Indice

1 INTRODUZIONE 10

1.1 Classificazione dei segnali analogici 10

1.2 Rappresentazione dei segnali analogici nei sistemi digitali 11

1.2.1 Campionamento 12

1.2.2 Quantizzazione 14

1.3 L'interferometro laser 14

1.4 Presentazione del lavoro svolto 16

1.4.1 Costituenti del sistema digitale 17

1.4.2 Comunicazione tra le parti del sistema 19

1.4.3 Interfaccia di comando su rete 20

2 SISTEMA OPERATIVO 21

2.1 Introduzione ai sistemi operativi 21

2.1.1 Sistemi "mainframe" 21

2.1.2 Sistemi da scrivania 22

2.1.3 Sistemi con più unità d'elaborazione 22

2.1.4 Sistemi distribuiti 22

2.1.5 Sistemi palmari 22

2.1.6 Sistemi d'elaborazione in tempo reale 23

2.2 Ambienti d'elaborazione embedded 25

2.3 Il sistema operativo LynxOS 25

2.3.1 Gestione della memoria 27

2.3.2 Strumenti di sincronizzazione a livello del kernel 29

2.3.3 Gestione degli interrupt 30

2.3.4 Thread a livello del kernel 30

3 IL BUS VME 32

3.1 Generalità dei bus 32
3.2 Storia del VMEBus 34
3.2.1 VMEBus Standard 35
3.2.2 VME64 36
3.2.3 Le evoluzioni di VME: VME64x e VME 320 37
3.3 Principali caratteristiche del VMEBus Standard 39
3.4 Gestione degli interrupt del VMEBus Standard 42
3.5 Arbitraggio del VMEBus Standard 44
3.6 Sommario delle caratteristiche 45
4 DISPOSITIVI UTILIZZATI: ADC E DAC 47
4.1 Dispositivo MPV955 47
4.2 Impostazioni hardware del dispositivo MPV955 48
4.3 Impostazioni software del dispositivo MPV955 50
4.3.1 Organizzazione della memoria interna 51
4.3.2 Registro "Control/Status" 52
4.3.3 Registri "Start/Stop Address" 53
4.3.4 Registro "Interrupt Control" 54
4.3.5 Registro "Rate Timer Control" 55
4.3.6 Registro "Timeout Control" 55
4.3.7 Registro "DAC Disable" 56
4.4 Esempio d'uso del dispositivo MPV955 56
4.5 Dispositivo VGD5 57
5 INGEGNERIZZAZIONE DEL SISTEMA 58
5.1 Paradigma di ingegneria del software 58
5.2 Raccolta dei requisiti 60
5.2.1 Requisiti funzionali 60
5.2.2 Requisiti di interfaccia 61
5.2.3 Requisiti di sicurezza 62
5.2.4 Requisiti prestazionali 62
5.2.5 Requisiti implementativi o di contesto 62
5.2.6 Requisiti particolari di test e collaudo 63
5.3 Analisi architetturale e progettazione del driver 63
5.3.1 Collocazione operativa 64
5.3.2 Interfacce esterne 64

5.3.3 Moduli del driver ... *64*
5.3.4 Interfacce interne ... *72*
5.4 Analisi architetturale e progettazione sistema di trasmissione ... 72
5.4.1 Architettura del sistema di comunicazione ... *72*
5.4.2 Interfacce interne comuni ... *73*
5.4.3 Moduli dell'unità di calcolo ... *73*
5.4.4 Moduli dell'unità di controllo ... *74*
5.5 Analisi architetturale e progettazione dell'interfaccia di comando ... 75
5.5.1 Architettura del sistema di interfaccia ... *75*
5.5.2 Interfacce interne comuni ai programmi demoni ... *76*
5.5.3 Moduli del programma demone lato unità di calcolo ... *77*
5.5.4 Moduli del programma demone lato unità di controllo ... *79*
5.5.5 Struttura interfaccia di comando ... *79*
5.5.6 Interfaccia grafica (GUI) ... *82*
5.5.7 Interfaccia a linea di comando ... *84*
5.5.8 Funzioni offerte dall'interfaccia di comando ... *85*
5.6 Piano di testing del sistema ... 85
5.7 Configurazione ... 86
5.8 Norme per l'implementazione ... 86
5.9 Matrice di copertura dei requisiti ... 87

6 SVILUPPO DEL DRIVER ... 88

6.1 Scrivere un driver ... 88
6.2 Struttura dei driver per LynxOS ... 90
6.2.1 Entry Point ... *92*
6.2.2 Struttura delle informazioni del dispositivo ... *93*
6.2.3 Struttura statica ... *93*
6.3 Analisi del codice ... 94
6.3.1 mpv955_install() ... *94*
6.3.2 mpv955_uninstall() ... *95*
6.3.3 mpv955_open() ... *96*
6.3.4 mpv955_close() ... *97*
6.3.5 mpv955_write() ... *97*
6.3.6 mpv955_ioctl() ... *98*
6.4 Installazione del driver nel sistema operativo ... 98
6.4.1 Installazione dinamica ... *99*

