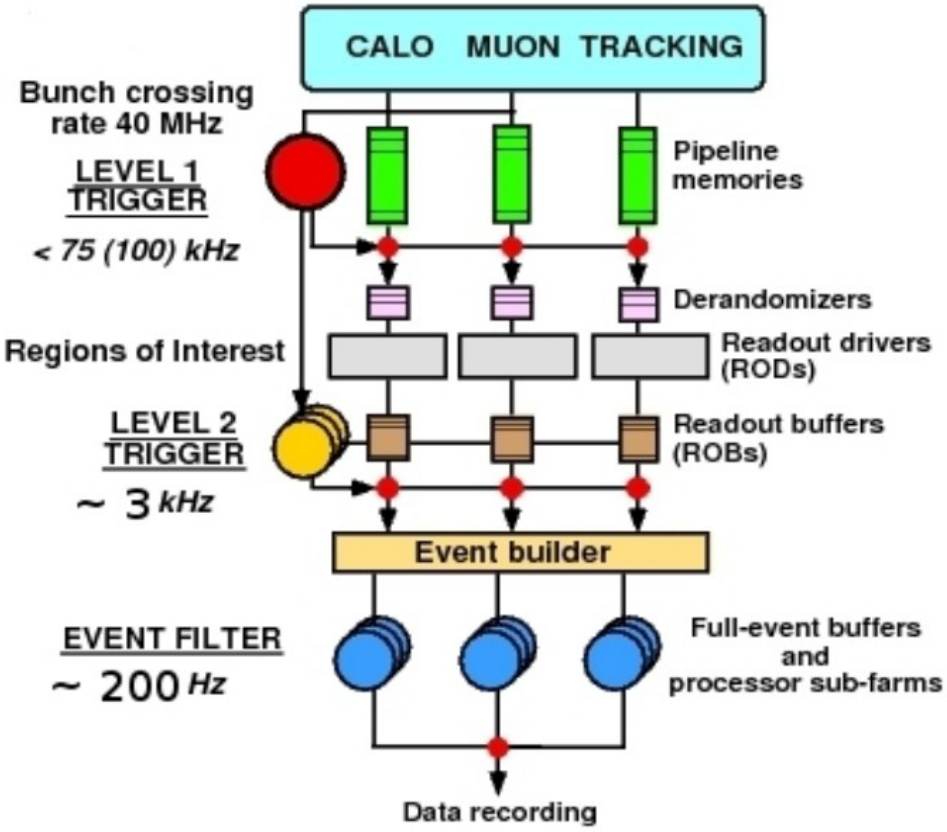

Figura 2.1: Schema dell'intero sistema di trigger e di acquisizione dati di ATLAS

decisione vera e propria del trigger di primo livello viene presa dal *Central Trigger Processor* (CTP) il quale confronta i dati ricevuti con i requisiti preimpostati (anche detto *menù di trigger*). Se l'evento analizzato ha le caratteristiche corrispondenti ad una delle voci del menù di trigger, il CTP manda un segnale, detto *LVL1-Accept,* a tutti i dispositivi del primo livello che rispondono inviando i propri dati (relativi all'evento in questione) tramite il *ReadOut System*.

Il trigger di livello 2 (LVL2) insieme a quello del livello 3 costituiscono i trigger di alto livello (*High Level Trigger*) ed utilizzano componenti hardware e software, integrati con il sistema di acquisizione dati e posti fuori dal rivelatore (*off-detector*). Una volta che l'evento è stato accettato dal primo livello, il sistema di acquisizione dati informa il secondo livello di trigger che un nuovo evento è pronto per essere analizzato. Per fare ciò il livello 2 dovrebbe andare a leggere tutti i canali del rivelatore alla ricerca dei dati interessanti; tuttavia, dato il grande numero di canali

di lettura presenti in ATLAS (dell'ordine di 10^8), questa operazione risulta essere proibitiva a causa del tempo richiesto. Per ovviare a questo problema, il LVL1 indica al LVL2 la regione (in termini di η e ϕ) in cui si trova l'evento che ha fatto scattare il trigger. Queste aree sono chiamate *Region of Interest* (RoI). Le RoI sono mandate dal primo livello al secondo livello in maniera tale che il livello 2, basandosi sulle informazioni delle RoI, possa accedere solo ai dati corrispondenti a pochi *Read Out Buffer* (le pipeline in cui vengono conservati i dati fino alla decisione del trigger), ed andare a guardare in dettaglio solo quelli necessari per prendere una decisione. Grazie a questo meccanismo delle RoI, solo pochi dati dell'intero evento sono necessari al trigger di secondo livello per prendere una decisione. La latenza del LVL2 è variabile da evento ad evento ma deve essere comunque dell'ordine dei 10 ms per non introdurre tempo morto rispetto al rate del primo livello.

Infine la selezione finale dei dati è fatta dall'*Event Filter* (EF), o trigger di terzo livello, che si basa esclusivamente sul codice off-line ed esegue un'accurata ricostruzione dell'evento combinando le informazioni di tutti i sotto-detector interessati (costanti di allineamento, calibrazione, mappa completa del campo magnetico).

2.1 Il sistema Pad

L'algoritmo di trigger di primo livello [12] per la regione centrale (*barrel*) dello spettrometro, basata sui rivelatori RPC, è implementato da una complessa struttura hardware che si articola in due tipi: l'elettronica *on-detector* e quella *off-detector*. L'elettronica *on-detector* è alloggiata direttamente sulle camere RPC come si può vedere in Fig. 2.2.

L'algoritmo di trigger di primo livello e la trasmissione elettronica dei dati sono eseguiti da una scheda disegnata, ad hoc, nei laboratori INFN, chiamata **Pad**,

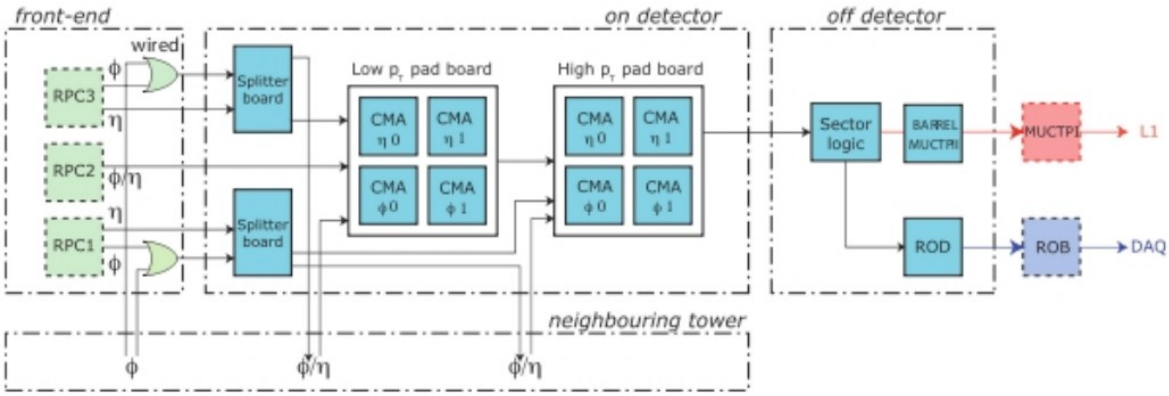

Figura 2.2: Elettronica on-detector ed off-detector del trigger

montata sulle camere RPC. In base al tipo di algoritmo che devono implementare, le Pad si divino in Pad di *alto* p_T e Pad di *basso* p_T.

Essendo le schede Pad posizionate direttamente sul rivelatore, esse devono affrontare una serie di problematiche particolarmente critiche. Le componenti elettroniche di cui è costituita una Pad devono resistere ad un campo magnetico e radioattivo molto intenso, senza avere alcuna perdita di dati (sia inerenti all'esperimento che alla configurazione dell'hardware stesso); è quindi importante che le schede Pad siano resistenti a tali situazioni ambientali, data la difficoltà di intervenire "fisicamente" per la sostituzione di una componente difettosa o guasta.

La scheda Pad è composta da una mother-board sulla quale sono montate le seguenti schede *Add-on-board*:

- le **Matrici di Coincidenza** (CM), ossia i chip progettati per la prima analisi dei dati provenienti dagli RPC per il trigger di primo livello;
- il **TTC**, per inviare le informazioni di sincronizzazione (*clock*) e controllo di LHC dal sistema centrale e di distribuirli a tutti i componenti della Pad;
- la **FPGA** (*Field Programmable Gate Array*), un circuito integrato riprogrammabile e tollerante alle radiazioni, le cui funzionalità sono programmabili da remoto;

- la **ELMB** (*Embedded Local Monitor Box*), una scheda *general-purpose* a basso costo usata per il controllo e l'inizializzazione dei vari componenti della Pad;
- il **GLinkTX**, un link ottico di uscita che trasferisce i dati, elaborati dalla Pad, su fibra ottica verso l'elettronica *off-detector* (presente solo sulle Pad di *high* p_T).

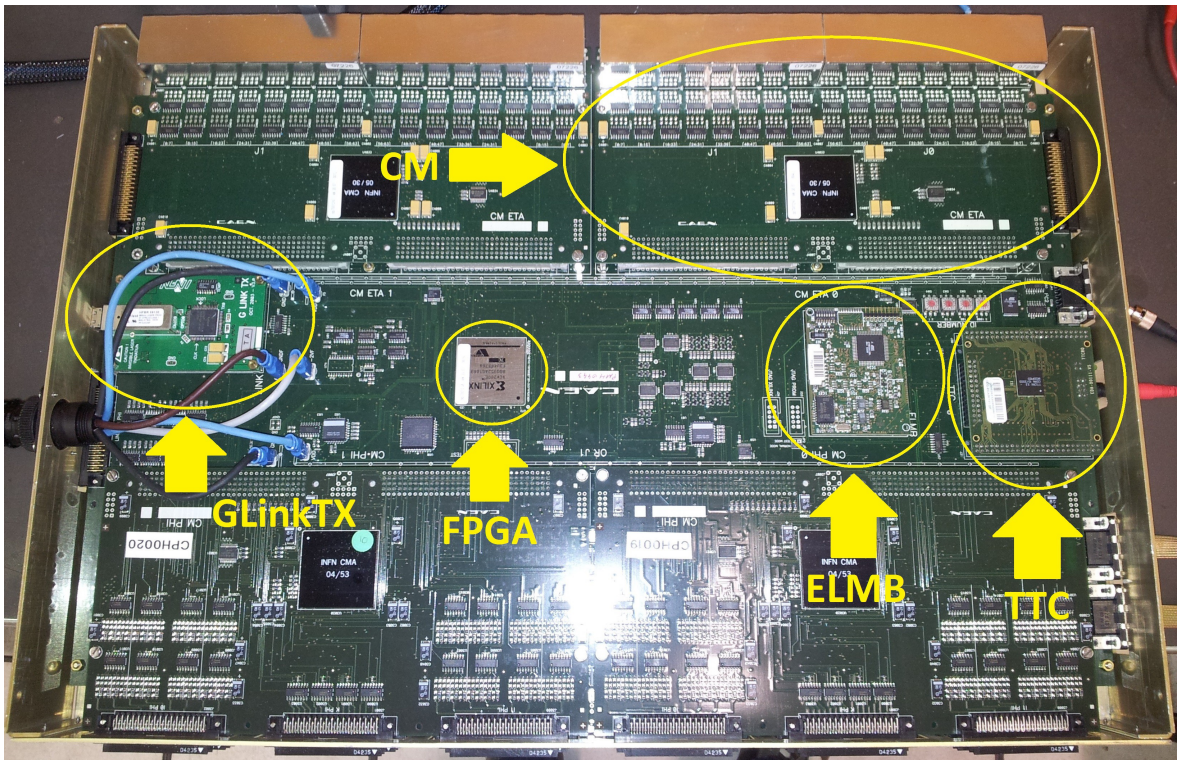

Figura 2.3: Sistema Pad con schede Add-on-board evidenziate

Nei successivi paragrafi saranno descritte le caratteristiche hardware e funzionali degli elementi che compongono la Pad.

2.2 Struttura hardware della Pad

Descriviamo ora le differenze tra Pad di $high_{PT}$ e di low_{PT}. La Pad di *basso* p_T invia le proprie informazioni, ricevute serialmente dalle quattro CM (descritte

successivamente), ed il pacchetto di dati di *trigger* alla Pad di *alto* p_T. Quest'ultima, tramite l'FPGA, raccoglie ed elabora i dati di *read-out* e quelli di *trigger* di entrambe le Pad ed implementa l'algoritmo di *trigger*, per poi formattare tutti i dati e spedirli via *link ottico* all'elettronica *off-detector*.

A parte le differenze riportate nella tabella seguente, le due Pad sono identiche e ciascuna di essa può essere divisa in quattro blocchi funzionali:

- blocco di *timing;*
- blocco di *read-out*;
- blocco di controllo ed inizializzazione;
- connettori ed altri servizi.

	PAD Low	**PAD High**
Collocazione	Montata sugli RPC2, riceve i segnali dalle RPC1 e RPC2	Montata sugli RPC3, riceve i segnali dalle RPC3 e dalle PAD di Low p_t
Connettori di non pivot delle CM	Ricevono i segnali dalle camere RPC1 attraverso gli Splitter	Ricevono i segnali dalle camere RPC3 attraverso gli Splitter
Connettori di pivot delle CM	Ricevono i segnali direttamente dalle RPC2	Ricevono il segnale di trigger (K-pattern) dalle CM della PAD low
FPGA	Gestisce i segnali di Test Pulse	implementa la logica di trigger, i segnali di Testpulse, gestisce i dati di read-out e genera i test di controllo
Link ottico d'uscita (G^2 $LinkTX$)	Assente	S'interfaccia con la Sector Logic
Link di PAD (LPAD-HPAD Link)	Settati come driver, servono ad inviare i segnali di trigger e read-out alla PAD High	Settati come receiver, servono a ricevere i segnali di trigger e read-out provenienti dalle PAD Low

Tabella 2.1: Principali differenze tra la Pad low e la Pad high

Come mostrato dalla tabella, le differenze sostanziali tra i due tipi di Pad risiedono nella ricezione dei segnali di trigger (il tipo *high* riceve anche i segnali provenienti dalle quattro CM del tipo *low*), nell'utilizzo della FPGA (la Pad *high* implementa l'algoritmo di trigger mentre la *low* si limita a gestire i segnali di Test Pulse, ossia cariche elettriche utilizzate per simulare gli eventi interessanti) e nella mancanza o meno della scheda *Add-on-Board GLinkTX* (presente solo sulla Pad di tipo *high* per potersi interfacciare con la Sector Logic). Degli *Splitter* se ne parlerà nei successivi paragrafi.

2.2.1 Il blocco timing

Il blocco di timing distribuisce il *clock*, i segnali di L1A (*Level 1 Accepted*) e quelli di BCNTR (*Bunch CouNTeR*) ed EVNCTR (*Event CouNTeR*) a tutti i componenti della scheda Pad e controlla la loro sincronizzazione in fase. Esso è composto da tre componenti: il **TTCrx**, i chip **PRODE** ed il **Phase Detector**.

Il **TTCrx** è un ASIC (*Application Specific Integrated Circuit*, circuito integrato realizzato per applicazioni specifiche) personalizzato per ricevere informazioni di sincronizzazione e controllo dal sistema centrale del TTC e di distribuirli a tutti i componenti della Pad. Questi segnali sono il clock di LHC (pari a 40 MHz, con il minimo *jitter*), il L1A, il BCRST (*Bunch Counter Reset*), l'ECRST (*Event Counter Reset*) ed i segnali interni di BCNTR ed ECNTR.
Il TTCrx genera, oltre al *Clock40* (in fase con quello di LHC), altri due diversi tipi di clock: il *Clock40Des1* (*Clock de-skewed*) ed il *Clock40Des2*. Questi due segnali possono essere programmati per essere sfasati rispetto al segnale di clock a 40 MHz: il primo ha 16 ritardi di fase diversi di circa 1,5 ns, mentre il secondo ne ha 240 di circa 10 ps ed è utilizzato dal trasmettitore ottico *GLinkTX*.
Il TTCrx è provvisto di molti registri interni usati per il controllo e per la sincronizzazione dei segnali. Essi sono:

- Il *Configuration Register* contiene i bit di configurazione letti serialmente dalla PROM esterna durante l'inizializzazione. In esso sono memorizzati i 14 bit dell'ID del TTCrx, i dati per l'esecuzione di alcune operazioni e le modalità di test.
- Il *Control Register* permette di abilitare o disabilitare le diverse funzioni della scheda per minimizzare il consumo energetico.
- L'*Event Counter Register* è un contatore a 24 bit incrementato dal segnale di L1A. Può essere resettato in due modi: con il comando di *broadcast* ECRST oppure dalla procedura di inizializzazione che avviene tramite il bus seriale I^2C (vedi App. B).
- Il *Bunch Counter Register* è un contatore a 12 bit incrementato dal segnale di clock di LHC. Può essere azzerato in due modi: con il comando di *broadcast* BCRST oppure dalla procedura di inizializzazione che avviene tramite I^2C.
- L'*Error Counter Register* è in grado di contare il numero di errori tramite tre registri interni (*Single Bit Error, Double Bit/Frame Counter*, *SEU Counter*).

Il chip **PRODE** è un ASIC usato per compensare i ritardi nella distribuzione di uno qualsiasi dei segnali di timing ricevuto dal TTCrx (il CLK40, il L1A, il Bunch Counter Reset e l'Event Counter Reset) ai componenti della Pad. La sua resistenza alle radiazioni ed il basso consumo energetico sono adatte alle richieste di ATLAS.
La Pad ha complessivamente quattro chip PRODE indipendenti, uno per ciascuno dei quattro segnali di timing da sincronizzare. La programmazione del ritardo sui quattro canali avviene sempre tramite I^2C.
Al suo interno è implementa una logica di *auto-reset* che, all'accensione, permette l'inizializzazione di tutti i suoi registri interni. Il chip ha una struttura semplice basata su 4 linee di ritardo (*delay line*), un interfaccia I^2C (*I^2C Interface*) per la configurazione esterna ed un DLL (*Delay Locked Loop*) per il controllo della temporizzazione del chip.

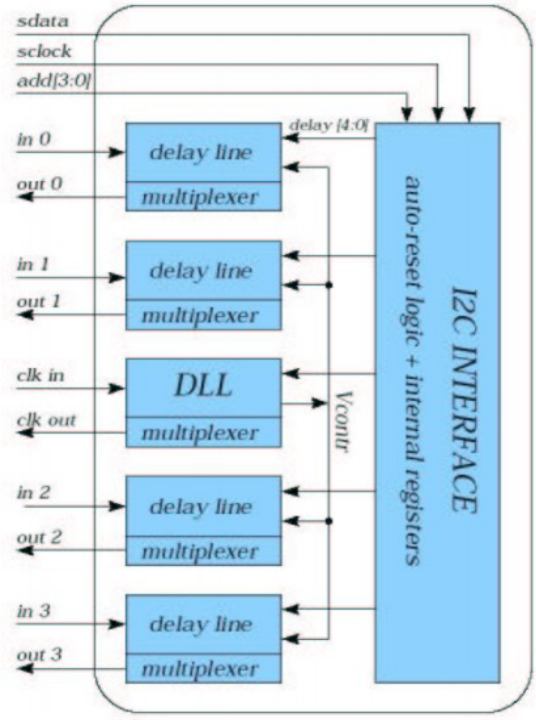

Figura 2.4: Architettura del chip PRODE

Il **Phase Detector** (PD) controlla le fasi dei vari segnali di clock, sia fra le CM di una Pad sia fra le due Pad di *basso* ed *alto* p_T. Il Phase Detector è dotato di un multiplexer ad otto ingressi, ognuno dei quali associato ad un clock di uscita di una delle otto CM. Di volta in volta, il clock R_2 in uscita dal PD (chiamato anche segnale di *feedback*) è confrontato con il segnale di riferimento R (il CLK40), rispetto al quale dev'essere allineato [15].

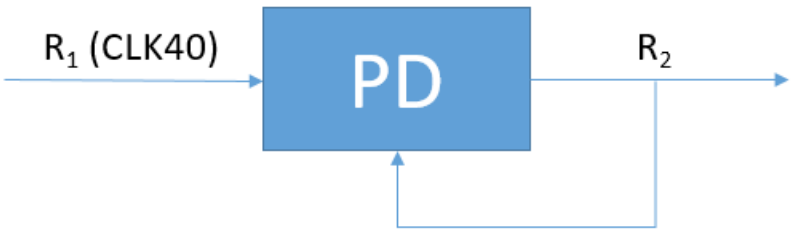

Figura 2.5: Schema a blocchi del funzionamento del Phase Detector

2.2.2 Il blocco di read-out

Il blocco di *read-out* gestisce la lettura dei dati provenienti dagli **Splitter** ed è composto da quattro componenti: la **scheda OR**, le quattro **CM**, la **FPGA** ed il **link ottico**.

Lo scopo delle schede **Splitter** è quello di convertire i segnali, provenienti dalle camere RPC, da **logica ECL** (*Emitter Coupled Logic*, è una logica di tipo dipolare, ad un bit corrispondono due canali) a **logica LVDS** (*Low Voltage Differential Signaling*) e di trasmettere tali segnali su diverse Pad o su due ingressi della stessa Pad.
La scheda degli Splitter è costituita da una *motherboard* fornita di sensori di temperatura, sensori di tensione e di un connettore I^2C tramite il quale la Pad collegata può leggere i vari sensori. Sulla *motherboard* sono montate 8 schede Splitter, 4 di tipo η e 4 di tipo ϕ.

La **scheda OR** si occupa di mettere in OR logico i segnali provenienti da tre Splitter associati a tre camere RPC adiacenti. Ogni Pad è provvista di due schede OR, la **OR-J0** e la **OR-J1**, che si riferiscono ai due piani di *strip* di ciascuna camera RPC.

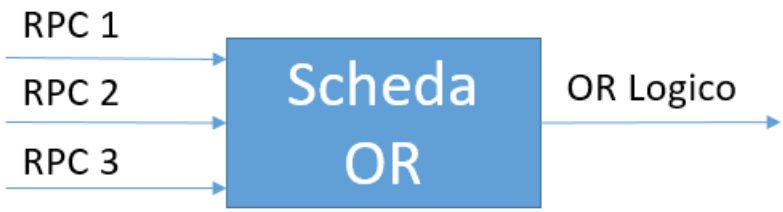

Figura 2.6: Schema a blocchi del funzionamento della scheda OR

La scheda **CM** (*Coincidence Matrix*) ha come componente principale il chip **CMA** (*Coincidence Matrix ASIC*); le sue funzioni sono:

- ricevere e temporizzare i segnali dall'elettronica di *front-end* degli RPC;

- mascherare i segnali indesiderati di ingresso e di uscita;
- eseguire l'algoritmo di *trigger*, identificare le tracce di muoni candidati e classificarli in base ai loro valori di p_T ;
- memorizzare i dati di *read-out* durante tutto il periodo di latenza per poterli eventualmente trasmettere, se convalidati da un segnale di *L1Accept*;
- formattare i dati e trasmetterli serialmente alla FPGA.

I dati in ingresso, dopo essere stati sincronizzati e formattati, vengano trasmessi nel blocco di memorie per poi proseguire in due percorsi diversi: verso il blocco di *read-out* e verso il blocco di *trigger*. Il blocco di *trigger* esegue il suo algoritmo per tre volte, in modo tale da aprire altrettante finestre di coincidenza contemporaneamente e fornire differenti tagli sull'impulso trasverso del muone.
I dati di *trigger* a 32 bit vengono poi inviati in parte all'uscita del *chip* (questi dati prendono il nome di *k-pattern*) ed in parte al blocco di *read-out*, sincronizzato con il clock a 40 MHz.
Il blocco di *read-out* raccoglie i dati uscenti dal blocco di *trigger* e quelli provenienti dall'ingresso del chip in memorie FIFO che memorizzano i dati per tutto il periodo di latenza del LV1. I dati riconosciuti validi, tramite il segnale di *L1Accept*, sono formattati ed inviati ai *buffer* che li rendono pronti alla lettura seriale.

La **FPGA** montata sulla Pad è la Xilinx Virtex XCV200BG352 basata su architettura CMOS e tollerante alle radiazioni. La FPGA della Pad High si occupa di implementare la logica dei dati e di *trigger*.
Essa riceve il clock a 40 MHz deskewed-1 dal TTCrx; il firmware implementato è scritto in linguaggio Verilog.
La FPGA è costituita da un array di porte logiche elementari ed il progettista può programmare, via software, le interconnessioni tra i vari elementi del circuito. Una caratteristica fondamentale è che il device può essere riprogrammato più volte, in

base alla tecnologia di implementazione (SRAM, *Static Random Access Memory*). Ciò ha reso questo tipo di tecnologia sempre più importante nel campo dell'elettronica industriale e, nel corso degli ultimi due decenni, il continuo progredire delle tecniche di miniaturizzazione ha portato il numero di porte logiche da poche migliaia a qualche milione per singola FPGA.

I vantaggi dell'utilizzo di FPGA rispetto ad altri circuiti integrati sono:

- sono adatte ad un gran numero di applicazioni dato che vengono programmate direttamente dall'utente finale;
- bassi tempi di progettazione e di simulazione;
- correzione degli errori in tempo reale riprogrammando il circuito;
- ambienti di progettazione di facile apprendimento.

Tra le principali tecnologie di implementazione, oltre alla SRAM, troviamo:

- *Antifusibile*: programmabile una sola volta;
- *EPROM* (*Erasable Programmable Read Only Memory*): programmabile più volte mediante radiazione ultravioletta;
- *EEPROM* (*Electrically Erasable Programmable Read Only Memory*): programmabile più volte mediante via elettrica;
- *Flash*: simile alla EEPROM ma utilizzando memorie a stato solido, di tipo non volatile.

Più nel dettaglio, come anche rappresentato in Fig. 2.7, la struttura di una FPGA è costituita da CLB (*Configurable Logic Blocks*), connessi fra loro da interconnessioni programmabili, e da blocchi di ingresso/uscita chiamati *I/O Block*. Quindi, in generale, i CLB realizzano le funzioni logiche e, tramite le interconnessioni, si arriva agli I/O Block che si interessano dell'interfacciamento con il mondo esterno. Scendendo ancora più in profondità, all'interno dei CLB troviamo le LUT (*Look Up Table*), utilizzate per implementare le funzioni booleane generiche. Queste sono composte da una memoria SRAM da 16 bit e da un

multiplexer a 4 ingressi; le connessioni di più CLB permettono, poi, di realizzare funzioni booleane più complesse.

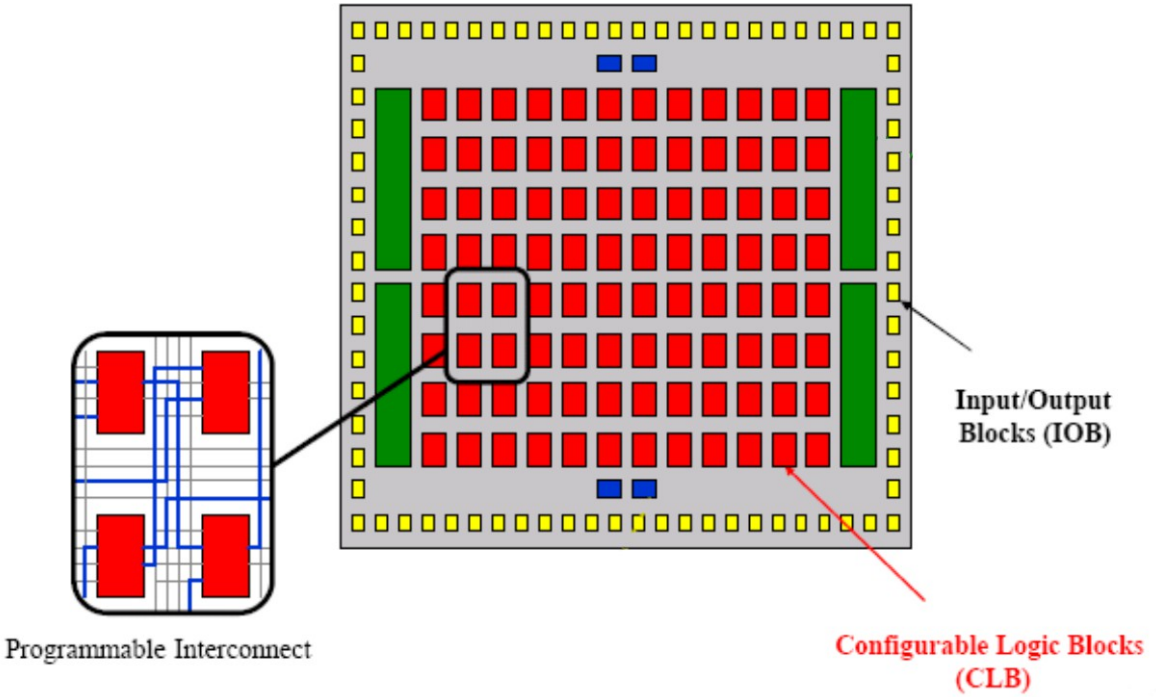

Figura 2.7: Struttura FPGA

Il **link ottico** utilizzato per trasmettere i dati memorizzati nell'FPGA è il *G²LinkTX*. Esso, insieme alla scheda di ricezione montata sulla Sector Logic (*G²LinkRX*), è stato progettato, disegnato e realizzato nei laboratori dell'INFN di Napoli.

La scheda di trasmissione è montata sulla Pad di *high* p_T ed è composta da un serializzatore commerciale che trasmette in maniera seriale 16 bit ogni 40 MHz; è inoltre equipaggiata con un laser ottico [13] per la trasmissione su fibra ottica dell'informazione *off-detector* ogni 25 ns con bandwidth di 800 Mbit/s. L'intero sistema ha due canali di trasmissione a 16 bit: nel canale 0 sono trasmessi i dati di *read-out*, mentre nel canale 1 quelli di *trigger*.

La scheda *G²LinkTX* (transmitter) [14] invia i dati alla scheda *G²LinkRX* (receiver) montata sulla scheda VME_RX. Il suo funzionamento è regolato dal clock-deskewed-2, ricevuto dal TTCrx, e dal segnale di abilitazione della trasmissione della FPGA. La scheda è configurabile tramite I²C, col quale è possibile anche

visualizzare informazioni di controllo relative alle temperature ed alle tensioni di alimentazione.

2.2.3 Il blocco di controllo ed inizializzazione

Il blocco di controllo e di inizializzazione si occupa di svolgere una serie di funzioni mirate al controllo delle attività dei singoli componenti che compongono la Pad ed alla loro configurazione. Per poter adempiere a questi compiti, la Pad è stata fornita di un sistema di comunicazione che le permette di dialogare con l'elettronica *off-detector* e con i componenti della Pad. La gestione di tale comunicazione è completamente affidata all'**ELMB**. Dal punto di vista della comunicazione esterna, l'ELMB costituisce il nodo *CAN* (si veda App. A) attraverso il quale la Pad comunica con un modulo USB-CAN a 16 porte della Systec, collegato ad un PC, che assolve al ruolo di server CAN e tramite il quale si possono lanciare una serie di comandi per coordinare l'attività dell'elettronica di *read-out* e di *trigger*.

2.3 ELMB 128

L'*Embedded Local Monitor Box* è una scheda *general-purpose* a basso costo usata per il controllo e l'inizializzazione dei vari componenti della Pad.

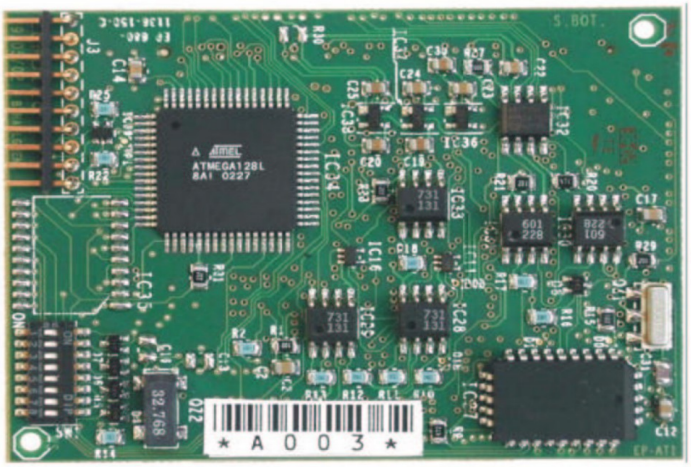

Figura 2.8: La ELMB 128

La ELMB è stata scelta per la sua tolleranza alle radiazioni presenti nella regione in cui è collocato ATLAS e per la possibilità di programmarla da remoto senza la necessità di rimuoverla dal sistema che controlla. Inoltre è provvisto di un alto numero di canali di I/O attraverso i quali può assolvere le funzioni di controllo ed inizializzazione che le sono richieste.

La ELMB 128 è divisa in tre regioni alimentate separatamente. Il **blocco CAN**, alimentato a 5V, è costituito dal ricetrasmettitore CAN PCA82C25 della Philips che è l'interfaccia fra il controllore del protocollo CAN (CAN controller) ed il CAN bus. Il **blocco digitale** è alimentato a 3.3V ed è costituito dal microcontrollore ATmega128 e dal CAN controller Infineon SAE81C91; quest'ultimo gestisce automaticamente, a livello hardware, il protocollo CAN bus. Il SAE81C91 è provvisto di registri che contengono fino a 16 diversi identificatori di messaggi CAN; all'arrivo di ogni messaggio, l'identificatore è confrontato con uno dei 16 memorizzati nei registri e, nel caso in cui l'identificatore è riconosciuto valido, il messaggio è trasmesso, altrimenti è cancellato. Il CAN controller, inoltre, è provvisto di un generatore di clock che fornisce una frequenza programmabile al microcontrollore ATmega128 tramite il pin CLKOUT.

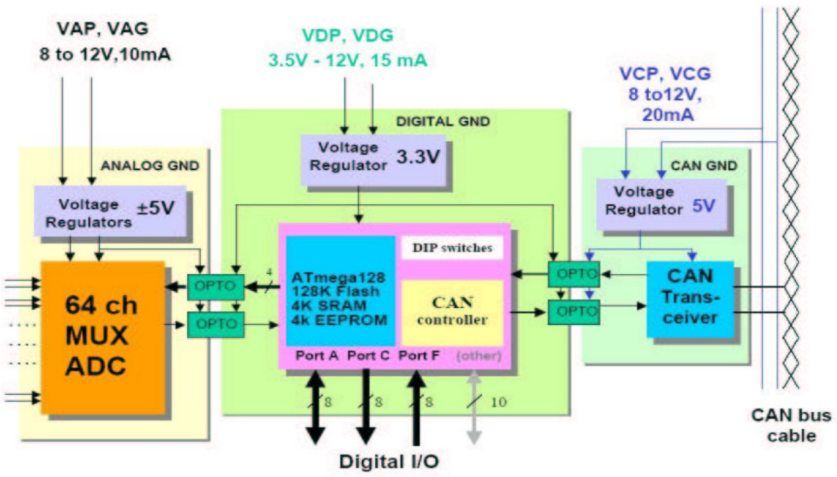

Figura 2.9: Diagramma a blocchi della ELMB 128

L'ATmega128 è collegato ad uno *switch* che permette all'utente di assegnare manualmente la velocità di trasmissione del CAN bus ed il nodo identificatore della

scheda (*Node-ID*). Tale numero deve essere univoco sul CAN bus e permette di indirizzare univocamente i comandi all'ELMB. Quindi il *Node-ID* altro non sarà che l'identificativo della Pad sulla quale la ELMB è montato. Nella figura sottostante si può notare come il *Node-ID* può variare fra 1 e 64, mentre le possibili velocità di trasmissione sono 50, 125, 250 o 500 kbit/s.

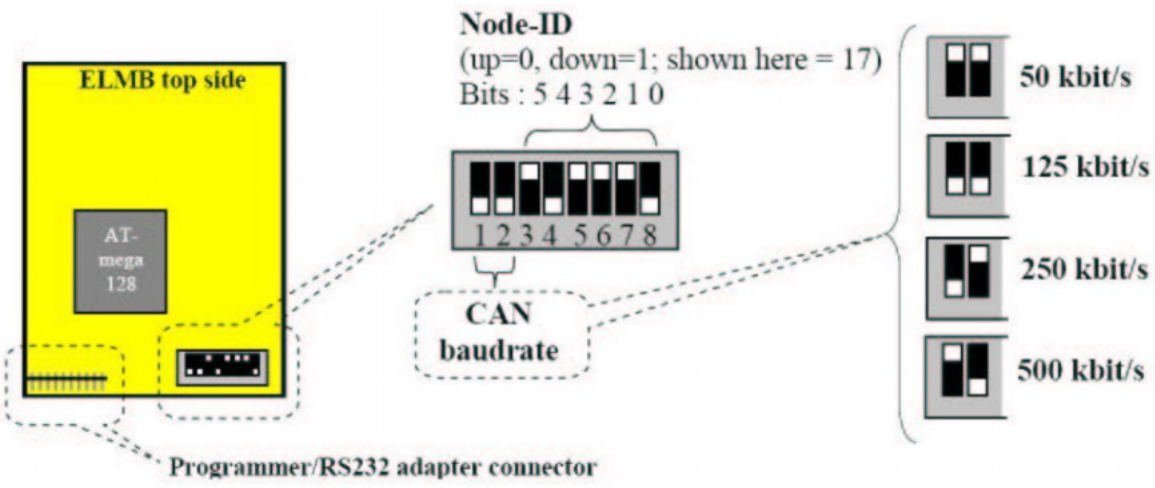

Figura 2.10: Disposizione e configurazione del DIP-switch

Il sistema di comunicazione interno è basato su quattro tipi di bus, implementati dai rispettivi protocolli:

- **Bus-I²C**: nella Pad sono presenti 4 canali I²C che connettono, in cascata, l'ELMB ai sensori di temperatura della Pad e degli Splitter, al TTC, ai PRODE, alla FPGA, alle CM, al link ottico.
- **Bus-SPI**: attraverso questo bus, il microcontrollore ATmega128 riceve i messaggi CAN.
- **Bus-JTAG**: nella Pad sono presenti 2 canali JTAG che collegano l'ATmega128 alla memoria EEPROM della FPGA.

Per concludere, la ELMB è fornita di un connettore ISP (*In-System-Programming*) per la programmazione da PC.

Capitolo 3

3 - L'ambiente di test e monitoring

Nel capitolo precedente abbiamo introdotto il sistema di trigger di primo livello evidenziandone anche tutte le criticità e le problematiche inerenti a condizioni quali la radioattività o il forte campo magnetico. Questo ci fa intuire l'importanza di dover progettare e sviluppare un ambiente di test che ci permetta di pianificare in anticipo quali interventi attuare in caso di malfunzionamento di uno o più componenti.
Testare l'elettronica da installare in caverna e quella disinstallata portata nella stazione di test è essenziale a ridurre i tempi di intervento. L'obiettivo primario è, quindi, tenere continuamente sotto controllo ogni eventuale malfunzionamento, in uno qualsiasi dei dispositivi lungo la catena di trigger, ed individuarlo il prima possibile.

L'obiettivo principale di questa tesi è stato quello di progettare e sviluppare un ambiente di test inerente ai livelli superiori di logica del trigger di primo livello, ossia le schede Pad: permettere ai tecnici di ATLAS di testare tutte le schede *Add-on-board*, da una postazione in superficie o nel sito dell'esperimento, è di cruciale importanza per poter pianificare gli interventi da attuare. Va infatti ricordato che gli accessi alla caverna di ATLAS non sono sempre permessi (in base all'accensione o meno di LHC ed alle condizioni ambientali) e, quindi, sapere esattamente quali modifiche apportare è di notevole importanza.
Il luogo scelto per la collocazione della stazione di test è un ufficio dell'INFN situato nel building BB5 del CERN, in località Meyrin. Questa è costituita da una parte hardware (server PC, Pad da testare, bus seriale di collegamento, ecc.) ed una software (tra cui l'applicativo da me implementato). Nei seguenti paragrafi verranno illustrati tali componenti nel dettaglio.

3.1 Hardware

Il cuore della stazione di test è il server *Dell XPS 8700* al quale vengono collegate le Pad (dotate di tutte le schede *Add-on-board* da testare) mediante bus seriale CAN (*Controller Area Network*).
Il server, dotato di processore *Intel Core i7-4790 @ 3.6 GHz* ed *8 GB di memoria DDR3L a doppio canale*, monta come sistema operativo la distribuzione Linux *Scientific Linux CERN release 6.7 (Carbon)* con installato l'ambiente di sviluppo per applicativi software *TDAQ* (*Trigger and Data Acquisition*) [16]. Questo è di fondamentale importanza dato che, mediante repository SVN (*SubVersioN*) e tool di gestione software come CMT (*Configuration Management Tool*) [17], ci permette di recuperare, compilare, linkare ed eseguire applicazioni utilizzate per l'esperimento ATLAS (tra cui il programma *testpad* che vedremo in seguito).

Come già accennato, le schede Pad (vedi Cap. 2) vengono connesse al server per effettuarne i test mediante bus seriale CAN (descritto a breve ed approfondito nell'App. A). Il PC è collegato tramite due USB 2.0 al modulo della Systec *USB-CANmodul16*, dotato di 16 canali CAN. Questa interfaccia USB CAN permette, in modo efficiente, di trasferire messaggi CAN in sistemi distribuiti che richiedono un nodo centralizzato che smisti i dati.
Il sistema supporta tutti i tipi di protocolli CAN di alto livello, uno su tutti *CANopen* (vedi Par. A.4); il bit rate va dai 10 kbps ad 1 Mbps.

Figura 3.1: Schema di collegamento tra le varie componenti hardware

Tutta la comunicazione tra il PC e la Pad da testare è controllata dalla ELMB (vedi Par. 2.3): essa usa il bus I^2C (*Inter Integrated Circuit*, vedi App. B) per comunicare con tutti i registri delle schede *Add-on-board* della Pad. L'ELMB è interfacciata con il PC, come appena detto, tramite il bus seriale CAN utilizzando il protocollo CANopen. Tramite questo protocollo si possono trasmettere dal PC all'ELMB i dati e le informazioni necessarie per comunicare con un singolo componente.
Ogni Pad, con la sua ELMB, è identificata da un valore di nodo univoco che la contraddistingue da tutte le altre. L'ELMB controlla 4 canali I^2C per la lettura e scrittura dei registri interni dei diversi dispositivi della Pad.

3.1.1 Accenni al bus seriale CAN

Il *Controller Area Network*, noto anche come *CAN-bus*, è uno standard seriale per bus, principalmente usato in ambiente automotive, introdotto negli anni ottanta dalla Robert Bosch GmbH per collegare diverse unità di controllo elettronico. Tra le caratteristiche principali del protocollo CAN annoveriamo:

- la comunicazione fino ad 1 Mbit/sec;
- la capacità di lavorare in condizioni ambientali ostili;
- l'auto-diagnostica e la facilità di configurazione ed utilizzo.

Sebbene inizialmente applicata, come già detto, in ambito automotive, attualmente è usata in molte applicazioni industriali di tipo embedded, dove è richiesto un alto livello di immunità ai disturbi. Il bitrate può raggiungere 1 Mbit/s per reti lunghe meno di 40 m. Bitrate inferiori consentono di raggiungere distanze maggiori (ad es. 125 kbit/s per 500 m).

Il protocollo CAN si colloca all'interno della pila ISO/OSI nel *Physical Layer* e *Data Link Layer*, lasciando totale libertà per quanto riguarda l'*Application Layer*.
Per quanto riguarda il livello fisico, il cavo trasmissivo del CAN è un doppino intrecciato, schermato o meno a seconda delle applicazioni. Il Physical Layer ha

anche il compito di garantire la rappresentazione dello stato *dominante* (0 logico = 0 V) e dello stato *recessivo* (1 logico = 5 V).

Il Data Link Layer è il cuore del protocollo CAN dato che definisce la politica di accesso al mezzo di trasmissione. È caratterizzato dall'assenza degli indirizzi per i dispositivi utilizzando la tecnica del *Multicast*, la quale permette un alto grado di modularità e di flessibilità del sistema.
Altro compito importante del Data Link Layer è quello di definire il formato dei messaggi CAN, suddividendoli in quattro tipi (*Data Frame*, *Remote Frame*, *Error Frame*, *Overload Frame*).
Sempre su questo livello sono definite le procedure di rivelazione degli errori e le relative correzioni. Un errore può essere rivelato a livello di messaggio o a livello di singolo bit e, grazie ad un meccanismo di auto-isolamento dei guasti, il CAN provvede ad isolare il dispositivo danneggiato.

Per quanto riguarda l'*Application Layer*, lo standard lascia molta più libertà all'utilizzatore. Ogni protocollo di alto livello implementa queste funzionalità in modo diverso, adattandole alle specifiche esigenze del campo applicativo per cui è stato sviluppato. *CANopen*, utilizzato nell'esperimento, nasce per applicazioni industriali standardizzate ed, a differenza degli altri protocolli, definisce in dettaglio il frame che transita sul bus. Un messaggio CANopen è costituito da tre campi principali:

- **COB-ID** (*CAN Open Block Identity*): formato da altri tre sotto-campi, indica la tipologia di dato che transita sul bus (*Function Code*), a chi è destinato (*Node ID*) ed il bit di controllo che identifica il dato vero e proprio (*RTR, Remote TRanser*);
- **Data Lenght**: identifica la lunghezza effettiva del dato da trasmettere sul bus;
- **Data**: il dato vero e proprio, di lunghezza variabile da 0 ad 8 byte.