6.4.2 Installazione statica *100*
6.5 COLLAUDO 101
6.5.1 Primo test *102*
6.5.2 Secondo test *102*
6.5.3 Terzo test *103*
7 SISTEMA DI TRASMISSIONE DEI DATI 104
7.1 MODELLI DI RIFERIMENTO DELLE RETI 104
7.1.1 Il modello di riferimento OSI *104*
7.1.2 Il modello TCP/IP *105*
7.2 PROTOCOLLI OFFERTI DALLO STRATO TRASPORTO DI TCP/IP 107
7.2.1 Il protocollo di trasporto TCP *108*
7.2.2 Il protocollo di trasporto UDP *111*
7.2.3 Protocollo scelto per il nostro sistema *111*
7.3 STRUTTURA DEI PROGRAMMI DI COMUNICAZIONE 112
7.3.1 Prima soluzione *113*
7.3.2 Seconda soluzione *114*
7.3.3 Terza soluzione *114*
7.3.4 Quarta soluzione *114*
7.3.5 Soluzione adottata per l'implementazione del sistema *115*
7.4 DESCRIZIONE DELLA SOLUZIONE SCELTA 116
7.4.1 Interfacce interne comuni *116*
7.4.2 Moduli per l'unità di calcolo *119*
7.4.3 Moduli per l'unità di controllo *121*
8 CONTROLLO REMOTO DEL SISTEMA 124
8.1 STRUTTURA DEL SISTEMA DI INTERFACCIA 124
8.2 PROGRAMMI DEMONI 125
8.3 LINGUAGGIO DI PROGRAMMAZIONE JAVA 126
8.3.1 Programmazione orientata agli oggetti *126*
8.3.2 Caratteristiche del linguaggio Java *129*
8.3.3 Unified Modeling Language: UML *131*
8.3.4 La libreria grafica Swing *133*
8.4 INTERFACCIA DI COMANDO 136
8.4.1 Comunicazione mediante socket TCP *138*
8.4.2 Pacchetto CdafCtrl *141*

8.4.3 Pacchetto InterfacciaGrafica *143*
8.4.4 Pacchetto InterfacciaTestuale *146*
8.5 TEST DELL'INTERFACCIA DI COMANDO 147
9 CONCLUSIONI 149
BIBLIOGRAFIA 152

Elenco delle figure

FIGURA 1: ESEMPIO DI UN SEGNALE ANALOGICO PERIODICO ... 12
FIGURA 2: ESEMPIO DI UN SEGNALE PERIODICO CAMPIONATO ... 12
FIGURA 3: SEGNALE A BANDA LIMITATA DA B HZ ... 13
FIGURA 4: ESEMPIO DI SEGNALE DIGITALE ... 14
FIGURA 5: STRUTTURA DI UN INTERFEROMETRO LASER ... 15
FIGURA 6: COSTITUENTI DEL SISTEMA DIGITALE ... 17
FIGURA 7: IL KERNEL HA ACCESSO ALLO SPAZIO D'INDIRIZZAMENTO DEI PROCESSI ... 26
FIGURA 8: MAPPA DELLA MEMORIA VIRTUALE DI LYNXOS SU UN PROCESSORE POWERPC ... 27
FIGURA 9: LA MEMORIA VIRTUALE VIENE PARTIZIONATA IN PIÙ PAGINE DI MEMORIA FISICA ... 28
FIGURA 10: GESTIONE INTERRUPT CON I KERNEL THREAD ... 31
FIGURA 11: SEMPLICE STRUTTURA A BUS ... 32
FIGURA 12: SUDDIVISIONE LOGICA DELLE LINEE DEL BUS ... 33
FIGURA 13: IL VMEBUS HA LA FORMA DI UN CESTELLO (CRATE) ... 35
FIGURA 14: FORMATO 6U E 3U ... 36
FIGURA 15: ASSEGNAZIONE PIN PER IL CONNETTORE P1 ... 37
FIGURA 16: ASSEGNAZIONE PIN PER IL CONNETTORE P2 ... 38
FIGURA 17: CODICI DEGLI ADDRESS MODIFIER ... 40
FIGURA 18: TIPICO CICLO DI LETTURA ... 41
FIGURA 19: SISTEMA DI INTERRUPT ... 43
FIGURA 20: TIPICO CICLO DI INTERRUPT ... 44
FIGURA 21: POSIZIONE DEI JUMPER SUL DISPOSITIVO MPV955 ... 49
FIGURA 22: ORGANIZZAZIONE DELLA MEMORIA ... 50
FIGURA 23: CONTENUTO DELLA MEMORIA DI CONTROLLO ... 52
FIGURA 24: BIT COMPONENTI IL REGISTRO CONTROL/STATUS ... 52
FIGURA 25: STRUTTURA DEL REGISTRO PER GLI INTERRUPT ... 54
FIGURA 26: LE FASI DI UN CICLO DI RISOLUZIONE ... 58
FIGURA 27: IL MODELLO INCREMENTALE ... 59