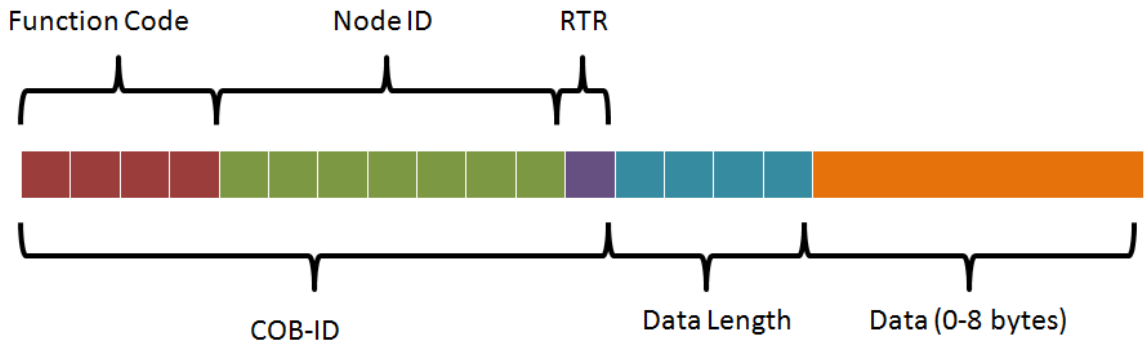

Figura 3.2: Struttura di un frame CANopen

Una caratteristica fondamentale di CANopen è quella di sfruttare il concetto dei *profile* per garantire ai sistemisti la possibilità di far dialogare dispositivi realizzati da differenti produttori.

Per maggiori informazioni inerenti il CANbus ed il protocollo CANopen, si rimanda il lettore alla lettura dell'App. A.

3.2 Il software di inizializzazione e controllo: testpad

L'inizializzazione ed il controllo della Pad da PC remoto è reso possibile tramite il modulo Systec USB-CANmodul16 ed il programma *testpad*. Il primo fornisce, come scritto nel paragrafo precedente, un'interfaccia hardware tra PC, che opera in ambiente Linux, e CANbus; il secondo invece è un'interfaccia software, sviluppata in linguaggio C++, che implementa il protocollo CANopen.
Durante il periodo di tesi ho utilizzato *testpad* (essendo il programma già utilizzato in ATLAS per la verifica del corretto funzionamento delle schede Pad) modificandolo in alcune routine non perfettamente funzionanti ed introducendone altre totalmente innovative. Di questo se ne discuterà più in avanti.

L'architettura di *testpad* è mostrata in Fig. 3.3 sotto forma di diagramma UML (*Unified Method Language*).

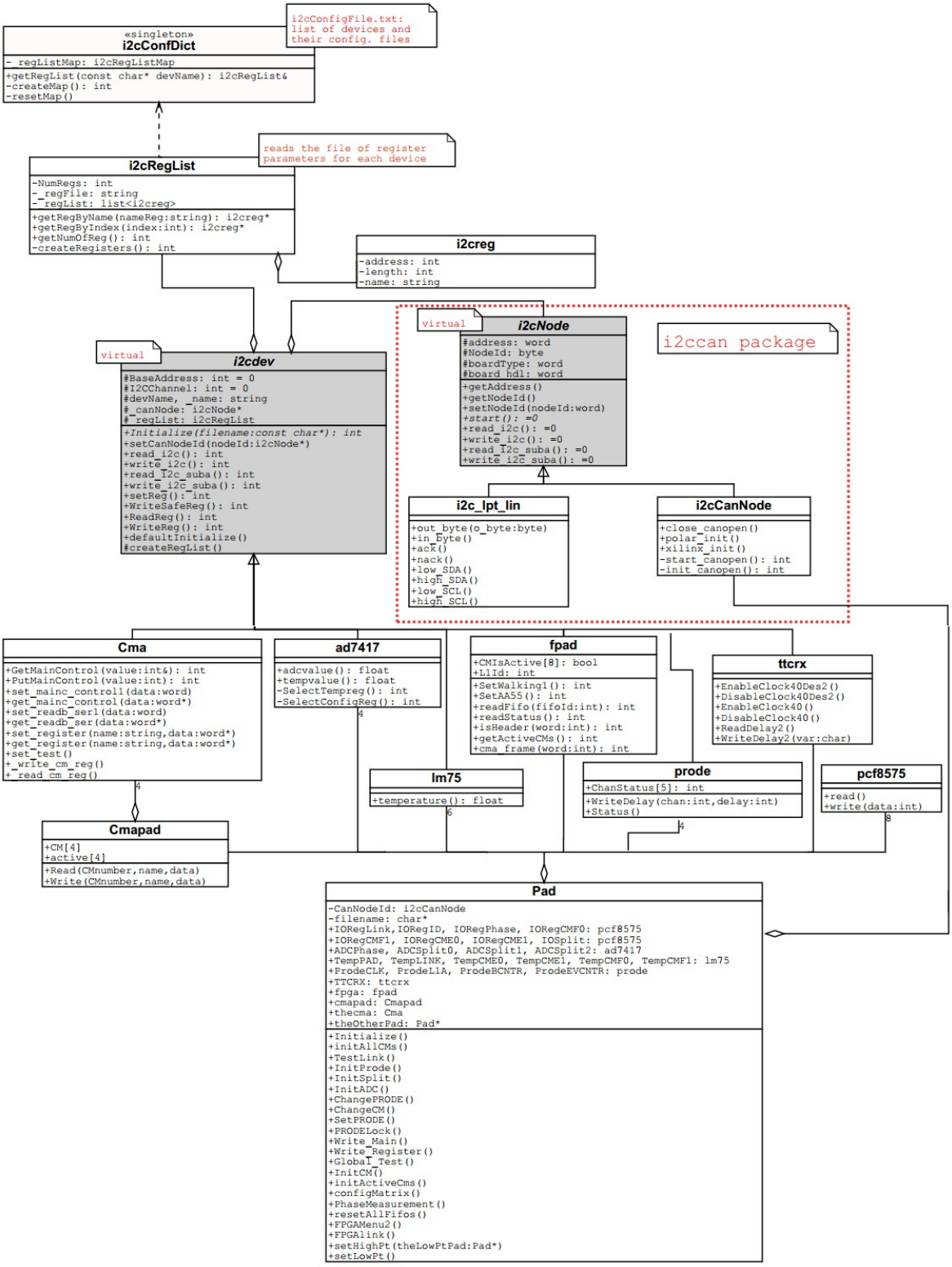

Figura 3.3: Diagramma UML del sistema di configurazione e controllo di testpad

Il codice è contenuto in due pacchetti chiamati *"RPCi2ccan"* e *"RPCgen_i2cprg"*. Il primo contiene l'interfaccia per la comunicazione con i nodi CANbus mentre il secondo contiene il codice per le componenti I^2C montate sulla Pad. A ciascuna delle componenti montate sulla Pad è associata una classe (la classe CMA, la classe AD7417, ecc.) che eredita dalla classe astratta *"i2cdev"*. Quest'ultima utilizza la classe *"i2cCanNode"* per la connessione CAN e per implementare il protocollo I^2C specializzandolo attraverso il protocollo CANOpen che, a sua volta, permette la comunicazione con l'ELMB. Se invece si vuole comunicare via I^2C attraverso la porta parallela del PC, allora si può utilizzare la classe *"i2c_lpt_lin"*. La classe *Pad* invece gestisce la comunicazione tra i vari componenti e la loro inizializzazione.
Per distinguere il comportamento di ogni oggetto della Pad, la classe *"i2cdev"* contiene alcune descrizioni del componente come: il tipo (*devName*), il nome (*_name*) e la lista dei registri (gestita dalla classe *"i2cRegList"*).
La classe *"i2cConfdict"* è una classe *singleton* (un design pattern creazionale che ha lo scopo di garantire che, di una determinata classe, venga creata una ed una sola istanza) che ha il ruolo di configuratore. Il configuratore ha il compito di leggere il file *"i2cConfigFile.txt"*, in cui sono scritte tutte le informazioni di configurazione dei diversi dispositivi e la relativa procedura di inizializzazione, per poi creare un dizionario che associa ad ogni componente il suo file di configurazione. In questo modo, scrivendo un semplice file di testo, si possono riconfigurare durante l'esecuzione (comunemente detto a "*runtime*") tutti i dispositivi della Pad, senza dover ogni volta compilare il programma.
Il programma *testpad* è quindi un'interfaccia software alle librerie del sistema di controllo ed inizializzazione appena descritte, che tramite menù testuali permette all'utente di eseguire tutte le operazioni che il sistema offre. Le scelte principali del menù offerto da *testpad* sono:

1. **Initialize PAD**: effettua l'inizializzazione della Pad tramite la funzione *"Initialize()"*. Questa funzione è implementa dalla classe *"Pad"* che

provvede ad effettuare la connessione con il CAN ed ad assegnare il numero di nodo (*nodeID*).

2. **PRODE MENU**: permette di impostare i possibili ritardi sulle 4 linee dei 4 chip PRODE ed offre la possibilità di accenderli o spegnerli.
3. **TTC MENU**: permette di inizializzare il TTCrx, di accenderlo e di spegnerlo.
4. **CM MENU**: permette di inizializzare una singola CM, effettuare il reset ed accedere ai registri di configurazione delle 4 CM.
5. **FPGA MENU**: permette di impostare la logica per i diversi test, di trasmettere dei pacchetti dati attraverso il link ottico per testarne il corretto funzionamento e di azzerare e leggere le FIFO.
6. **Link MENU**: inizializza il link ottico ed attiva/disattiva il laser.
7. **CAN MENU**: apre/chiude il canale 22 del CAN.
8. **ELMB MENU**: reset, accensione e spegnimento dell'ELMB, lettura della versione hardware/firmware ed attivazione o disattivazione del debug dei messaggi CAN.
9. **SPLITTER MENU**: consente di accendere o spegnere gli 8 diversi SPLITTER posti sulla scheda connessa tramite I^2C alla scheda Pad e di effettuarne misure di tensione.
10. **Change current PAD**: permette di cambiare la Pad con cui si vuole comunicare.
11. **Reset full PAD**: comando di reset di tutte le FIFO delle CMA e della FPGA.
12. **Warm Initialize PAD**: inizializza TTC, PRODE, FPGA e SPLITTER.
13. **Change PAD configuration**: permette di cambiare le configurazioni di Trigger e quelle delle CM.
14. **Change CM latencies**: cambia le latenze delle CM.
15. **Phase measurement**: legge il valore di fase per la calibrazione temporale interna della Pad.
16. **Power ON/OFF**: accende e spegne la Pad.

17. **Print PAD Status**: stampa lo stato dei dispositivi della Pad.
18. **Measurement loop**: legge tutti i sensori di temperatura sulla Pad e sugli SPLITTER, le tensioni dei singoli dispositivi ed i bit che segnalano il superamento dei valori soglia.
19. **Read CM trigger frequencies**: legge le frequenze di Trigger delle CM.
20. **Fast check of locks**: verifica se i PRODE, le CM, il TTCRX ed il GLinkTX sono agganciati al clock.
21. **TRIGGER MENU**: permette di settare l'OR e l'AND logico, abilitare il Trigger sulle CM, settare il *dead-time* di Trigger e stamparne lo stato.
22. **Read CM BC ids**: legge i BC ID delle CM.
23. **Test CM BC ids with prode**: testa i BC ID delle CM con i PRODE.
24. **Test CM BC ids with TTC**: testa i BC ID delle CM con il TTC.
25. **Test init low-high**: testa l'inizializzazione low-high.

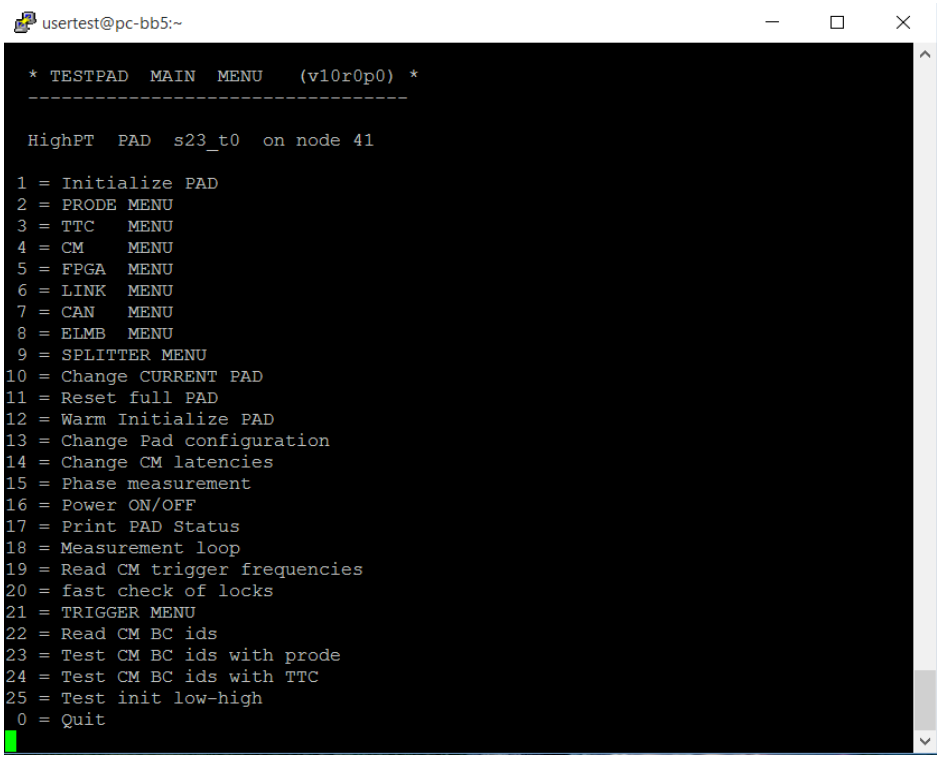

Figura 3.4: Menù principale di testpad

3.3 Problemi riscontrati e possibili soluzioni

La funzionalità principale di *testpad* è quella di fungere da driver per le schede Pad e consentirne il corretto interfacciamento con il PC che esegue il test. In questo modo, il software è entrato nel repository ufficiale di ATLAS ed è l'applicativo utilizzato dagli esperti RPC, nell'ambito dell'esperimento, nel corso degli ultimi dieci anni.
Purtroppo, negli ultimi tempi, le esigenze sono cambiate e sono venute alla luce una serie di bug che ne compromettono relativamente l'affidabilità e le prestazioni. Ciò nonostante, non si può per adesso pensare di implementare un software ex novo dato che, al momento della stesura di questa tesi, è in corso una nuova presa dati da parte dell'esperimento ATLAS ed il software deve essere perennemente disponibile per gli RPC.

Per il lavoro di questa tesi si è pensato di risolvere questi problemi andando ad implementare un nuovo "strato" software che interagisca con l'attuale versione di *testpad* (per quanto riguarda l'interazione con la scheda Pad mediante CANbus), superandone i problemi noti ed implementando diverse feature, riguardanti la fase di verifica di tutte le schede *Add-on-board*. Uno dei risultati più eclatanti, riguardante la parte di test, consiste nel perfezionamento della procedura di verifica della comunicazione, migliorandone le prestazioni di tempo di un fattore 8; di questo se ne parlerà nel prossimo capitolo.
Si è quindi pensato di implementare un'interfaccia grafica che renda anche più intuitivo e veloce l'utilizzo del software; il linguaggio di programmazione scelto è stato *Java SE 7 (1.7.0)* per le caratteristiche che offre:

1. è indipendente dalla piattaforma di esecuzione;
2. è orientato agli oggetti;
3. è progettato per eseguire codice da sorgenti remote in modo sicuro.

I punti 1 e 3 sono i principali per il contesto in cui andrà calato il software implementato: al CERN non tutte le macchine hanno lo stesso sistema operativo (o, quanto meno, non alla stessa versione) e la sicurezza da remoto è un aspetto indispensabile da garantire, considerando gli innumerevoli collegamenti che ogni giorno vengono effettuati dagli scienziati degli stati membri. Quindi il codice compilato che viene eseguito su una piattaforma non deve essere ricompilato per essere eseguito su una piattaforma diversa. Il prodotto della compilazione è infatti in un formato chiamato *bytecode* che può essere eseguito da una qualunque implementazione di un processore virtuale detto *Java Virtual Machine*.

Il software può essere diviso in due macro-parti: la prima inerente alla connessione con la scheda Pad, mediante collegamento a *testpad*, e la seconda di interazione con la stessa. Per quanto riguarda la fase di connessione, va fatta una premessa sul funzionamento di *testpad*. L'applicativo viene lanciato dal terminale di Linux con la seguente sintassi:

testpad -i -cCANALE_CAN -nID_PAD FILE_CONFIGURAZIONE

dove con il parametro "*-c*" si indica a quale canale CAN, del modulo USB della Systec, è connessa la Pad con identificativo riportato dal parametro "*-n*"; l'attributo "*-i*" indica, semplicemente, la modalità "interattiva" di esecuzione del software (ossia quella tramite menù testuale, ne esistono altre ma non rilevanti ai fini di questa tesi), mentre nel campo "*FILE_CONFIGURAZIONE*" va, appunto, riportato il file utile al test di funzionamento della scheda. Un esempio di esecuzione può, quindi, essere il seguente:

testpad -i -c0 -n42 test.txt

dove, in questo caso, ci stiamo connettendo alla Pad con identificativo *42*, tramite il canale *0*, caricando il file di configurazione *test.txt*.

Questa premessa ci è utile a capire quali parametri sono fondamentali al fine del collegamento con la scheda Pad e, quindi, quali valori vanno presi in considerazione per l'implementazione della fase di connessione all'hardware. Nella Fig. 3.5 è riportata l'interfaccia grafica all'avvio del software da me sviluppato (per i dettagli implementativi si rimanda la lettura al prossimo capitolo).

Figura 3.5: Schermata di connessione della versione in Java di testpad

Come è possibile notare, con i due menù a tendina è possibile selezionare il canale CAN ed il nodo Pad, mentre con il pulsante "*Configuration File*" è possibile scegliere il file da sottomettere durante il test. Particolare attenzione va posta alla check box "*All pads on can channel*" che, se selezionata, identifica automaticamente le Pad connesse in cascata al canale CAN selezionato. Una volta selezionato il tutto, è possibile lanciare l'applicativo cliccando sul pulsante "*Connect*" che, dopo una verifica di correttezza dei dati inseriti, ci mostrerà la schermata principale del programma (fig. 3.6). Le varie funzionalità di *testpad* sono

adesso organizzate in schede, ognuna delle quali corrispondente ad una voce del menù testuale della versione originale del software. Per motivi di spazio è riportata, a titolo di esempio, solo la schermata di inizializzazione.

Le opzioni sono così partizionate:

- **Initialize**: contiene i tre tipi di inizializzazione delle schede *Add-on-board* (dal file di configurazione, dalla memoria EEPROM, personalizzata) ed il selettore per le schede Pad (se al momento della connessione è stata selezionata la check box "*All pads on can channel*").
- **PRODE**: inizializza, setta i ritardi sui canali del chip e visualizza lo stato del PRODE selezionato dal relativo menù a tendina.
- **TTC**: inizializza e visualizza lo stato del clock.
- **CM**: inizializza (singolarmente o tutte insieme) le CM e ne visualizza lo stato. È anche presente l'opzione per poter scrivere o leggere il contenuto di uno dei 158 registri della CM (con il relativo controllo della lunghezza).
- **FPGA**: inizializza il circuito integrato dal file di configurazione, ne visualizza lo stato e manipola il contenuto dei 32 registri della FPGA.
- **Link**: inizializza e visualizza lo stato del link ottico GLinkTX.
- **ELMB**: visualizza la versione del firmware dell'ELMB.
- **Splitter**: inizializza e visualizza lo stato dello Splitter.
- **Test**: effettua il test di funzionamento di tutte le schede *Add-on-board* delle Pad selezionate. In più, sono presenti le opzioni per visualizzare informazioni quali lo stato generale della Pad e misure di tensione elettrica e di temperatura.

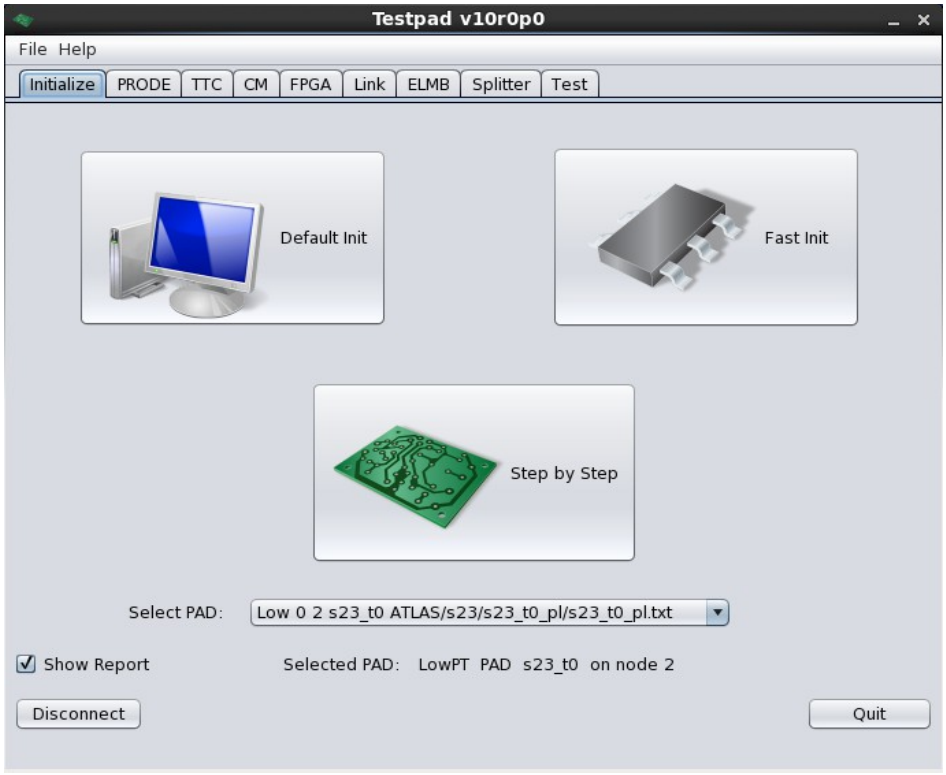

Figura 3.6: Schermata principale della versione in Java di testpad

Se è stata selezionata la check box "Show Report", per ogni azione elencata precedentemente verrà visualizzato un report testuale che permetterà di capire le funzionalità dell'hardware testato ed, eventualmente, quali interventi attuare al fine della riparazione.

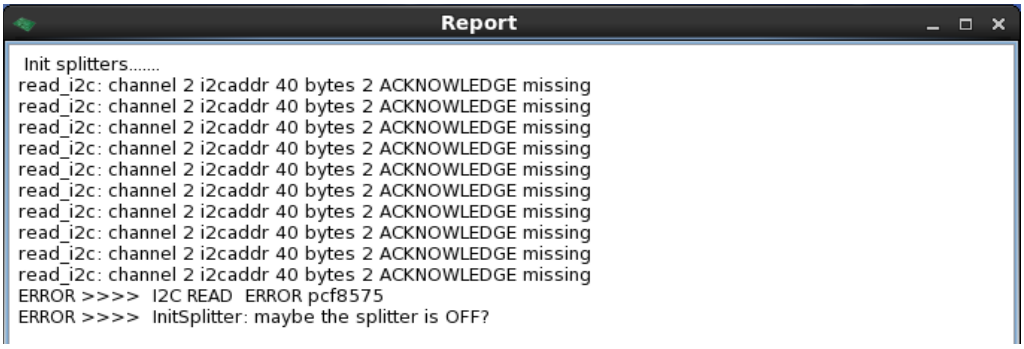

Figura 3.7: Esempio di report inerente ad uno Splitter spento o non funzionante

Per i dettagli sull'implementazione e sulla fase di test si rimanda il lettore al prossimo capitolo.

Capitolo 4

4 – Implementazione e test

In questo capitolo mostreremo i dettagli implementativi relativi alla versione grafica da me implementa di *testpad*. Come già accennato nel capitolo precedente, il linguaggio di programmazione scelto è Java; più nel dettaglio, è stato utilizzato il framework *Swing* (appartenente alle JFC, *Java Foundation Classes*), ossia la libreria ufficiale per la realizzazione di interfacce grafiche in Java. Tale scelta porta notevoli vantaggi:

- **indipendenza dalla piattaforma**: Swing è indipendente dalla piattaforma di esecuzione sia per il linguaggio (Java appunto) sia per la sua implementazione (i *widget*, ossia gli elementi grafici come ad esempio i pulsanti, vengono renderizzati in modo universale);
- **estensibilità**: tutte le diverse parti della libreria possono essere "collegate" tra di loro per crearne versioni estese delle stesse;
- **orientata ai componenti**: il framework è basato su componenti, ossia oggetti con determinate caratteristiche di comportamento conosciute e specificate. Ogni componente emette eventi e risponde ad un preciso set di comandi;
- **configurabilità**: Swing ha la possibilità di modificare a runtime i propri settaggi. Ad esempio, un'applicazione basata su tale framework, può cambiare il suo *Look and Feel* (ossia le caratteristiche percepite da un utente di un'interfaccia grafica, sia in termini di apparenza visiva (il *Look*) che di modalità di interazione (il *Feel*)) senza che l'utente finale vada a modificare il sorgente.

La scelta del framework è intuibile anche osservando le classi, nel formato UML, in Fig. 4.1. Swing definisce una gerarchia di classi che forniscono ogni tipo di componente grafico (finestre, pannelli, frame, bottoni, aree di testo, ecc.).

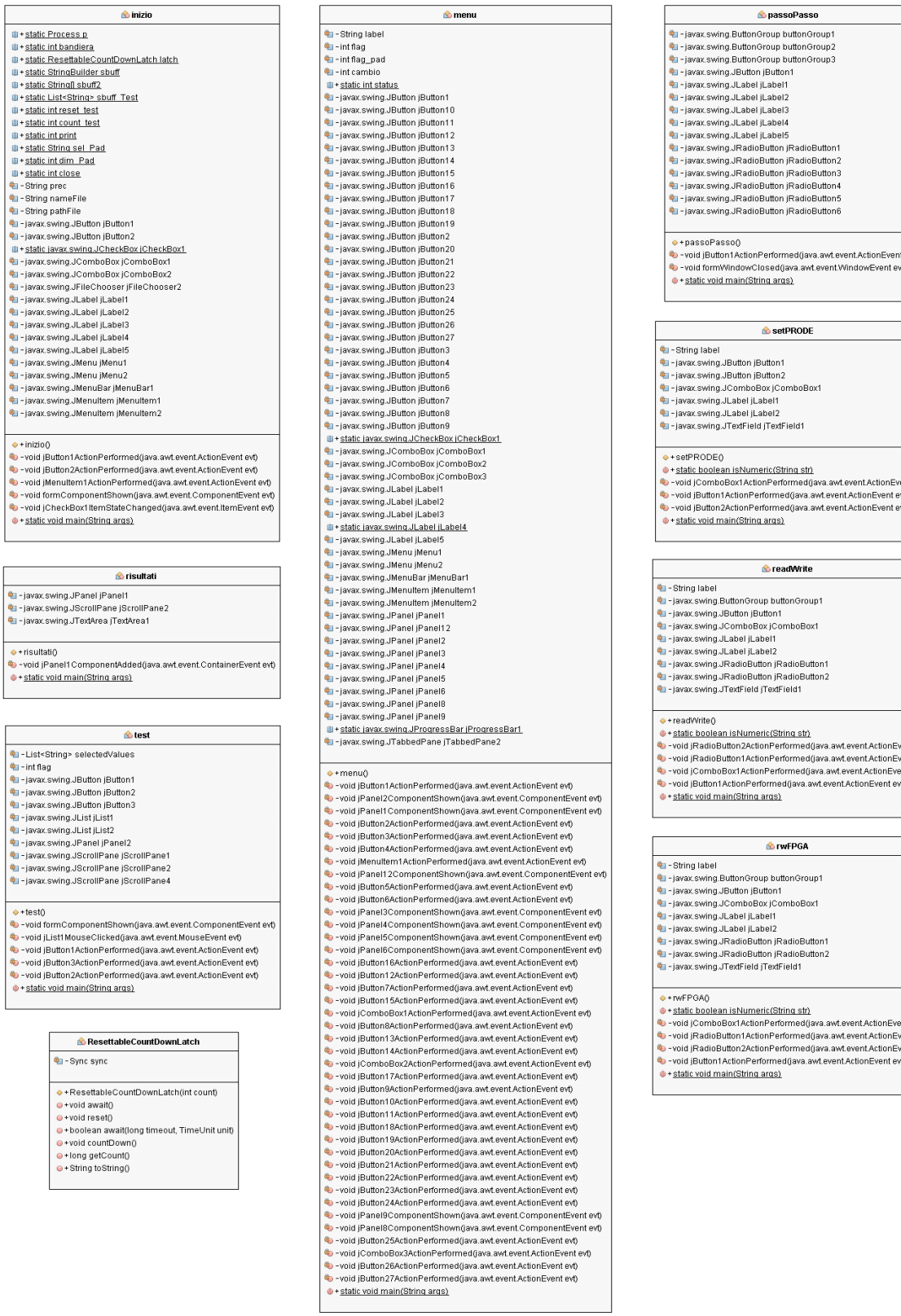

Figura 4.1: Classi UML della versione Java di testpad

Come è possibile notare, tutti gli attributi che hanno per tipo "*javax.swing*" rappresentano una determinata componente dell'interfaccia grafica. Ad esempio, con "*javax.swing.JButton*" si rappresenterà un pulsante, con "*javax.swing.JLabel*" un'etichetta testuale e così via.
Concludiamo il discorso sulla libreria Swing spendendo qualche parola sull'interazione tra essa e l'utente. Questa avviene tramite il paradigma "*event-driven*", ossia gli algoritmi reagiscono agli eventi che l'utente, in modo interattivo, genera sui componenti grafici. Abbiamo quindi a che fare con due nuovi concetti: quello, appunto, di *evento* e quello di *ascoltatore dell'evento*. Facciamo un esempio analizzando il metodo

void jButton1ActionPerformed(java.awt.event.ActionEvent evt)

presente nella classe "*inizio*": l'utente genera un *evento* (la variabile *evt*, altro non è che il click del mouse sul pulsante) che verrà catturato dal componente grafico *JButton1* (mediante il metodo precedente), andando a ricoprire il ruolo dell'*ascoltatore dell'evento*. Tutte le interazioni tra l'utente e l'applicativo seguono questa logica.

Come già accennato nel capitolo precedente, il software da me implementato può essere diviso in due macro-parti: una parte inerente la connessione con la scheda Pad ed un'altra inerente l'interazione con essa. Nei prossimi paragrafi analizzeremo questi aspetti nel dettaglio.

4.1 Connessione alla scheda Pad

La parte del software inerente la connessione con la scheda Pad segue anch'essa il criterio dell'*event-driven* ed è gestita dalla classe *"inizio"* (vedi Fig. 4.1). L'utente seleziona il canale CAN, il nodo Pad ed il file di configurazione e "lancia" l'evento cliccando sul pulsante "*Connect*" (vedi Fig. 3.4).

All'avvio il programma controlla l'esistenza o meno dell'ultima configurazione utilizzata dall'utente.

Questa viene memorizzata in un file di inizializzazione (*init*) nella home directory del software e gestita dal metodo

private void formComponentShown(java.awt.event.ComponentEvent evt)

L'evento che invoca la procedura è l'avvio del componente *JFrame* della libreria Swing. Il metodo controlla dapprima l'esistenza del file *init* e, se presente, ne legge il contenuto andando a settare le relative componenti e variabili.

```
1  try {
2      can = br.readLine();
3      pad = br.readLine();
4      pathFile = br.readLine();
5      nameFile = br.readLine();
6      while((sCurrentLine = br.readLine())!=null){
7          sbuff2[i]=sCurrentLine;
8          i++;
9      }
10     dim_Pad=i;
11
12 }
13 ...
14 //Settaggi
15 jComboBox1.setSelectedItem(can);
16 jComboBox2.setSelectedItem(pad);
17 jLabel4.setText(nameFile);
```

Figura 4.2: Estratto del metodo formComponentShown

Dal listato in Fig. 4.2 è possibile intuire come prima si vadano a leggere i valori inerenti alla porta CAN, al nodo Pad ed al file di configurazione; con questi parametri si andranno a settare i componenti Swing presenti alle linee 15, 16 e 17. Con il ciclo *while*, invece, andiamo a riempire un array di stringhe contenente tutte le informazioni inerenti le schede Pad presenti nel file di configurazione; ogni elemento di tale struttura dati avrà la seguente forma:

Low 0 2 s23_t0 ATLAS/s23/s23_t0_pl/s23_t0_pl.txt

Particolare attenzione va posta al primo parametro che indica il tipo di scheda (*Low* o *High*) e l'ultimo che indica il file di configurazione specifico per quella Pad.
In caso di ultima configurazione non presente o di primo avvio del programma o, semplicemente, se si desidera cambiare file di configurazione da caricare sulla scheda Pad, bisogna cliccare sul pulsante "*Configuration File*" (vedi Fig. 3.4). Dal click scaturisce l'evento che viene catturato dal metodo

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt)

Questo gestisce l'apertura del modulo "*jFileChooser2*" di Swing che permette la selezione del file dal PC e di leggerne il contenuto; anche in questo caso si andrà a riempire un array di stringhe inerente alle caratteristiche delle Pad.

Il cuore della classe "*inizio*" è il metodo

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)

invocato dal click sul pulsante "*Connect*". Recuperati i valori selezionati da entrambe le *jComboBox* (la porta CAN ed il nodo Pad), questi vengono utilizzati per eseguire il processo *testpad* da shell:

p = Runtime.getRuntime().exec("testpad -i -c"+can+" -n"+pad+" "+pathFile);

(in caso di selezione della check box "*All pads on can channel*", il parametro *-n* viene sostituito con *-all*).

```
1  final InputStream inStream = p.getInputStream();
2  //Thread testpad
3  Thread uiThread = new Thread("UIHandler") {
4      @Override
5      public void run() {
6          InputStreamReader reader = new InputStreamReader(inStream);
7          Scanner scan = new Scanner(reader);
8          ...
9          //Leggi ogni linea di testpad
10         while (scan.hasNextLine()) {
11             ...
```

Figura 4.3: Estratto del metodo jButton1ActionPerformed inerente alla gestione del thread

Si è pensato ad una logica multithreading per gestire l'I/O tra l'applicativo in Java ed il processo *testpad*.

La linea 1, nel listato in Fig. 4.3, identifica il metodo che ottiene il flusso di input da un sottoprocesso. Esso ottiene i dati convogliati dallo standard output del processo rappresentato dall'oggetto *p*.

Con la linea 3 si crea un nuovo *thread*: il processo principale genera un nuovo sottoprocesso ed entrambi vengono eseguiti concorrentemente da un sistema di elaborazione monoprocessore (*multithreading*) o multiprocessore (*multicore*). Così facendo, il thread si occuperà soltanto dell'esecuzione e della presa dati dal programma *testpad*, mentre il processo iniziale continuerà a gestire la parte grafica del programma.

Tutto il codice eseguito dal thread si trova all'interno del metodo *run()* (linea 5). Prima di tutto si decodifica il flusso di byte, del programma *testpad*, in caratteri (linea 6); da qui si istanzia un oggetto *scan* che rappresenterà tutto l'output di *testpad*, delimitato questa volta da spazi per una più semplice gestione (linea 7).

```
1  //Cattura output
2  prec=scan.nextLine();
3  System.out.println(prec);
4  sbuff.append(prec);
5  sbuff.append('\n');
6  ...
7  //Rileva PAD
8  if(prec.contains("LowPT") || prec.contains("HighPT")){
9      sel_Pad=prec;
10     if(close==1){
11         menu.jLabel4.setText(sel_Pad);
12     }
13 }
14 ...
15 //Errore collegamento CAN
16 if(prec.contains(err2)){
17     Exception e = new Exception("Error!");
18     Component f = null;
19     JOptionPane.showMessageDialog(f, "WARNING: there are pads not connected. Exit!",
20         e.getMessage(), JOptionPane.WARNING_MESSAGE);
21     System.exit(1);
22 }
```

Figura 4.4: Cattura output, rilevazione Pad ed errore CAN

La linea 10 mostra come l'output viene processato: finché *testpad* continua ad inviare nuovi dati, il thread resta in ascolto per elaborare le nuove informazioni ricevute.

Dapprima vengono settate le variabili di inizializzazione del processo in atto (reset del risultato, avvio della barra di caricamento), poi viene memorizzato l'output di *testpad* in una struttura dati di tipo *StringBuilder* (*sbuff* in Fig. 4.4). In base al contenuto di quest'ultima si valuta il risultato atteso. Sempre in Fig. 4.4 vengono riportati due esempi di valutazione dell'output: la linea 8 controlla se *testpad* sta analizzando una Pad low od una high, mentre la linea 16 cattura un eventuale errore di collegamento CAN (quando il programma da terminale manda il messaggio "*Check the CAN cable status!*"). In quest'ultimo caso viene generata un'eccezione (linea 17), mostrata a video tramite un *JoptionPane* di Swing (linea 19).

```
1  if(prec.contains("Quit") || prec.contains("Exit")){
2      if(close==1){
3          menu.jProgressBar1.setValue(menu.jProgressBar1.getMaximum());
4          //Stampa risultati se richiesti
5          if(menu.jCheckBox1.isSelected() && print==1 ){
6              new risultati().setVisible(true);
7          }
8          menu.jProgressBar1.setVisible(false);
9      }
10     ...
11     latch.countDown();
12     start=0;
13 }
14 ...
15 };
16 uiThread.start();
```

Figura 4.5: Condizione di uscita, stampa dei risultati, sincronizzazione ed avvio del thread

La condizione di uscita è mostrata nel sorgente in Fig. 4.5. Catturato uno dei messaggi alla linea 1, si verifica se è stata richiesta la stampa a video dei risultati (linea 5); se si, viene istanziata una nuova finestra di output (Fig. 3.7).

Particolare attenzione va posta alla linea 11: *latch* è un'istanza della classe "*ResettableCountDownLatch*", la quale si differenzia dalla classe ufficiale di Java

SE 7 per la presenza del metodo di reset del valore di "conteggio". Questa istruzione funge da barriera di sincronizzazione tra il thread *uiThread* ed il processo che lo ha generato (il *Main Thread*): appena il thread raggiunge questa riga di codice, il processo può elaborare l'output.

Concluso il metodo *run()*, si lancia il thread con il metodo *start()* (linea 16) che, appunto, eseguirà il codice scritto nel suddetto metodo.

```
1  //Memorizzazione ultimo settaggio
2  PrintWriter writer = new PrintWriter("init", "UTF-8");
3  writer.println(can);
4  writer.println(pad);
5  writer.println(pathFile);
6  writer.println(nameFile);
7  for(String linea_pad : sbuff2 ){ //PAD collegate
8      if(linea_pad!=null){
9          writer.println(linea_pad);
10     }
11     else{
12         break;
13     }
14 }
15 writer.close();
16
17 this.dispose();
18 new menu().setVisible(true);
```

Figura 4.6: Memorizzazione ultima configurazione ed apertura del menù

Il metodo *jButton1ActionPerformed(...)* si conclude con l'estratto di codice in Fig. 4.6. Con le linee dalla 2 alla 15 si memorizza l'ultima configurazione di connessione utilizzata salvando, nel file di testo "*init*", la porta CAN, i nodi Pad, il path ed il nome del file di configurazione.

Fatto ciò, si passa dalla schermata di connessione a quella del menù principale (Fig. 3.6), quest'ultima analizzata nel seguente paragrafo.

4.2 Gestione della scheda Pad

La fase successiva a quella di connessione con la scheda Pad è la gestione di quest'ultima. Come già accennato nel Cap. 3, le varie opzioni sono ripartizionate in "schede" (mediante il componente *jTabbedPane* di Swing) in base alla tipologia delle opzioni stesse. Ogniqualvolta si passa da una scheda all'altra, si ha un'interazione con il programma *testpad* da shell. Ad esempio, nel listato in Fig. 4.7 è mostrato l'evento di selezione del menù "*PRODE*" (*Component Shown* del pannello *jPanel* che contiene le varie opzioni di gestione dei chip PRODE).

```
1  private void jPanel2ComponentShown(java.awt.event.ComponentEvent evt) {
2      ...
3      OutputStream outStream = inizio.p.getOutputStream();
4      PrintWriter pWriter = new PrintWriter(outStream);
5      pWriter.println("0");
6      pWriter.println("2");
7      pWriter.flush();
8
9      jComboBox2.setSelectedIndex(0);
10     ...
11 }
```

Figura 4.7: Metodo di selezione del menù PRODE

La linea 3 prende il flusso di output del programma Java e lo convoglia in quello di input del processo *testpad*, rappresentato dalla variabile pubblica *p* della classe "*inizio*". Con la linea 4 si formatta l'oggetto *outStream* in un flusso di dati di tipo testuale. Così facendo, il programma Java seleziona il relativo menù del processo *testpad* (nel caso specifico, invia prima "0" per uscire dal menù precedente, poi "2" per entrare nel menù dei chip PRODE). Infine, con il metodo *flush()* alla linea 7, si svuota il flusso di dati da mandare in input al processo *p*.

Il codice rappresentato in Fig. 4.7 è, in linea di massima, quello che accade per ogni menù selezionato.

Nella scheda "*Initialize*" è collocato il selettore di Pad connesse al canale CAN scelto in fase di collegamento; questo è rappresentato da un menù a tendina (*jComboBox* di Swing) il quale viene riempito, al primo avvio, nel seguente modo:

```
1  if(inizio.jCheckBox1.isSelected()){
2      for(String linea_pad : inizio.sbuff2){
3          if(linea_pad!=null){
4              linea_pad=linea_pad.replaceAll("\\s+", " ");
5              jComboBox3.addItem(linea_pad);
6          }
7          ...
8      }
9      String stringa;
10     stringa=(String)jComboBox3.getSelectedItem();
11     label=stringa.split(" ");
12     pWriter.println("10");
13     pWriter.println(label[2]);
14     ...
15 }
```

Figura 4.8: Sorgente inerente al riempimento del menù a tendina delle schede Pad

La linea 1 verifica se la check box "*All pads on can channel*" è selezionata; successivamente si cicla sulla struttura dati *sbuff2* (riempita in precedenza con i dati delle Pad connesse, vedi Par. 4.1). Con la linea 4 si formattano, mediante espressione regolare, i dati inerenti ad una singola scheda Pad, dividendoli per spazi. Ciò sarà utile successivamente (linea 13) per poter estrarre solo il dato inerente all'ID della scheda Pad. Con la linea 5 si riempie il menù a tendina "*Select PAD*" (Fig. 3.6), mentre con le linee dalla 10 alla 13 si comunica con il processo *testpad* per selezionare la Pad scelta.

Sempre nella scheda "*Initialize*" trovano spazio tre opzioni: *Default init*, *Fast init* e *Step by step*. Il concetto di funzionamento delle singole opzioni è pressoché identico a quello di selezione dei vari menù a scheda. A titolo di esempio, viene adesso mostrato, in Fig. 4.9, il sorgente inerente all'evento scaturito dal pulsante "*Default init*".

```
1  private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
2      OutputStream outStream = inizio.p.getOutputStream();
3      PrintWriter pWriter = new PrintWriter(outStream);
4      pWriter.println("1");
5      pWriter.println("1");
6      pWriter.flush();
7      ...
8  }
```

Figura 4.9: Metodo inerente al pulsante "Default init"

Come già visto in precedenza, dapprima si incanala l'output del programma Java nell'input del processo *testpad* (linea 2), poi si converte l'oggetto istanziato in un flusso di dati di tipo testuale ed, infine, si selezionano le voci richieste dal menù testuale del programma da shell (linee 4 e 5).

Il click sul pulsante "*Step by step*" istanzia un nuovo form, inerente alla classe "*passoPasso*".

Figura 4.10: Form "Step by step"

Da qui è possibile personalizzare l'inizializzazione della scheda Pad selezionando, tramite i *radio button*, una configurazione di tipo veloce, di tipo "*warm*" (non inizializza le CM) o configurando le CM dal file selezionato.

Nella scheda "*PRODE*" si possono gestire i relativi chip. Oltre al menù a tendina per la scelta di uno dei quattro PRODE montati sulla Pad, è possibile inizializzare il chip selezionato, settarne i ritardi (in *ns*) dei quattro canali (*L1A*, *CLOCK*, *BCRST*, *L1RST*), verificarne lo stato di funzionamento e se i PRODE sono agganciati o meno al clock.

Figura 4.11: Form per settare il ritardo dei quattro canali del chip PRODE

In Fig. 4.11 è rappresentato il form che setta i ritardi dei canali del PRODE; esso è implementato mediante la classe "*setPRODE*". Dal menù "*Select Channel*" è possibile selezionare il canale al quale va applicato il ritardo (*Delay [ns]*). Cliccando sul pulsante "*Apply All*" è invece possibile settare il ritardo inserito contemporaneamente a tutti i canali del PRODE.

```
1  private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
2      ...
3      if(!(jTextField1.getText().equals("")) && isNumeric(jTextField1.getText())){
4          OutputStream outStream = inizio.p.getOutputStream();
5          PrintWriter pWriter = new PrintWriter(outStream);
6          pWriter.println("3");
7          pWriter.println(label[0]+" "+jTextField1.getText());
8          pWriter.flush();
9          inizio.print=1;
10     }
11     else{
12         Exception e = new Exception("Error!");
13         Component f = null;
14         JOptionPane.showMessageDialog(f, "Please insert a correct delay!", e.getMessage(), JOptionPane.WARNING_MESSAGE);
15     }
16 }
```

Figura 4.12: Evento scaturito dal click del pulsante "Apply" del form "Set PRODE"

In Fig. 4.12 è mostrato un estratto del sorgente inerente al metodo del pulsante "*Apply*". La linea 3 verifica se il campo "*Delay [ns]*" contiene un input corretto; se no, viene catturata un'eccezione (linee dalla 11 alla 15). Il ritardo viene inviato al processo *testpad* nel medesimo modo visto in precedenza (linea 7); con la variabile *label[0]* si identifica il canale scelto dalla *jComboBox*, al quale va impostato il ritardo. La stessa logica implementativa viene utilizzata anche per il metodo del pulsante "*Apply All*".

Nel menù "TTC" trovano spazio le opzioni di inizializzazione e di controllo del sistema di timing della scheda Pad.

La scheda "CM" gestisce i relativi chip e permette di inizializzarli (singolarmente o tutti insieme), leggerne e scriverne i registri e controllarne lo stato; anche in questo menù è possibile passare da una CM all'altra mediante una *jComboBox*.

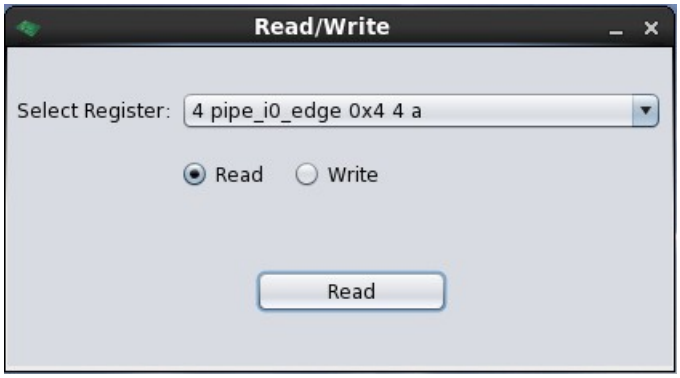

Figura 4.13: Form di lettura/scrittura dei registri della CM

In Fig. 4.13 è rappresentato il form di lettura e scrittura dei registri della CM. Dal menù "*Select Register*" è possibile selezionare uno dei 158 registri e con i radio button si sceglie se leggerne o scriverne il contenuto; nel caso di quest'ultima scelta, si abilita un campo di inserimento testo (*jTextField* di Swing), affiancato da una label che riporta la dimensione specifica del dato da inserire. Il metodo inerente al pulsante "*Read/Write*" segue la logica di quello "*Apply*" del form *Set PRODE*.

Il menù “FPGA” gestisce tutto ciò che riguardi il circuito integrato montato sulla Pad. È possibile inizializzarla dal file di configurazione, leggerne e scriverne i registri (Fig. 4.14) e visualizzarne lo stato di funzionamento.

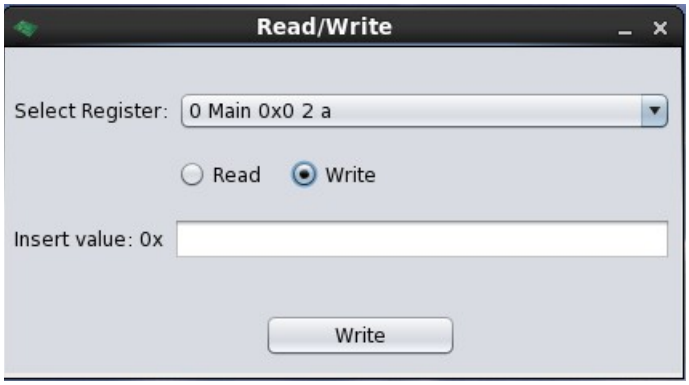

Figura 4.14: Form di lettura/scrittura dei registri della FPGA

Nella scheda “Link” trovano spazio le opzioni inerenti al GLinkTX. È possibile inizializzare il laser e visualizzarne varie informazioni sullo stato (ad esempio, se è agganciato o meno al clock).

Dal menù “ELMB” è possibile stampare i dettagli inerenti al firmware installato sulla scheda stessa e la versione hardware della Pad.

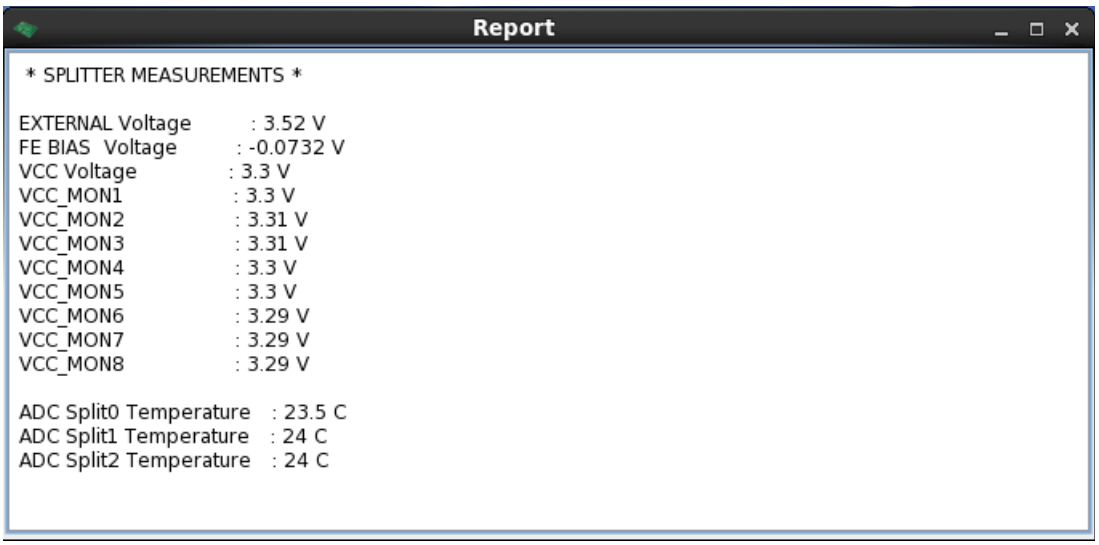

Figura 4.15: Voltaggio e temperatura dello Splitter collegato alla Pad

Il menù “Splitter” permette di inizializzare il modulo stesso, di vederne lo stato di funzionamento e di interrogare i sensori di temperatura e di tensione per leggerne i vari dati acquisiti (Fig. 4.15). Della scheda “Test” se ne discuterà nel prossimo paragrafo.

Tutti i report testuali, generati dalle interazioni con l'hardware, vengono gestiti dalla classe “*risultati*”. Se la check box “*Show Report*” è selezionata e l'output è pronto (nel sorgente: *if(menu.jCheckBox1.isSelected() && print==1)*), il thread che gestisce il driver *testpad* istanzia un oggetto della suddetta classe.

```
1  private void jPanel1ComponentAdded(java.awt.event.ContainerEvent evt) {
2      jTextArea1.setText(null);
3      ...
4      //Controllo stampa
5      if(menu.status==0){ //Stampa normale
6          jTextArea1.append(inizio.sbuff.substring(0, inizio.sbuff.length()));
7      }
8      ...
9      else{ //Stampa Test
10         int flag=0;
11         String pad = null;
12         for(String str : inizio.sbuff_Test){ //Cicla su tutto l'output
13             //Scarta righe inutili
14             if(str.length()>1 && !Character.isDigit(str.charAt(0)) && !str.equals(pad) ){
15                 ...
16                 jTextArea1.append(str+"\n");
17                 flag=0;
18                 if(str.contains("LowPT") || str.contains("HighPT")){ //Evita la ripetizione sulla pad selezionata
19                     pad=str;
20                 }
21             }
22             else{
23                 if(flag==0){ //Aggiungi una riga vuota al posto delle righe eliminate
24                     jTextArea1.append("\n");
25                     flag=1;
26                 }
27             }
28         }
29     }
30     menu.status=0;
31     sbuff.setLength(0);
32 }
```

Figura 4.16: Estratto del sorgente inerente alla visualizzazione dei report

L'output viene presentato in una *jTextArea* di Swing, la quale contiene al suo interno il contenuto della variabile *sbuff*, processata dal thread nella classe “*inizio*”. In caso di report proveniente dalla prime otto schede del menù, questo viene elaborato, nella maggior parte dei casi (per motivi di spazio non sono state messe tutte le possibili eccezioni), dalla linea 6 del codice in Fig. 4.16. Nel caso inerente alla

stampa dei risultati del test (opzione contenuta nell'ultima scheda del menù), si processa l'output mediante la logica implementata dalle righe dalla 9 alla 29. Qui si valutano quali informazioni far visualizzare o meno a video, dato che l'output generato dal driver *testpad* mostra spesso messaggi ridondanti o addirittura confusionari. Così facendo, ad esempio eliminando le voci del menù da shell (linea 14) o evitando ogni volta di citare la Pad che si sta testando (linea 18), si migliora la qualità dell'interpretazione dei risultati del test da parte degli addetti ai lavori. Al posto delle righe omesse viene semplicemente inserito un carattere di ritorno a capo (linea 24).

4.3 Test e risultati ottenuti

L'ultima voce presente nel menù a schede è "Test". Qui trovano spazio tre opzioni:

- *Test*: è la nuova modalità di inizializzazione e di verifica del funzionamento delle schede *Add-on-board* da me implementa. Permette il test simultaneo di più Pad collegate tra loro in cascata ad un canale CAN, inizializzando prima tutte le schede di low_{PT} e poi quelle di $high_{PT}$.
- *PAD Status*: rapido test che verifica lo stato di accensione della Pad, se il TTC ed i chip PRODE sono agganciati al clock, lo stato di funzionamento della FPGA e quello delle quattro CM.
- *Measurement*: misure le temperature delle schede *Add-on-board*, i voltaggi e mostra le componenti con eventuali problemi di sovracorrente ("*overcurrent*").

Sulle ultime due opzioni c'è poco da dire. L'attenzione va invece rivolta alla voce "*Test*", implementata dalla omonima classe. Questa si differenzia dall'inizializzazione di default (scheda "*Initialize*", opzione "*Default Init*") per i seguenti motivi:

- è stata ottimizzata la parte inerente al settaggio dei registri delle CM. Dato che, ai fini del test di funzionamento, è interessante l'avvenuta

comunicazione tra il server e le quattro CM, si inizializza, per ognuna di esse, soltanto il registro "*main control*", scrivendo e leggendo 3 byte generici al suo interno. L'inizializzazione di default prevedeva, invece, la scrittura, sempre per ogni CM, di 87 registri dei 158 presenti, senza, tra l'altro, verificarne l'avvenuta memorizzazione del dato scritto. Tutto ciò ha portato ad una sostanziale ottimizzazione dei tempi di test, i quali verranno analizzati nel prosieguo del capitolo.

- I test adesso sono personalizzati in base alla tipologia della Pad. Nel caso di una low_{PT} vengono eseguite, nell'ordine, le seguenti operazioni:
    1. viene visualizzata la versione del firmware installato sulla ELMB;
    2. viene inizializzato il TTC;
    3. vengono impostati i tempi di ritardo dei canali dei quattro chip PRODE;
    4. si verifica se le quattro CM sono agganciate al clock e vengono settati i quattro registri "*main control*" per testare la comunicazione tra il server e le CM;
    5. vengono visualizzate le temperature ed i voltaggi delle componenti della Pad;
    6. se selezionata l'apposita opzione, viene inizializzato lo Splitter e ne viene mostrato lo stato di funzionamento.

  Al contrario, nel caso del test di una Pad di $high_{PT}$, alle sei operazioni di sopra si aggiungono anche le seguenti:

    7. viene scritto e verificato un valore generico nel "*main register*" della FPGA; successivamente viene visualizzato lo stato di funzionamento del circuito integrato;
    8. viene inizializzato il modulo GLinkTX.

  Nell'inizializzazione di default ciò non accadeva e, per entrambe le tipologie di Pad, si controllavano anche componenti inutili ai fini del test (ad esempio la FPGA nella Pad di low_{PT}).

- Come già accennato in precedenza, ora è possibile testare più Pad contemporaneamente senza dover necessariamente passare da una scheda all'altra (in caso di collegamento multiplo) o, addirittura, riavviare la sessione di *testpad* (nel caso di collegamento ad una singola Pad). Ovviamente, per poter sfruttare tale feature, le schede devono essere collegate in cascata tra di loro (mediante la seconda interfaccia CAN di cui sono dotate) dato che il software permette la connessione tramite una sola porta CAN per sessione. Per motivi inerenti all'esperimento, vengono prima inizializzate e testate tutte le Pad di low_{PT} e solo successivamente quelle di $high_{PT}$.

In Fig. 4.17 è possibile osservare il form che viene mostrato a video dopo aver cliccato sul pulsante "*Test*".

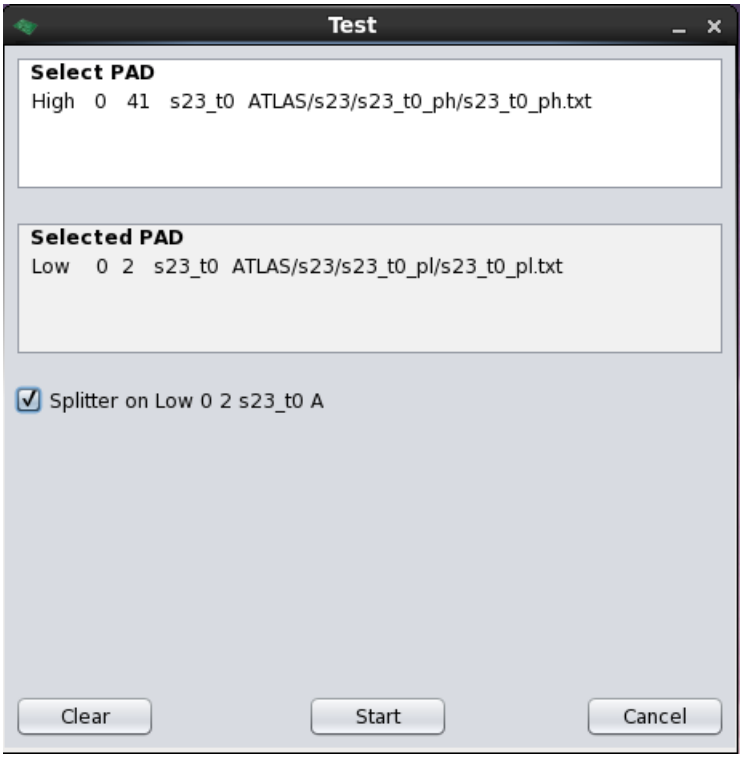

Figura 4.17: Schermata inerente alla fase di test

Nella parte superiore della schermata trova spazio l'elenco delle schede Pad collegate alla porta CAN, quest'ultima selezionata al momento della connessione (nell'esempio, il canale CAN è "0" e le Pad collegate sono la "2" e la "41").

Cliccando col mouse su una di esse, la Pad scelta viene aggiunta all'elenco delle schede selezionate ("*Selected PAD*") ed, automaticamente, apparirà una *check box* che permetterà all'utente di indicare, o meno, la presenza di uno Splitter collegato ad essa. Sempre nell'esempio in Fig. 4.17, la Pad selezionata è quella di low_{PT} con ID "2" e, selezionando la *check box*, si informa il programma che è richiesto eseguire i test inerenti allo Splitter collegato ad essa. Ogniqualvolta viene selezionata una Pad, questa viene rimossa dall'elenco delle schede selezionabili; per poter tornare allo stato iniziale o semplicemente ovviare ad una scelta errata, è possibile cliccare sul pulsante "*Clear*". L'inizio del test è scandito dal click sul bottone "*Start*" che invocherà il metodo

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt)

```
1  private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
2      HashMap<String, HashMap<String,String>> map = new HashMap<>();
3      String label[];
4      JCheckBox casella=new JCheckBox();
5      ...
6      if(jList2.getModel().getSize()>0){ //Se sono state selezionate PAD
7          for(int i=0; i<jPanel2.getComponentCount(); i++){ //Controlla se le PAD hanno gli splitter
8              HashMap<String, String> map2 = new HashMap<>();
9              if( (casella=(JCheckBox) jPanel2.getComponent(i)).isSelected() ){ //Si
10                 label=(casella.getText()).split(" ");
11                 ...
12                 map2.put(label[4], "1");
13                 ...
14             }
15             else{ //No
16                 ...
17                 map2.put(label[4], "0");
18                 ...
19             }
20             map.put(label[2], map2);
21         }
22         //Ordinamento e processamento: prima le Low e poi le High
23         SortedSet<String> keys = new TreeSet<>(map.keySet()).descendingSet();
24         ...
```

Figura 4.18: Estratto del sorgente relativo al metodo del pulsante "Start"

In Fig. 4.18 è mostrata la prima parte del sorgente del metodo succitato. Prima di tutto, si valuta la presenza o meno di Pad selezionate dal menù "*Select PAD*" (linea 6). Fatto ciò, si controlla su quali schede bisogna testare l'eventuale Splitter collegato. Il *for* (linea 7) cicla sulle componenti aggiunte al pannello "*jPanel2*" di Swing, le quali altro non sono che le *check box* che denotano la presenza (o meno)

dello Splitter sulla relativa Pad. Il "check" vero e proprio viene eseguito dalla condizione dell'*if* (linea 9): se la casella è selezionata, viene aggiunta una "flag" (*1* Splitter presente, *0* no) alla struttura dati *map2* (una *HashMap*, un tipo di array parametrico), la quale identifica la i-esima Pad da testare. Questa è composta da due campi:

- *ID Pad*: rappresenta l'identificatore univoco della scheda;
- *flag Splitter*: indica se è stata selezionata la *check box* inerente alla presenza dello Splitter.

Questi dati vanno aggiunti, a loro volta, in un altro array *map*, sempre di tipo *HashMap*, che ne amplia i dettagli aggiungendo un primo campo rappresentante il tipo di Pad (low_{PT} o $high_{PT}$). In definitiva, la struttura dati, che rappresenta una scheda scelta per il test, è così strutturata:

HashMap<String (Tipo Pad), HashMap<String (ID Pad), String (Flag Splitter)>>

Si è scelta come chiave dell'array *map* il tipo di scheda (e non l'ID di quest'ultima) proprio per poter ordinare le Pad in base a questa caratteristica, dato che, ai fini del test, ci necessita verificare il corretto funzionamento prima delle schede di low_{PT} e poi di quelle di $high_{PT}$. Questo spiega il significato della linea 23, dove ordiniamo le chiavi dell'array *map* in modo discendente e le memorizziamo in una struttura ordinata di stringhe (*keys*) di tipo *SortedSet*.

Questo vettore viene utilizzato nella seconda parte del codice, mostrato in Fig. 4.19. Dopo la consueta modalità di "collegamento" dell'I/O tra il driver *testpad* e l'applicativo in Java (linee 1 e 2), si cicla sul vettore *keys*: ogni chiave prelevata da questo array viene utilizzata per estrarre la relativa Pad, associata ad essa, contenuta nella variabile *map* (linea 4). Una volta recuperati tutti i dati della scheda, questi vengono convertiti in stringa di testo, formattati mediante espressioni regolari ed, infine, utilizzati per poter comunicare con il driver *testpad*. Si ha quindi la fase di selezione della Pad, di test della ELMB, di test del TTC, di test dei PRODE, di test delle CM, la scrittura e la verifica dei registri, ecc.

```
1          OutputStream outStream = inizio.p.getOutputStream();
2          PrintWriter pWriter = new PrintWriter(outStream);
3          for (String key : keys) { //Cicla su tutte le PAD scelte
4              value = map.get(key).toString();
5              ...
6              label=value.split("=");
7              //Scegli PAD
8              pWriter.println("10");
9              ...
10             //ELMB, TTC, PRODE, Check CM
11             ...
12             //Write and read CM Phi0, Write and read CM Phi1
13             //Write and read CM Eta0, Write and read CM Eta1
14             ...
15             //PAD Measurement
16             ...
17             //Se la PAD ha lo splitter
18             if(label[1].equals("1")){
19                 ...
20             }
21             //Se la PAD è High
22             if(key.equals("High")){
23                 //FPGA, Link
24                 ...
25             }
26             pWriter.flush();
27         }
28         this.dispose();
29     }
30     else{ //Nessuna PAD selezionata
31         Exception e = new Exception("Missing Parameters!");
32         Component f = null;
33         JOptionPane.showMessageDialog(f, "None PAD selected!", e.getMessage(), JOptionPane.WARNING_MESSAGE);
34     }
35 }
```

Figura 4.19: Seconda parte del sorgente relativo al metodo del pulsante "Start"

Il controllo sulla presenza dello Splitter viene effettuato dalla condizione dell'*if* alla linea 18, verificando la relativa *flag* memorizzata in precedenza. Il test sulla FPGA e sul GLinkTX viene eseguito soltanto se la Pad è di tipo $high_{PT}$; tale controllo viene effettuato sulla variabile *key* attuale, alla linea 22.

Il metodo si conclude con l'eventuale eccezione lanciata in caso di nessuna Pad selezionata al momento del click sul pulsante "*Start*" (linee dalla 30 alla 34).

Un altro metodo interessante da analizzare, nella classe "*Test*", è

private void jList1MouseClicked(java.awt.event.MouseEvent evt)

(Fig. 4.20) il quale cattura l'evento scaturito dal click del mouse sulla lista di selezione delle Pad. Gli elementi selezionati vengono gestiti da due liste di stringhe ("*selected*" e "*selectedValues*", linee 3 e 5), le quali vengono aggiornate appena viene scelta una nuova Pad per il test.

```
1  private void jList1MouseClicked(java.awt.event.MouseEvent evt) {
2      //Elementi selezionati
3      List<String> selected = jList1.getSelectedValuesList();
4      //Lista di stringhe con gli elementi selezionati.
5      selectedValues.addAll(selected);
6
7      //Aggiungi check box per splitter
8      jPanel2.setLayout(new javax.swing.BoxLayout(jPanel2, javax.swing.BoxLayout.Y_AXIS));
9      if(flag==1){ //Se ci sono più PAD gestisci la stringa in questo modo
10         jPanel2.add(new JCheckBox("Splitter on "+ ((selected.get(selected.size()-1)).replaceAll("\\s+", " ")).substring(0, 16) ));
11     }
12     else{
13         jPanel2.add(new JCheckBox("Splitter on"+ ( ((selected.get(selected.size()-1)).replaceAll("\\s+", " ")).replaceFirst("PT", "") ) ));
14     }
15     jPanel2.revalidate(); //Aggiorna il pannello
16
17     //Aggiungi alla lista 2
18     jList2.setListData(selectedValues.toArray(new String[0]));
19
20     //Elimina dalla lista 1
21     List<String>remainingValues = new LinkedList<>();
22     for(int index=0; index<jList1.getModel().getSize(); index++){
23         String str = jList1.getModel().getElementAt(index).toString();
24         if( ! selectedValues.contains(str)){
25             remainingValues.add(str);
26         }
27     }
28
29     jList1.setListData(remainingValues.toArray());
30 }
```

Figura 4.20: Metodo inerente alla selezione di una Pad per effettuarne il test

Per ogni selezione c'è l'aggiunta di una *check box* nel pannello *jPanel2* di Swing (linee dalla 8 alla 15). Dopo aver configurato l'impaginazione (il *layout*) del pannello (linea 8), vengono aggiunte le caselle di controllo delle relative Pad (linee 10 e 13) per poter permettere all'utente di indicare la presenza o meno dello Splitter. Fatto ciò, le schede selezionate vengono aggiunte alla lista "*Selected PAD*" (linea 18) ed eliminate dalla lista "*Select PAD*" (linee dalla 21 alla 29): ciò avviene ciclando su tutte le schede contenute in quest'ultima lista (linea 22) e verificando se sono contenute nell'array *selectedValues* (linea 24). Se questo non avviene, le Pad non scelte vengono aggiunte nella struttura dati *remainingValues* (sempre di tipo *List<String>*), la quale servirà a settare, appunto, la lista delle schede ancora non selezionate (linea 29).

Può essere ora svolta un'analisi dei tempi di esecuzione delle due modalità di inizializzazione e test: quella di default, già presente nell'applicativo *testpad*, e quella semplificata, appena descritta. Come già accennato, l'inizializzazione di default setta 87 registri per ciascuna CM (tra cui quelli di *trigger*, di *Level 1 Accept*, di *ReadOut*). Ciò è di fondamentale importanza nell'ambito dell'esperimento

ATLAS dato che, in base ai valori contenuti nei registri, ne consegue tutto il funzionamento dell'elettronica utilizzata e dell'algoritmo di trigger.

Ai fini del test, invece, si è pensato ad un'inizializzazione ad hoc dell'hardware; ciò è stato fatto per poter verificare soltanto lo stato di funzionamento delle schede *Add-on-board*. In particolar modo, per testare le comunicazioni con le CM basta scrivere (e leggere) soltanto un registro (è stato scelto il *main register*). Così facendo, si è arrivati ad ottimizzare le prestazioni temporali del test di circa un fattore 8.

Nella Tab. 4.1 sono mostrati tutti gli esperimenti effettuati.

N° Pad	Default Init	New Test
1	96 sec.	12 sec.
10	960 sec.	120 sec.
30	2880 sec.	360 sec.
50	4800 sec.	600 sec.
100	9600 sec.	1200 sec.

Tabella 4.1: Tempi di esecuzione dell'inizializzazione di default e della nuova modalità di test in relazione al numero di Pad testate

Com'era facilmente prevedibile, l'ottimizzazione nella scrittura dei registri delle quattro CM porta notevoli vantaggi nei tempi di esecuzione del test. Adesso, con quasi lo stesso tempo impiegato in precedenza per inizializzare 10 Pad, è possibile testare il funzionamento di ben 100 schede, ammortizzando, appunto, il tempo di un fattore 8.

Nel grafico in Fig. 4.21 si evince ancor di più il gap di performance tra l'inizializzazione standard e la nuova modalità di test implementata, rendendo ancor

più l'idea di come, ai fini del test, sia conveniente non inizializzare tutti i registri delle quattro CM.

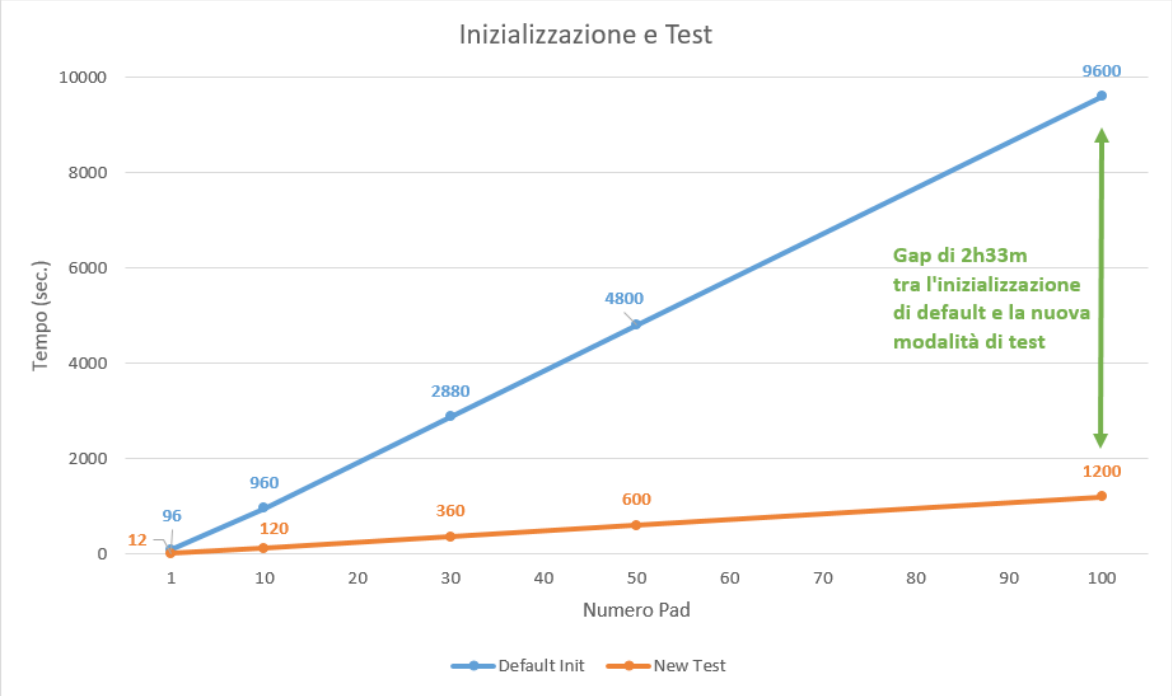

Figura 4.21: Tempi di esecuzione dell'inizializzazione standard e della nuova modalità di test

4.4 Casi d'uso ed esempi di test

In questa sezione viene presentata una particolare applicazione del tool implementato su due schede Pad (una di low_{PT} con ID 2 e l'altra di tipo $high_{PT}$ con ID 20), le quali sono collegate tra loro in cascata alla porta 0 del modulo *USB-CANMODUL16*. Si deve notare che tale caso di studio è di fondamentale importanza dato che, proprio durante la stesura di questa tesi, molte schede Pad sono in fase di controllo ed, eventualmente, in fase di riparazione. Quindi, considerando il loro ruolo primario nel funzionamento dell'intero sistema (LHC), è necessaria una loro completa ed esaustiva validazione.

Il controllo delle Pad si divide essenzialmente in due parti:

- una prima fase di selezione delle schede da verificare, con eventuale configurazione del test comunicando la presenza dello Splitter;
- una seconda fase riguardante il test vero e proprio, personalizzato in base al tipo di Pad da controllare.

Come già anticipato, per entrambe le tipologie di scheda viene verificato il corretto funzionamento della ELMB, del TTC, dei chip PRODE, delle CM, dello Splitter (se connesso); in più, per le Pad di $high_{PT}$, si verifica la FPGA ed il GLinkTX.

Prima di addentrarsi nel caso di studio scelto, si mostrano tutte le possibilità offerte all'utente per poter interagire con l'hardware sotto test (Fig. 4.22).

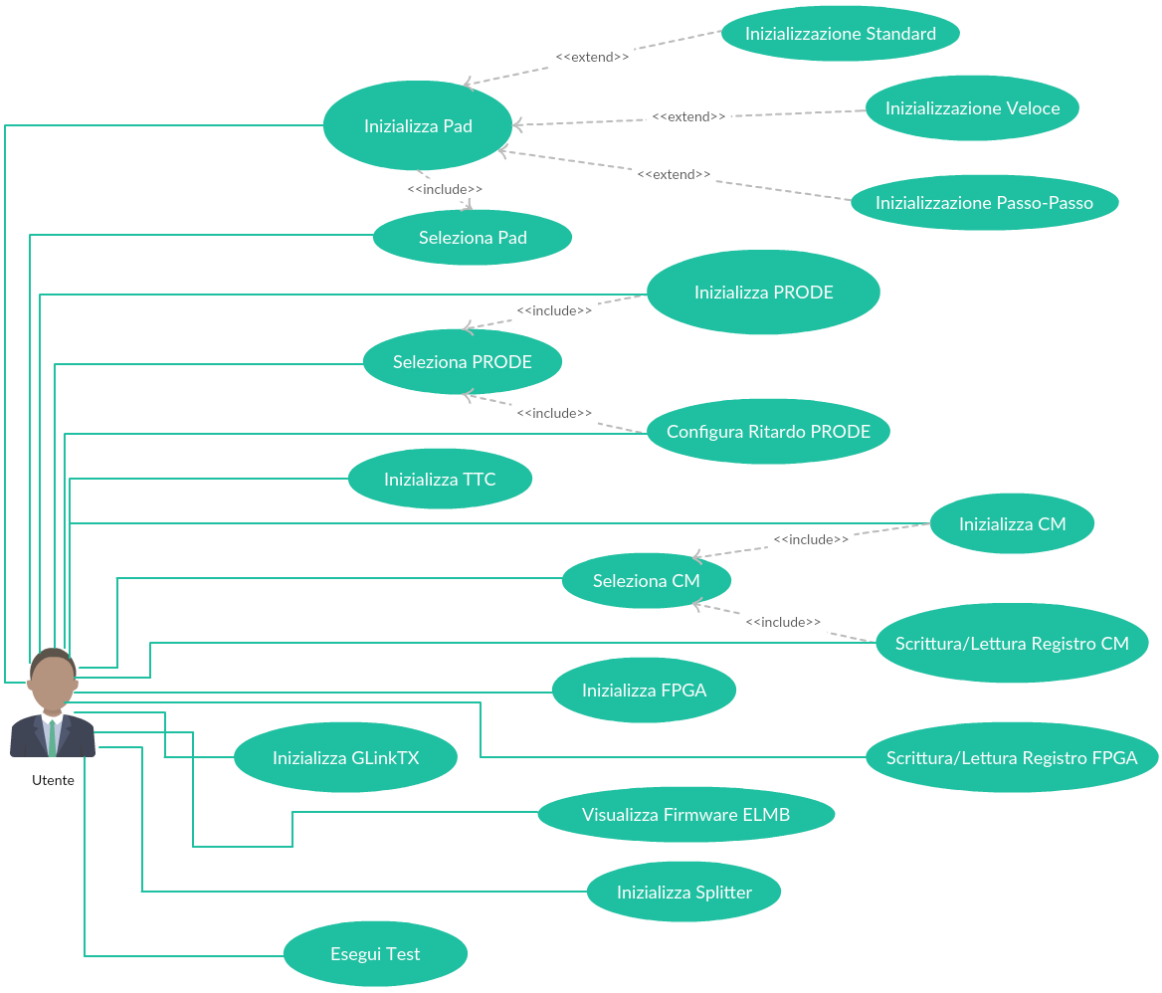

Figura 4.22: Use case dell'applicativo in Java

Viene mostrata, in Fig. 4.23, la sequenza temporale (mediante *Sequence Diagram*) dei messaggi che gli oggetti si scambiano per portare a termine la funzionalità "*Esegui Test*".

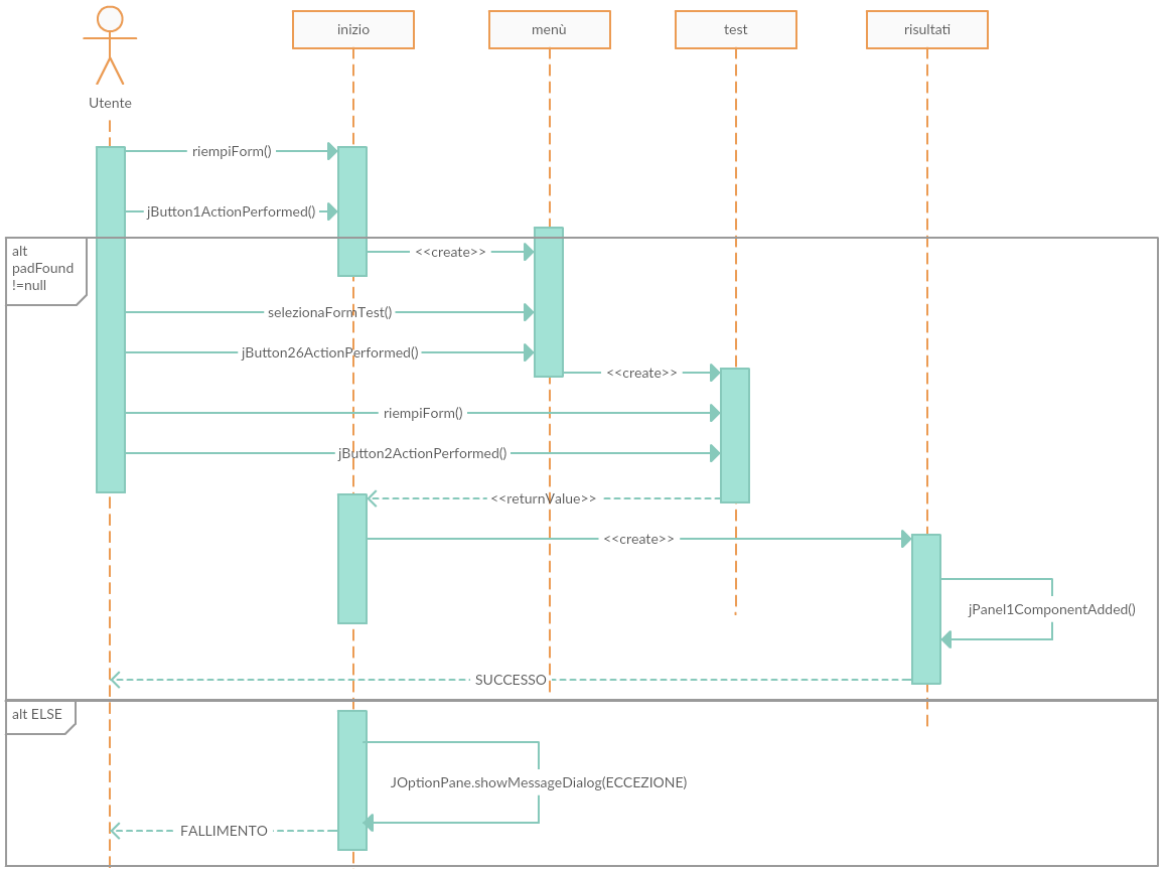

Figura 4.23: Sequence Diagram della funzionalità "Esegui Test"

Per quanto riguarda il caso di studio scelto ("*Esegui Test*" in Fig. 4.22), l'obiettivo che ci si è proposti è stato quello di validare le Pad succitate dando una particolare attenzione ai report generati dall'esecuzione del test. Dapprima è stata controllata la Pad di low_{PT} con ID 2 che ha riscontrato una serie di problematiche su alcune sue schede *Add-on-board*. Successivamente, dopo l'intervento del tecnico addetto alla riparazione, si è ripetuto il test per verificare l'efficacia delle riparazioni effettuate. Infine, è stato effettuato un ulteriore test comunicando, erroneamente, la presenza dello Splitter collegato alla Pad per verificare la presenza dell'eventuale messaggio di errore. I risultati del primo test (Fig. 4.24) hanno riportato i seguenti errori:

Figura 4.24: Report degli errori sulla Pad di lowPT (ID 2)

Come è possibile notare in questo estratto del report, la ELMB ha avuto problemi di comunicazione dato che non ha restituito la versione del firmware installato e la versione hardware della Pad. È stato poi riscontrato un problema nella scrittura dei registri del TTC: ciò è intuibile dall'errore del protocollo SDO (utilizzato dal bus CAN per poter scrivere e leggere i valori dei registri dei device in remoto). Questo implica anche che il TTC non agganci il clock di LHC. Infine, è stato segnalato il mancato aggancio del clock da parte delle quattro CM; questo però non ha influito sulla scrittura dei registri, attestando il corretto funzionamento dei quattro microcontrollori. Dopo l'intervento del tecnico è stato ripetuto il test; i risultati ottenuti sono riportati nel report in Fig. 4.25.

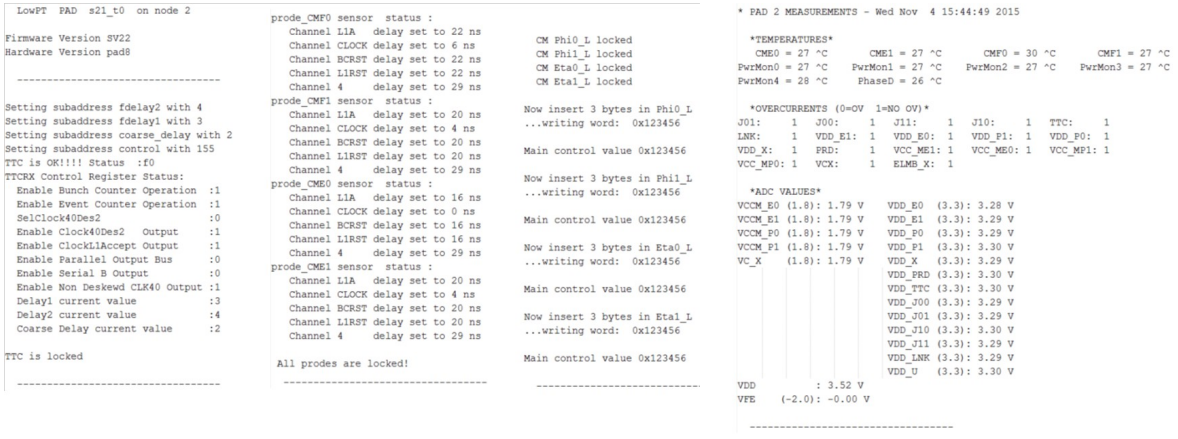

Figura 4.25: Report della Pad di lowPT (ID 2) dopo l'intervento del tecnico

E' possibile osservare che tutti i precedenti errori sono stati risolti (per completezza, nella figura precedente è stato riportato tutto il report, comprensivo dei risultati ottenuti sui chip PRODE e delle misure di temperatura e tensione della Pad).
L'ultimo test che è stato eseguito sulla Pad di low_{PT} è stato quello di comunicare, erroneamente, la presenza di uno Splitter collegato alla suddetta scheda. I risultati ottenuti (solo quelli inerenti, appunto, allo Splitter) sono mostrati in Fig. 4.26.

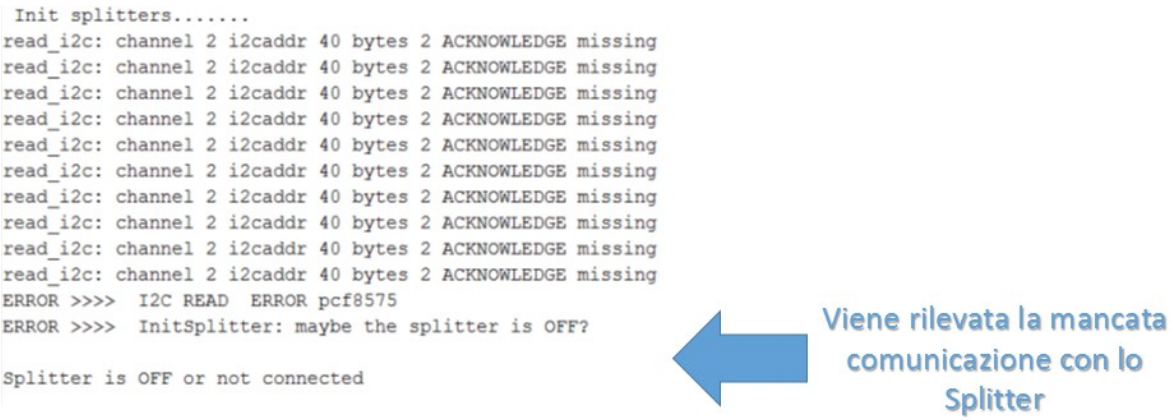

Figura 4.26: Errore rilevato per il mancato collegamento dello Splitter

Infine è stata controllata la Pad di $high_{PT}$ (ID 20) per mostrare le differenze del test inerenti a questa tipologia di scheda. In particolare, nell'ultimo report mostrato in Fig. 4.27, è possibile visionare lo stato della FPGA e dell'uscita ottica GLinkTX; alla Pad è stato collegato uno Splitter per mostrare le informazioni ricavate da questo componente.

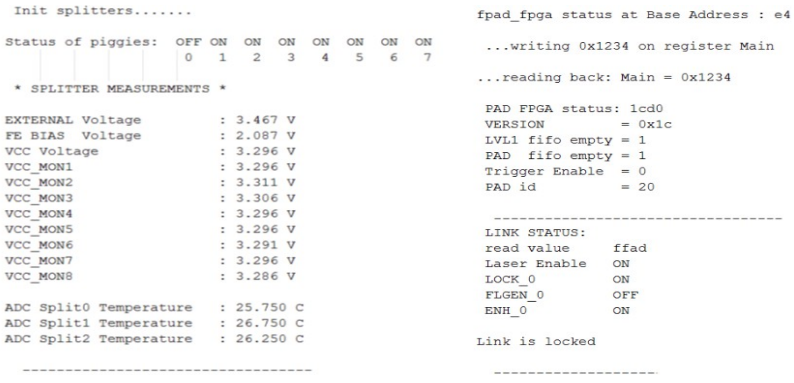

Figura 4.27: Report inerente allo Splitter, alla FPGA ed al GLinkTX della Pad di highPT con ID 20

Conclusioni

Il lavoro svolto nell'ambito di questa tesi ha permesso di inserire, all'interno del TDAQ dell'esperimento ATLAS, un nuovo ambiente di test per l'elettronica di trigger di livello 1. Il programma da me elaborato in questa sede è attualmente utilizzato nella fase di test, sulle schede Pad, presso il sito BB5 del CERN.

Durante questo periodo ho effettuato il debug di un software preesistente (scritto in C++), utilizzato per l'interfacciamento dell'hardware da testare; di questo ne ho esteso alcune sue funzionalità e ne ho realizzato un'interfaccia grafica implementata in Java.

Prima della conclusione del mio lavoro, il test di funzionamento di una scheda Pad era effettuato tramite la modalità di inizializzazione, dell'intero hardware, offerta dal driver/software *testpad*; questa impiegava circa 96 secondi a scheda. Adesso invece il test di una Pad è gestito da una nuova feature presente all'interno dell'interfaccia grafica da me implementata in Java; il tempo di verifica del funzionamento delle schede *Add-on-board* avviene ora in 12 secondi. Questa forte riduzione dei tempi di test è stata resa possibile andando a semplificare la fase di inizializzazione della Pad, eliminando tutto ciò che fosse inutile al fine della verifica del funzionamento della scheda. Più nel dettaglio, è stata snellita la parte inerente alla scrittura di tutti i registri delle quattro CM, operazione utile solo al fine della presa dati.

Inoltre, del software esistente di interfacciamento dell'hardware (questo scritto in C++) ne è stato fatto il debug, andando a correggerne parti critiche come l'inizializzazione dei registri della FPGA ed il settaggio dei tempi di ritardo dei canali dei chip PRODE.

In futuro si prevede di rinnovare il driver di interfacciamento della Pad. A partire da gennaio 2016, tutti i software del CERN dovranno migrare su architetture a 64 bit e, essendo il driver *testpad* pieno di utili funzionalità ma scritto a 32 bit (avendo più di 10 anni di vita alle spalle), ciò implicherà un rinnovamento dell'interfaccia utilizzata

per comunicare, mediante bus CAN, con la scheda Pad. Per l'implementazione del driver sono in fase progettuale strategie atte alla minimizzazione ed all'ottimizzazione di quest'ultimo, andando ad eliminare tutte le componenti non funzionanti o ormai deprecate. Anche per il driver del CAN è prevista una fase di riscrittura del sorgente per renderlo compatibile con la nuova architettura a 64 bit.
Per quanto riguarda il software in Java implementato, se ne potranno estendere le funzionalità e le modalità di test previste. Ad esempio, si potrebbero implementare delle feature che vadano ad interessare altre componenti hardware utilizzate nel trigger di livello 1 di ATLAS; ROD e Sector Logic sono elementi critici presenti all'interno dell'esperimento e necessitano anch'essi di una nuova modalità di test che ne vada a verificare il corretto funzionamento. Infine si è pensato di accostare al report testuale anche uno grafico (ad esempio, facendo apparire l'immagine del componente danneggiato), andando a semplificare ancor di più il rilevamento del guasto da parte degli addetti ai lavori.

Appendice A

A - Il protocollo CAN

Il protocollo CAN (*Controlled Area Network*) è un bus seriale di comunicazione digitale progettato per applicazioni real-time. Le caratteristiche del protocollo CAN permettono a controllori, sensori ed attuatori di:

- comunicare fino ad 1 Mbit/sec;
- lavorare in condizioni ambientali ostili;
- svolgere funzioni di autodiagnostica;
- avere bassi costi per l'hardware di controllo (CAN Controller);
- permettere l'invio di messaggi senza indirizzo di destinazione (un messaggio inviato da un nodo è ricevuto da tutti i nodi collegati al bus e viene filtrato dagli stessi in modo che ognuno utilizzi i soli messaggi che ritiene utili);
- rilevare automaticamente gli errori;
- ritrasmettere automaticamente i dati in caso di errore;
- disconnettere automaticamente (in modo temporaneo o permanente) un nodo che presenta un sospetto guasto fisico;
- semplicità di cablaggio e di configurazione della rete;
- possibilità di inserire in un sistema nuovi nodi con modifiche minime di hardware e software.

Il protocollo CAN si colloca all'interno della pila ISO/OSI nel *Physical Layer* e *Data Link Layer*, lasciando totale libertà per quanto riguarda l'*Application Layer*.

A.1 Il Physical Layer

Il *Physical Layer* è il primo livello dello standard ISO/OSI e si occupa di tutti gli aspetti fisici della trasmissione dati tra i diversi nodi della rete.

Il cavo trasmissivo del CAN è un doppino intrecciato (schermato o meno a seconda delle applicazioni) terminato con impedenze di valore pari a 120 Ohm. La lunghezza massima del bus dipende dalla velocità trasmissiva usata. Il Physical Layer deve garantire la rappresentazione dello stato *dominante* (0 logico = 0 V) e dello stato *recessivo* (1 logico = 5 V). Il Physical Layer, per garantire che un bit dominante vada a sovrascrivere un contemporaneo bit recessivo deve:

- codificare/decodificare i bit e la loro sincronizzazione e temporizzazione;
- gestire il *Transceiver* (driver di linea);
- indicare le proprietà dei cavi e dei connettori.

A.2 Data Link Layer

Il *Data Link Layer* è il cuore del protocollo CAN perché definisce la politica di accesso al mezzo di trasmissione. Essa può essere riassunta nei seguenti punti:

- assenza di indirizzi mittente/destinatario: i pacchetti trasmessi da una stazione CAN non contengono indirizzi; essi sono sostituiti da un identificatore di *contenuto messaggio*, unico per tutta la rete. Un nodo ricevitore, verificando il contenuto del messaggio tramite questo identificatore, può decidere se filtrare o meno il pacchetto in trasmissione in quel preciso istante. Questa tecnica prende il nome di *Multicast* e permette un alto grado di modularità e flessibilità del sistema (ad esempio, non c'è il bisogno di dover assegnare un nuovo indirizzo di rete ad un dispositivo che viene aggiunto nel sistema CAN);
- arbitraggio non distruttivo: due o più stazioni che iniziano a trasmettere competono per l'accesso al bus sul loro valore di priorità determinata proprio dall'identificatore. Tale competizione è risolta in accordo con il meccanismo del wired-and, secondo il quale lo stato dominante (0 logico) sovrascrive lo stato recessivo (1 logico).

Altro compito importante del Data Link Layer è quello di definire il formato dei messaggi CAN; esistono quattro tipi di messaggi:

- **Data Frame**: serve per trasferire i dati;
- **Remote Frame**: serve per inviare la richiesta di trasmissione di un data frame con lo stesso identificatore;
- **Error Frame**: viene trasmesso da un qualsiasi nodo che rileva un errore;
- **Overload Frame**: è utilizzato per inserire un ritardo aggiuntivo tra due Data Frame o tra due Remote Frame successivi.

Come si può vedere in *fig. A.1*, il formato standard del Data Frame è costituito dai seguenti campi:

- **SOF** (*Start Of Frame*): è lungo un bit ed è sempre a livello dominante, indica l'inizio di un frame.
- **AF** (*Arbitration Field*): è lungo 12 bit e si divide in un identificatore di 11 bit (*CAN-identifier*) ed in un bit che rappresenta l'RTR (*Remote Transmission Request*). Quest'ultimo serve ad identificare il tipo di frame inviato: se è settato a 0 indica che il frame è un Data Frame, se settato ad 1 indica che è un Remote Frame. Il Remote Frame è una richiesta, da parte di un nodo, del Data Frame corrispondente (avente lo stesso identificatore) e non presenta un campo dati.
- **CF** (*Control Field*): è lungo 6 bit, due (R0 ed R1) riservati per sviluppi futuri del protocollo e quattro (DLC, *Data Lenght Code*) per indicare il numero di byte dei dati inviati.
- **DF** (*Data Frame*): contiene i dati inviati; la sua lunghezza può variare da 0 ad 8 byte.
- **CRC** (*Cyclic Redundancy Check*): contiene 15 bit di *CRC code* ed un bit recessivo come delimitatore; serve a verificare che l'invio dei dati sia avvenuto correttamente.

- **ACK** (*ACKnowledge*): è lungo 2 bit. Il primo bit è lo *Slot ACK* che è trasmesso come recessivo ma è sovrascritto con un bit dominante da ogni stazione che riceve correttamente il messaggio; il secondo bit è recessivo e svolge il compito di delimitatore.
- **EOF** (*End Of Frame*): è costituito da 7 bit recessivi ed indica la fine del frame.

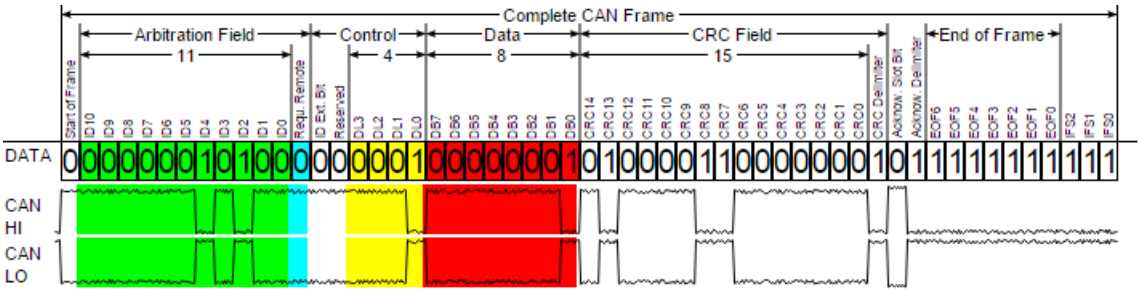

Figura A.1: Struttura del Data Frame

Su questo livello sono definite anche le procedure di rivelazione degli errori e le relative correzioni. Un errore può essere rivelato in cinque modi, tre dei quali a livello di messaggio e due a livello di singolo bit:

- **Bit Stuffing Error**: normalmente un nodo in trasmissione inserisce dopo 5 bit consecutivi della stessa polarità un bit di polarità opposta; ciò è chiamato *Bit Stuffing*. Un nodo che riceve più di 5 bit consecutivi della stessa polarità rileverà un errore di questo tipo.
- **Bit Error**: un nodo in trasmissione ascolta sempre il bus per verificare la corrispondenza con ciò che sta trasmettendo: se esso ascolta un bit diverso dal suo (che non appartiene all'Arbitration Field o all'ACK Slot) verrà segnalato un errore.
- **Checksum Error**: ogni ricevente ricalcola il CRC in base a ciò che ha ricevuto e se non corrisponde a quello inviato dal mittente viene segnalato un errore.

- **Frame Error**: viene segnalato questo tipo di errore quando vengono violati alcuni campi fissi del pacchetto.
- **Acknowledgement Error**: se il trasmettitore non ha alcun riscontro con il frame appena inviato.

Un *Error Frame* è costituito da un *Error Flag* ed un *Error Delimiter*. L'Error Flag è lungo 6 bit dominanti e viola volontariamente la regola del bit stuffing in modo che tutte le altre stazioni rilevino un errore e spediscano anch'esse un Error Flag. A seguito c'è un Error Delimiter costituito da 8 bit recessivi.

Il CAN offre anche un meccanismo di auto isolamento dei guasti: esso è in grado di distinguere tra condizioni di guasto transitorie (sbalzi di tensione, condizioni esterne di disturbo) e guasti permanenti (cattive connessioni, cavi rotti). Ogni CAN controller mantiene due registri che contano gli errori: uno in trasmissione e l'altro in ricezione. Essi sono inizialmente azzerati per essere incrementati ogni qualvolta si presenta un errore (+1 per un errore in ricezione, +8 per un errore in trasmissione). In questo modo ogni nodo CAN può trovarsi in uno dei seguenti tre stati:

- **Error Active**: nessuno dei due contatori ha superato il valore di 127. Il nodo è nel pieno delle sue funzionalità e decrementa di 1 i contatori ogni volta che riceve un messaggio andato a buon fine. Quando è in questo stato, il nodo che rileva un errore spedisce un Error Flag costituito da 6 bit dominanti in modo da interrompere sempre la trasmissione.
- **Error Passive**: almeno uno dei due contatori ha superato 127. Il nodo è ancora in grado di eseguire tutte le sue funzioni ma è probabile che esso presenti dei disturbi o condizioni di guasto. Per questo quando esso rileva un errore spedisce un Error Flag di 6 bit recessivi che vengono interpretati come errore solo se nessun nodo sta spedendo un suo proprio messaggio.
- **Bus Off**: se uno dei contatori supera 255 il nodo si stacca dal bus e non partecipa alla comunicazione lasciando gli altri nodi nella possibilità di

continuare a scambiare informazioni. In questo stato il nodo presenta un problema permanente che richiede un intervento esterno per renderlo di nuovo funzionante.

A.3 Application Layer

I protocolli di alto livello sono stati progettati per sfruttare al meglio le potenzialità del protocollo CAN. Essi definiscono:

- il comportamento dei nodi e quindi della rete nella fase di start-up;
- le regole con cui assegnare gli identificatori dei messaggi ai vari nodi;
- il controllo del flusso dei messaggi, la sincronizzazione delle funzioni e la definizione di un riferimento temporale;
- la possibilità di avere messaggi in grado di trasportare più dei tradizionali 8 byte di dati;
- il contenuto dei Data Frame, cioè forniscono uno standard di codifica dell'informazione;
- i meccanismi di report dello stato del sistema (notifica degli errori e diagnostica);
- un modello per la descrizione delle funzionalità e dei parametri del dispositivo;
- un appropriato modello di comunicazione che normalmente può essere di tipo “Broadcast” o “Client-Server”.

Ogni protocollo di alto livello implementa queste funzionalità in modo diverso adattandole alle specifiche esigenze del campo applicativo per cui è stato sviluppato. *CanOpen*, ad esempio, nasce per applicazioni industriali standardizzate, mentre *CAL* definisce una base standard da impiegare per l'ottimizzazione delle applicazioni di rete.

A.3.1 CAN Application Layer (CAL)

Il CAL (sviluppato da Philips) è il protocollo di alto livello adottato da un gruppo di produttori ed utilizzatori del CAN denominato *CiA User Group*. La sua peculiarità è quella di offrire un sistema di sviluppo per l'implementazione di reti distribuite basate sul CAN indipendente dall'applicazione da realizzare e consente un approccio *object-oriented* alla descrizione della stessa. Il CAL fornisce al progettista quattro servizi di alto livello:

1. **CMS** (*CAN-based Message Specification*): definisce 3 tipi di oggetti per descrivere le proprietà di un nodo CAN:
   1. *Variabile*, per trasmettere dati fino ad un massimo di 8 byte.
   2. *Evento*, per le operazioni di inizializzazione della trasmissione.
   3. *Dominio*, per trasmettere dati con dimensioni superiori agli 8 byte.
2. **NMT** (*Network ManagemenT*): offre servizi di supporto per la gestione della rete come la sua inizializzazione, start e stop dei nodi, rivelazione di malfunzionamenti; il servizio è implementato sfruttando connessioni di tipo master-slave.
3. **DBT** (*DistriBuTor*): offre una distribuzione dinamica degli identificatori CAN ai nodi della rete. Questi identificatori sono chiamati *Communication Object Identifier* (COB-ID).
4. **LMT** (*Layer ManagemenT*): offre la possibilità di modificare i parametri di configurazione dei nodi come ad esempio l'indirizzo NMT di uno dei dispositivi o il *bit timing* ed il *baud rate* della rete.

Il servizio CMS definisce otto classi di priorità per i messaggi ed, a loro volta, ogni classe dispone di 220 identificatori riservati. Gli identificatori 0 e da 1761 a 2031 sono dedicati agli altri servizi (tab. A.1).

CAN Application Layer	
COB-ID	**Servizio**
0	NMT servizi di start/stop
1-220	oggetti CMS con priorità 0
221-440	oggetti CMS con priorità 1
441-660	oggetti CMS con priorità 2
661-880	oggetti CMS con priorità 3
881-1100	oggetti CMS con priorità 4
1101-1320	oggetti CMS con priorità 5
1321-1540	oggetti CMS con priorità 6
1541-1760	oggetti CMS con priorità 7
1761-2015	NMT Node Guarding
2016-2031	servizi dei NMT, LMT, DBT

Tabella A.1: Identificatori associati ai servizi offerti dal CAL

A.4 CANopen

Il protocollo CAL mette a disposizione una completa serie di servizi per la gestione della rete e dei messaggi ma non definisce il contenuto degli oggetti CMS; sostanzialmente definisce come avviene la comunicazione e come il protocollo CAN viene utilizzato ma non definisce cosa transita sul bus. È questo il contesto dove si inserisce il *CANopen*.

CANopen usa un sottoinsieme dei servizi definiti dal CAL per fornire l'implementazione di un sistema di controllo distribuito completo. Esso è stato sviluppato dal CiA User Group per il controllo di sistemi industriali, successivamente impiegato anche per veicoli, strumenti elettromedicali e nella gestione dell'elettronica in campo navale e ferroviario. Questo protocollo di alto livello sfrutta il concetto dei *profile* per garantire ai sistemisti la possibilità di far

dialogare dispositivi realizzati da differenti produttori. Per mezzo dei *Device Profile*, le funzionalità più comuni sono a disposizione del progettista; funzionalità più specifiche (e strettamente legate all'hardware dell'interfaccia) sono gestite attraverso *profile* messi a disposizione dai produttori stessi.

A.4.1 Object Dictionary

I profili sono descritti attraverso un database di tipo standard detto *Object Dictionary* (OD). L'OD è un concetto base dell'implementazione CANopen: può essere schematizzato in una tabella in cui i vari gruppi di oggetti vengono identificati attraverso un indice a 16 bit e da qui accedere ai singoli elementi della struttura dati utilizzando un ulteriore sotto-indice a 8 bit.
Per ogni nodo esiste un OD che contiene tutti i parametri descrittivi del dispositivo ed il suo comportamento all'interno della rete. La parte dell'OD che descrive i parametri di comunicazione sono comuni a tutti i nodi mentre una sezione è riservata alla caratterizzazione del dispositivo. Il *Communication Profile* è unico e descrive la forma generale dell'OD e gli oggetti che devono contenere i parametri per la configurazione della comunicazione. Esistono invece più Device Profile che definiscono in differente maniera alcuni degli oggetti dell'OD; ad ogni oggetto viene assegnato un nome, la funzione, l'indice ed il sotto-indice (ossia la collocazione), il tipo di dati, se l'oggetto è obbligatorio o facoltativo, se l'oggetto è solo scrivibile, solo leggibile o scrivibile/leggibile, ecc. I campi di questa tabella sono accessibili in lettura ed in scrittura ed i messaggi, per mezzo dei quali avviene l'interrogazione e la risposta da parte dei nodi, sono detti *Service Data Object* (SDO).

A.4.2 Tipi di messaggi

Una rete CANopen deve avere un master ed uno o più nodi slave. Il master si occupa di seguire la sequenza di boot, manipola i campi dell'OD e gli identificatori CAN dei vari dispositivi connessi. I dati possono essere trasmessi in modo sincrono,

consentendo di coordinare l'acquisizione dei dati, oppure possono essere inviati in ogni istante, consentendo di notificare immediatamente ad un nodo una richiesta di trasferimento senza attenderne una sincrona. Il contenuto di entrambi i tipi di messaggi può essere riconfigurato al riavvio della rete dal master durante la fase di network management. I quattro tipi di messaggi previsti dal protocollo sono:

1. **Administrative Message**: sono i messaggi per la gestione della rete e la distribuzione degli identificatori. Sono messaggi ad elevata priorità che sfruttano i servizi NMT e DBT del CAL. Attraverso di essi, al riavvio del sistema, il master stabilisce un dialogo con ogni slave della rete e, dopo aver stabilito la connessione, trasmette le informazioni di configurazione al nodo.
2. **Special Function Objects**: in questa categoria rientrano messaggi predefiniti tra i quali:
   1. *SYNC*: utilizzato per la sincronizzazione del trasferimento dei dati, è un messaggio ad altissima priorità;
   2. *Time Stamp*: fornisce un riferimento temporale comune a tutti i nodi;
   3. *Emergency*: generato all'occorrenza per segnalare una o più condizioni di errore;
   4. *Node/Life Guarding*: il nodo che si occupa della gestione della rete (NMT master) monitora lo stato degli altri dispositivi. Ogni nodo viene continuamente interrogato per mezzo di un Remote Frame e risponde al master fornendo indicazioni sul proprio stato operativo. Si possono impostare tempi di inibizione (timeout) in modo tale che, se il master non riceve risposta entro un tempo prestabilito, esso identifica il nodo non funzionante e lo gestisce di conseguenza. CANopen prevede opzionalmente il controllo del nodo master da parte degli altri nodi e questa funzionalità è detta Life Guarding.
   5. *Boot-up*: inviando questo messaggio l'NMT slave comunica al master il suo stato operativo. Ogni slave funziona come una macchina a stati, dotata di quattro stati denominati: *Initialization, Preoperational,*

Operational e *Prepared Mode*. Attraverso specifici messaggi, il master è in grado di modificare lo stato di uno o più nodi simultaneamente.

3. **Process Data Object (PDO)**: sono utilizzati per il trasferimento real time dei dati; le informazioni trasferite sono limitate ad otto byte e la natura dei dati è codificata attraverso l'identificatore CAN (CAN-Identifier) associato al messaggio (la tipologia del dato definisce anche la priorità del messaggio e quindi l'importanza dell'informazione trasmessa). La trasmissione può avvenire in modo sincrono alla ricezione del messaggio SYNC, che ha cadenza periodica, o in modo asincrono su richiesta (a seguito della ricezione di un Remote Frame).
4. **Service Data Object (SDO)**: attraverso gli SDO, CANopen consente l'accesso ai campi dell'OD dei singoli dispositivi.

Gli SDO, essendo impiegati in fase di boot per la configurazione della rete, sono messaggi a bassa priorità che consentono il trasferimento di grosse quantità di dati (superiori agli otto byte), che costituiscono un limite invalicabile per i messaggi PDO. Chi richiede un accesso all'OD è chiamato *Client* mentre il componente CANopen che offre l'accesso al suo OD è chiamato *Server*. Ogni messaggio SDO, come qualsiasi messaggio CAN, è formato da 8 byte, sebbene non tutti contengano dati significativi. Ci sono due tipi di meccanismi per il trasferimento SDO:

- **Expedited Transfer**: usato per oggetti di dati lunghi fino a 4 byte;
- **Segmented Transfer**: per oggetti con lunghezza superiore a 4 byte.

A.4.3 Predefined Connection Set

Allo scopo di ridurre il lavoro del progettista, le specifiche CiA301 forniscono uno schema standard (*Predefined Connection Set*) per l'assegnazione degli identificatori. Per mezzo dei quattro bit più significativi (*Function Code Field*) vengono identificate 16 funzioni di base, che altro non sono che le quattro tipologie di

messaggi prima descritte ed ora riassunte nella tabella qui in basso. Il campo Node ID viene impostato, in ogni nodo, via harware per mezzo di un dip-switch.

Messaggio	Function code	COB-ID	Parametri OD
NMT	0000	000h	Alta priorità
SYNC	0001	080h	
TIME STAMP	0010	100h	
EMERGENCY	0010	081h-0FFh	080h+Node_ID
PDO1 (tx)	0011	181h-1FFh	180h+Node_ID
PDO1 (rx)	0100	201h-27Fh	200h+Node_ID
PDO2 (tx)	0101	281h-2FFh	280h+Node_ID
PDO2 (rx)	0110	301h-37Fh	300h+Node_ID
SDO (tx-server)	0011	581h-5FFh	580h+Node_ID
SDO (rx-client)	1100	601h-67Fh	600h+Node_ID
NMT Error Control	1110	701h-77Fh	580h+Node_ID

Tabella A.2: Identificatori per i messaggi del CANopen

Appendice B

B – Il protocollo I^2C

Il protocollo I^2C (*Inter Integrated Circuit*) è un sistema di comunicazione seriale utilizzato tra circuiti integrati. Esso specifica le regole di comunicazione fra più dispositivi collegati ad un bus formato da due linee; è stato ideato dalla Philips per permettere a dispositivi differenti di comunicare fra loro in modo da consentirne lo scambio di informazioni.

B.1 Specifiche del protocollo

Il protocollo I^2C prevede l'utilizzo di un bus formato da due linee bidirezionali. Le due linee, chiamate SCL (*Serial CLock*) ed SDA (*Serial DAta*), trasportano rispettivamente il segnale di clock e quello dei dati; i segnali che transitano sulle linee hanno valore 1 o 0. Le due linee non sono lasciate ad un valore indefinito ma vengono collegate all'alimentazione attraverso una resistenza di pull-up. Grazie ad essa, per ottenere un valore 1 sulla linea sarà sufficiente mettere il segnale di uscita in alta impedenza. In questo modo, se un dispositivo impone un valore 1 ed un altro un valore 0, quest'ultimo segnale sarà prevalente sul precedente.
Ogni dispositivo collegato alle due linee è dotato di un indirizzo univoco (di 7 o 10 bit) e può agire sia da master che da slave. Il master si occupa di iniziare la trasmissione e di generare la tempistica del trasferimento mentre lo slave è quello che riceve una richiesta. Entrambe le categorie appena descritte possono assumere il ruolo di trasmittente o ricevente. Le modalità di trasferimento dati sono le seguenti:

- A trasmette dati a B:
  - A (master) spedisce l'indirizzo di B (slave) sul bus;
  - A trasmette i dati a B;
  - A termina il trasferimento.

- A vuole ricevere dati da B:
  - A (master) spedisce l'indirizzo di B (slave) sul bus;
  - B spedisce i dati ad A;
  - A termina il trasferimento.

Un esempio di trasmissione completa utilizzando il protocollo I^2C è quella che compare nella seguente figura.

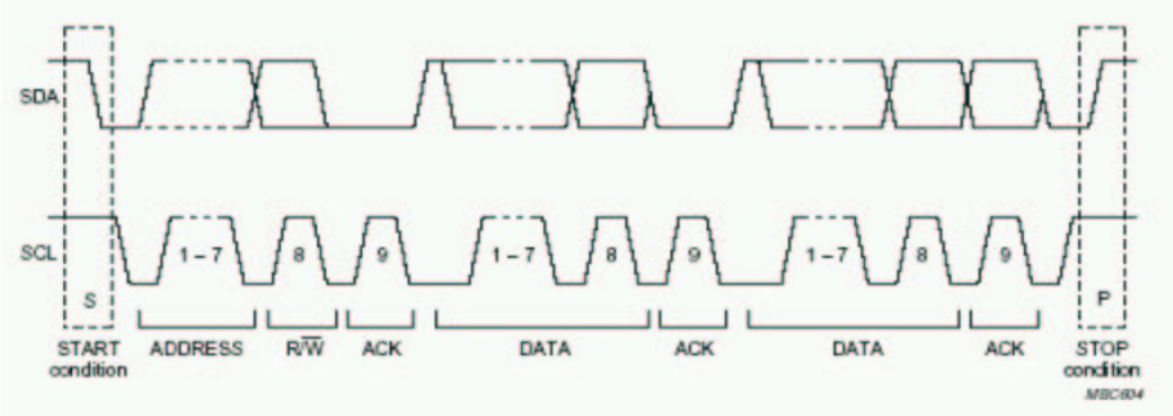

Figura B.1: Trasmissione tramite protocollo I2C con indirizzo lungo 7 bit

La trasmissione inizia con un segnale di start, immesso sulla linea dati dal master, dopo un controllo sull'occupazione del bus. La condizione di start consiste nel lasciare la linea SCL allo stato alto mentre la linea SDA subisce una transizione dallo stato 1 allo stato 0. Dopo la generazione del segnale di start, inizia la trasmissione dei dati: il primo byte trasmesso è quello composto dall'indirizzo dello slave con l'aggiunta di un ulteriore bit che indica al ricevente qual è l'operazione a lui richiesta (0: ricezione per la scrittura dati, 1: trasmissione per la lettura dati). L'ordine di trasmissione dei bit è quello dal più significativo al meno significativo.

Dopo la trasmissione di ogni byte, chi trasmette ha l'obbligo di lasciare la linea SDA allo stato alto, in modo da permettere a chi riceve di darne conferma tramite il meccanismo dell'*acknowledgement* (ACK): esso consiste nell'abbassare la linea SDA in corrispondenza del nono impulso presente sulla linea SCL. Il segnale ACK può essere generato in due casi:

- quando un dispositivo è inabilitato a ricevere (o perché sta eseguendo delle funzioni in real-time o perché non può più immagazzinare altri dati);
- quando chi ha iniziato la trasmissione vuole interrompere la comunicazione utilizzando il segnale di stop.

Il segnale di stop, come quello di start, è sempre generato dal dispositivo master. Esso consiste nel far variare la linea SDA dallo stato basso a quello alto in corrispondenza del periodo alto della linea SCL.

Durante la trasmissione avvengono contemporaneamente due processi: la *sincronizzazione dei clock* e l'*arbitraggio*.

Il primo processo permette a due dispositivi, con velocità di funzionamento diverse, di comunicare senza incorrere in perdite di dati. Il clock di un elemento è contraddistinto da un periodo alto, in cui assume il valore 1, e da uno basso, in cui assume il valore 0. Entrambi i valori devono essere mantenuti per un certo intervallo di tempo, il quale può essere diverso nei due dispositivi; un esempio di sincronizzazione è visibile in Fig. B.2. Nell'immagine sono rappresentati i segnali dei clock interni di due dispositivi (chiamati CLK1 e CLK2) che hanno periodi, sia alti che bassi, con durate differenti.

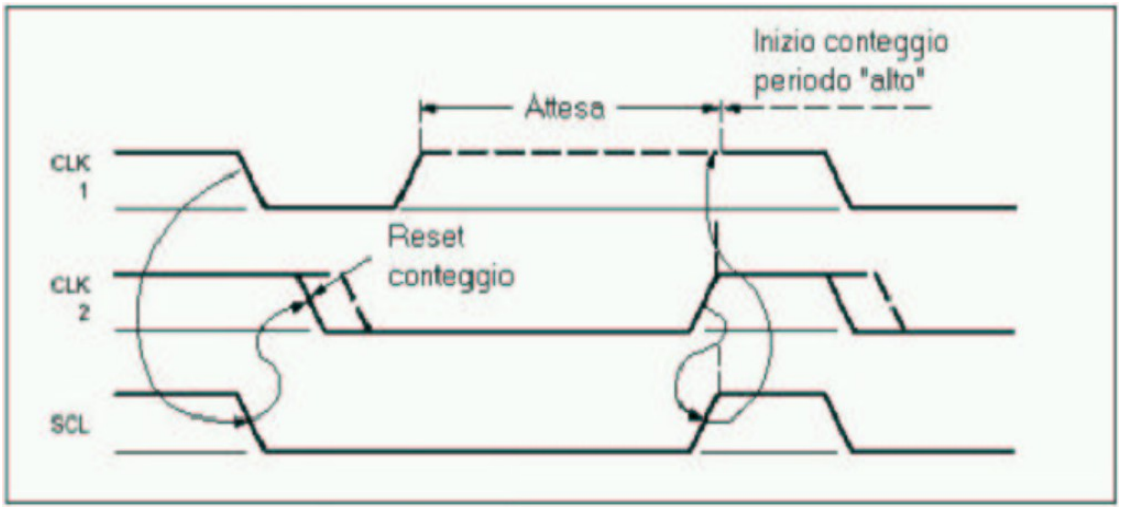

Figura B.2: Sincronizzazione dei clock

Nella figura si vede che entrambi i dispositivi lasciano, sulla linea SCL, un valore alto per la durata del periodo del fronte positivo del clock. Una volta esaurito questo intervallo di tempo, i due elementi mettono sulla linea SCL un valore 0, mantenendolo per il tempo di durata di questo periodo. Il fronte negativo di CLK2 risulta superiore a quello di CLK1, quindi sulla linea SCL avremo un valore negativo fino a che il secondo dispositivo non terminerà il periodo basso. La regola di sincronizzazione prevede che il primo elemento, rilevando sulla linea SCL un valore 0, debba attendere fino a che la linea non ritorni al valore 1. Una volta accaduto ciò, manterrà il valore 1 per la durata del periodo alto per poi passare al valore 0. Il processo continuerà sempre nello stesso modo. Usando questa tecnica il clock generato ha un periodo basso pari al più lungo periodo basso dei dispositivi, mentre il periodo alto è il più breve fra i periodi alti degli elementi collegati al bus.

L'*arbitraggio* consente di utilizzare il bus come un *multi-master*, ossia è possibile collegare fra loro più dispositivi con la facoltà di iniziare trasferimenti di dati, senza che avvengano perdite di informazione. Il processo consiste nel paragonare ciò che si trasmette con quello che effettivamente si trova sulla linea SDA. Quando due dispositivi trasmettono due valori differenti, quello prevalente risulta essere il valore basso. Nel caso in cui un dispositivo master trasmetta un livello 0 ed un altro trasmetta un livello 1, quest'ultimo dovrà disabilitare il suo stato d'uscita, poiché sulla linea vedrà un valore diverso dal suo. Nel caso in cui il dispositivo sia master e slave allo stesso tempo, è possibile che l'altro elemento lo stia contattando, quindi dovrà passare all'istante dallo stato master allo stato slave. La procedura d'arbitraggio permette a due elementi di iniziare entrambi la trasmissione e di continuarla fino a che il processo va a buon fine. In questo modo non si perdono dati. La seguente figura mostra un esempio d'arbitraggio fra due elementi.

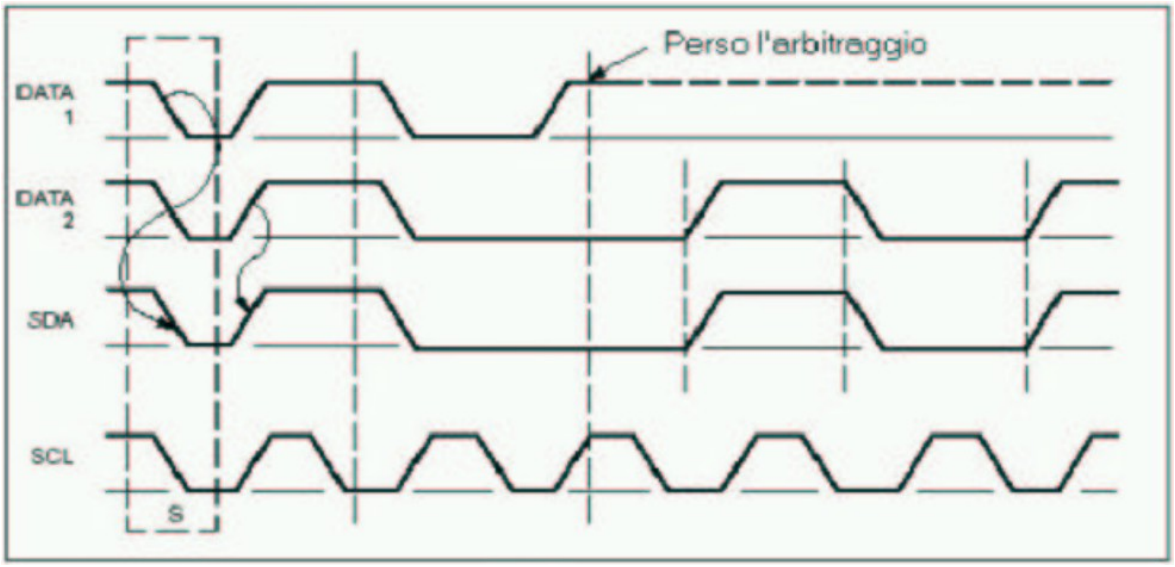

Figura B.3: Arbitraggio tra due elementi

Appendice C

C - Installazione del framework TDAQ su sistemi Linux

Per poter utilizzare il software *testpad* bisogna prima verificare la presenza dei seguenti requisiti:

- ambiente di programmazione TDAQ (*ayum*, *Java 1.7*, *CMT*, *gcc 4.8.1*);
- area di lavoro (determinata struttura di directory per lo sviluppo e l'utilizzo dei pacchetti e settaggio delle relative variabili di ambiente);
- driver Systec CAN installato e configurato;
- check-out dei pacchetti necessari al funzionamento del software.

C.1 Installazione dell'ambiente di programmazione TDAQ

Come prima cosa va scelta una *root location* per l'installazione del comando *ayum*:

```
> cd <inst_root>
> rpm -i --dbpath `pwd`/.rpmdb --prefix=`pwd` http://atlas-tdaq-
sw.web.cern.ch/atlas-tdaq-sw/yum/tdaq/slc6/x86_64/ayum_slc6-2-
0.x86_64.rpm
```

Successivamente creare l'alias per *ayum* (non necessario ma fortemente consigliato) ed installare l'ultima release di TDAQ (*tdaq-05-05-00* al momento della stesura di questa guida), *Java 1.7*, *CMT* e *gcc 4.8.1*:

```
> alias ayum=<inst_root>/ayum/ayum
> ayum update
> ayum install tdaq-05-05-00_DAQRelease_x86_64-slc6-gcc47-opt  tdaq-05-
05-00_databases jdk_1.7.0_x86_64_slc6
> ayum install CMTv1r25p20130606Linux-i686 gcc_4.8.1_x86_64-slc6 tdaq-05-
05-00_DAQRelease_src LCGCMT_LCGCMT_71_x86_64_slc6_gcc48_opt
> ayum clean all
```

C.2 Creazione di un'area di lavoro

Una "*Developer Working Area*" è un posto dove l'utente può fare il check-out di determinati pacchetti, modificarli, compilarli e farne il *commit*. In altre parole, quest'area è una "mini-release" contenente anche gli eseguibili installati. La struttura deve rispettare determinati canoni:

```
/home-users/user/detectors/muons/       <-- root CMT working area

/home-users/user/detectors/muons/cmt/project.cmt   <-- CMT 'project'
file, deve contenere un'unica riga con scritto 'use tdaq <release>' dove
<release> è la release utilizzata (ad esempio, tdaq-05-05-00)

/home-users/user/detectors/muons/installed   <-- qui verranno posizionati
tutti i file compilati
/home-users/user/detectors/muons/pkg/pkg-00-06-12/cmt <-- pacchetto
                                                  /src
                                    ....
```

Arrivati a questo punto bisogna settare le seguenti variabili di ambiente con il path dell'*installation area* (per comodità posizionare i seguenti comandi nel file *.bashrc*):

- *export PATH=$PATH:$HOME/detectors/muons/installed*
- *export TDAQ_CLASSPATH=$HOME/detectors/muons/installed*
- *export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: $HOME/detectors/muons/installed/i686-slc6-gcc48-opt/lib/*
- *export MAN_PATH=$HOME/detectors/muons/installed*
- *export IDLINCLUDE=$HOME/detectors/muons/installed*

La variabile **TDAQ_INST_PATH** è invece sempre settata alla directory della release:

export TDAQ_INST_PATH=$HOME/detectors/muons/cmt/

Un'area di lavoro è anche un progetto CMT e quindi va configurata anche la variabile **CMTPROJECTPATH** contenente il path di CMT su AFS:

export CMTPROJECTPATH=/afs/cern.ch/atlas/project/tdaq/inst

Infine è necessario settare anche le seguenti variabili che verranno utilizzate successivamente:

- *export TDAQ_DIR=/afs/cern.ch/atlas/project/tdaq/*
- *export TDAQ_VERSION=tdaq-05-05-00*
- *export RPC_INST_PATH=/det/muon/sw/$TDAQ_VERSION/installed*
- *export RPC_READOUT_CONFIG_DIR=/det/muon/RPC/data*
- *export RPC_LOGS=/tmp*

C.3 Installazione e configurazione del driver Systec CAN

Arrivati a questo punto bisogna installare il driver *Systec CAN* per poter far funzionare il relativo modulo. La versione consigliata è la seguente (da preferirsi alla più recente 0.9.0p-1): **Systec CAN 0.8.0p-6 SLC66 x86_64**.
Per l'installazione lanciare, con permessi di *root*, il seguente comando:

```
rpm -ivhH systec_can-0.8.0p-6.SLC66.x86_64.rpm
```

Una volta fatto ciò troveremo tutti gli script utili a gestire il modulo nella seguente directory: */opt/systec_can* . È importante, come prima cosa, lanciare il seguente script per poter generare il file di configurazione *udev*:

```
/opt/systec_can/experts/create_udev_rules.py
```

Successivamente bisogna caricare i seguenti moduli nel kernel:

```
modprobe can_dev
modprobe systec_can
modprobe can_raw
modprobe can_bcm
modprobe can
```

Adesso bisogna fare il *"ports up"* delle 16 porte CAN settandole a 250000 di bitrate (il seguente comando va lanciato con permessi di *root*):

```
/opt/systec_can/systec_ports_up.py 0 15 250000
```

Fatto ciò, per vedere se tutto è andato a buon fine, possiamo lanciare il seguente script:

```
/opt/systec_can/systec_status.py
```

L'output atteso deve essere il seguente:

```
Port          State                    | [green led]  [red led] |
can0          OK(certain)              |
can1          OK(certain)              |
can2          OK(certain)              |
can3          OK(certain)              |
can4          OK(certain)              |
can5          OK(certain)              |
can6          OK(certain)              |
can7          OK(certain)              |
can8          OK(certain)              |
can9          OK(certain)              |
can10         OK(certain)              |
can11         OK(certain)              |
can12         OK(certain)              |
can13         OK(certain)              |
can14         OK(certain)              |
can15         OK(certain)              |
```

C.4 Check-out e compilazione pacchetti

Siamo giunti alla fase conclusiva della configurazione, ossia quella di prelievo dei pacchetti necessari dal repository *SVN*. Il repository in questione è:

```
svn.cern.ch/reps/muondaq
```

ed i pacchetti, nell'ordine da prelevare, sono:

- *MUPolicy*
- *RPCi2ccan*
- *RPCgen_i2cprg*
- *RPCRxSl*
- *RPCRODModule*

- *RPCModules*
- *RPCRODUtils*
- *RPCDecoding*
- *RPCGnam*
- *RPClvl1Mon*

Può risultare utile definire la seguente variabile di ambiente:

export SVNROOT="svn+ssh://svn.cern.ch/reps/muondaq"

Questa può velocizzare le fasi di check-out senza dover ogni volta inserire il path completo del server SVN. Da ricordare che se il comando di check-out viene mandato da un utente generico non presente in *lxplus*, allora il comando di sopra diviene:

export SVNROOT="svn+ssh://utente_lxplus@svn.cern.ch/reps/muondaq"

Arrivati a questo punto ci posizioniamo nella root directory della nostra working area (nell'esempio del paragrafo B.2 *$HOME/detectors/muons/*) e lanciamo il comando di check-out:

```
svn co $SVNROOT/MUPolicy/trunk MUPolicy
```

(da ripetere per tutti i pacchetti elencati sopra).

Una volta prelevati, questi vanno compilati utilizzando il comando *cmt* (installato in precedenza, vedi B.1). Importante: se si sta configurando una macchina a 64 bit risulterà necessario installare le seguenti librerie a 32 bit:

```
sudo yum -y install --skip-broken glibc.i686 arts.i686 audiofile.i686
bzip2-libs.i686 cairo.i686 cyrus-sasl-lib.i686 dbus-libs.i686
directfb.i686 esound-libs.i686 fltk.i686 freeglut.i686 gtk2.i686 hal-
libs.i686 imlib.i686 lcms-libs.i686 lesstif.i686 libacl.i686 libao.i686
libattr.i686 libcap.i686 libdrm.i686 libexif.i686 libgnomecanvas.i686
libICE.i686 libieee1284.i686 libsigc++20.i686 libSM.i686 libtool-
ltdl.i686 libusb.i686 libwmf.i686 libwmf-lite.i686 libX11.i686
libXau.i686 libXaw.i686 libXcomposite.i686 libXdamage.i686 libXdmcp.i686
libXext.i686 libXfixes.i686 libxkbfile.i686 libxml2.i686 libXmu.i686
libXp.i686 libXpm.i686 libXScrnSaver.i686 libxslt.i686 libXt.i686
libXtst.i686 libXv.i686 libXxf86vm.i686 lzo.i686 mesa-libGL.i686 mesa-
libGLU.i686 nas-libs.i686 nss_ldap.i686 cdk.i686 openldap.i686 pam.i686
```

```
popt.i686 pulseaudio-libs.i686 sane-backends-libs-gphoto2.i686 sane-
backends-libs.i686 SDL.i686 svgalib.i686 unixODBC.i686 zlib.i686 compat-
expat1.i686 compat-libstdc++-33.i686 openal-soft.i686 alsa-oss-libs.i686
redhat-lsb.i686 alsa-plugins-pulseaudio.i686 alsa-plugins-oss.i686 alsa-
lib.i686 nspluginwrapper.i686 libXv.i686 libXScrnSaver.i686 qt.i686 qt-
x11.i686 pulseaudio-libs.i686 pulseaudio-libs-glib2.i686 alsa-plugins-
pulseaudio.i686

sudo yum -y install glibc-devel.i686 glibc-devel ibstdc++-devel.i686

sudo yum -y install libuuid.i686 libuuid-devel.i686 libuuid-devel.x86_64
openssl-devel.i686 openssl-devel.x86_64 freetype.i686 freetype.x86_64

sudo yum install pam.i686
```

Una volta fatto ciò possiamo proseguire con la compilazione (nell'ordine visto sopra) dei pacchetti scaricati. Per ognuno di essi ci posizioniamo nella cartella **cmt** (ad esempio, per il pacchetto MUPolicy, *$HOME/detectors/muons/MUPolicy/cmt*) e lanciamo i seguenti comandi:

```
cmt config
cmt make
cmt make inst
```

(se qualche pacchetto è stato installato precedentemente, far precedere ai comandi di sopra cmt make clean). Fatto ciò avremo nella directory

$HOME/detectors/muons/installed/i686-slc6-gcc48-opt/bin

tutti i file binari compilati (tra cui *testpad*, *programPad* e *testInitFile*). Può risultare utile, per concludere, creare un *alias* a *testpad*:

```
alias testpad=$HOME/detectors/muons/installed/i686-slc6-gcc48-
opt/bin/testpad
```

Bibliografia

1. CERN website, http://home.cern/
2. ATLAS Experiment website, http://atlas.ch/
3. Oracle, “Java Platform Standard Edition 7 Documentation”, http://docs.oracle.com/javase/7/docs/
4. ATLAS Experiment, “DAQ/HLT-I S/W releases building, usage, installation and patching procedures”, https://twiki.cern.ch/twiki/bin/viewauth/Atlas/TDAQReleaseBuilding
5. INFN Sezione di Napoli website, http://www.na.infn.it/it/
6. Linux @ CERN, http://linux.web.cern.ch/linux/
7. Systec Electronic, USB-CANmodul16 website, http://www.systec-electronic.com/en/products/industrial-communication/interfaces-and-gateways/usb-can-interface-usb-canmodul16
8. CMS Experiment website, http://cms.web.cern.ch/
9. LHCb Experiment website, http://lhcb.web.cern.ch/lhcb/
10. ALICE Experiment website, https://alice-collaboration.web.cern.ch/
11. ReCaS website, http://www.pon-recas.it/
12. INFN Sezione di Napoli e Roma, ATLAS Barrel Level-1 Muon Trigger website, http://maclabe.roma1.infn.it/muonl1/system/
13. Agilent Technologies, “Small Form Factor MT-RJ Fiber Optic Transceivers for Gigabit Ethernet”, http://maclabe.roma1.infn.it/muonl1/system/comp_datasheets/HFBR-5912E.pdf
14. Agilent Technologies, “Agilent HDMP-1032A/1034A Transmitter/Receiver Chip Set Data Sheet”, http://maclabe.roma1.infn.it/muonl1/system/comp_datasheets/HDMP-1032A.pdf

15. V. Izzo, ALLDIGITALL website, http://people.na.infn.it/~izzo/Alldigitall/Home.html
16. ATLAS Experiment, “ATLAS TDAQ Software CMT development page: version 3.1 (feb. 2014)”, http://atlas-tdaq-sw.web.cern.ch/atlas-tdaq-sw/cmt.html
17. LHCb Experiment, CMT website, http://lhcb-comp.web.cern.ch/lhcb-comp/Support/CMT/cmt.htm
18. National Instruments, “The Basics of CANopen”, http://www.ni.com/white-paper/14162/en/