Figura 28: Collocazione operativa del driver ... 63
Figura 29: Diagramma dei componenti e dei nodi ... 72
Figura 30: Struttura interfaccia di comando ... 76
Figura 31: Diagramma delle classi del pacchetto CdafCtrl ... 80
Figura 32: Diagramma di stato per la classe ControlCdaf ... 81
Figura 33: Diagramma di sequenza relativo al metodo sendStart() ... 82
Figura 34: Diagramma delle classi del pacchetto InterfacciaGrafica ... 83
Figura 35: Diagramma delle classi del pacchetto InterfacciaTestuale ... 84
Figura 36: Collocazione di un driver ... 89
Figura 37: Listato parziale della directory "/dev" ... 90
Figura 38: Componenti di un driver ... 91
Figura 39: Traduzione delle chiamate di sistema ... 92
Figura 40: Primo test del driver ... 101
Figura 41: Secondo test del driver ... 102
Figura 42: Terzo test del driver ... 103
Figura 43: Contenuto di un pacchetto TCP ... 108
Figura 44: Procedura three-way handshake ... 110
Figura 45: Contenuto di un pacchetto UDP ... 111
Figura 46: Test del sistema di acquisizione ... 115
Figura 47: Struttura del sistema di comando ... 125
Figura 48: Panoramica della libreria Swing di Java 2 ... 134
Figura 49: Finestra principale dell'interfaccia grafica sotto Linux ... 137
Figura 50: Soluzione che utilizza JNI ... 138
Figura 51: Diagramma della classe StringConverter ... 140
Figura 52: Diagramma della classe Sender ... 142
Figura 53: Diagramma della classe StartCdaf ... 143
Figura 54: Diagramma della classe ConsoleReader ... 145
Figura 55: Menu principale in modalità testuale ... 146
Figura 56: Finestra principale dell'interfaccia grafica sotto Windows ... 147

Prefazione

I sistemi di controllo elettronici, generalmente progettati per una determinata applicazione, sono identificati con il termine "embedded". Le caratteristiche di precisione di questi sistemi, abbinate con l'affidabilità dei sistemi operativi real-time hanno conquistato il mercato dei sistemi digitali. Contrariamente ai calcolatori generici, un sistema embedded ha dei compiti conosciuti già durante lo sviluppo, che svolgerà nel modo più opportuno possibile grazie ad un incastro hardware/software specificamente pensato per la tale applicazione.

Nasce di conseguenza il problema di interfacciare il sistema digitale con il mondo esterno, il più delle volte caratterizzato da grandezze analogiche non rappresentabili (se non discretamente) all'interno di un dispositivo elettronico.

In quest'area si colloca il lavoro esposto in questa tesi, diviso nelle seguenti sezioni:

- "Introduzione": presentazione del problema affrontato, introduzione allo sviluppo di driver e ai protocolli di comunicazione tra le parti del sistema. È presentato il problema di gestire l'intero sistema tramite interfaccia di comando e infine è discusso il problema della conversione analogico-digitale.
- "Sistema Operativo": panoramica dei vari tipi di sistemi con relativi sistemi operativi. In particolari sono trattati i sistemi operativi utilizzati in ambito embedded.
- "Il bus VME": architettura e caratteristiche tecniche del VMEBus, molto utilizzato nei sistemi integrati.
- "Dispositivi utilizzati: ADC e DAC": descrizione dei dispositivi utilizzati per interfacciare il nostro sistema digitale.

- “Ingegnerizzazione del sistema”: raccolta dei requisiti, specifica tecnica, e tutta l’ingegnerizzazione del sistema sviluppato.
- “Sviluppo del driver”: ruolo dei driver all’interno dei sistema operativo, scrittura degli stessi, implementazione e collaudo dei driver necessari al nostro sistema.
- “Sistema di trasmissione dei dati”: implementazione e collaudo del sistema di trasmissione dati tra le unità del sistema con presentazione dei protocolli di comunicazione usati.
- “Controllo remoto del sistema”: gestione dell’intero sistema tramite interfaccia grafica di comando.

L’attività è stata svolta mettendo a disposizione le conoscenze informatiche apprese nel corso degli studi, e, di conseguenza, riguarda per lo più lo sviluppo d’applicazioni software.

Capitolo 1

Introduzione

L'automazione è un complesso di tecniche volte a migliorare l'efficienza dell'intervento umano, o addirittura a sostituirlo, nell'esercizio di dispositivi e impianti. Negli ultimi tempi l'automazione ha ricevuto un notevole impulso, legato ai progressi fatti nel campo degli elaboratori elettronici.
Parte della scienza dell'automazione tratta i sistemi di controllo elettronici. Oggetto di questo primo capitolo è l'introduzione al sistema di controllo realizzato.

1.1 Classificazione dei segnali analogici

Le coordinate astronomiche, il suono, grandezze fisiche come l'intensità della luce, sono tutte grandezze analogiche rappresentabili mediante *segnali analogici temporali*. La "Teoria dei segnali" studia le proprietà matematiche e statistiche dei segnali, definiti come funzioni matematiche $y = f(t)$ dove il tempo è rappresentato dalla variabile t, e y è il corrispondente valore di un'opportuna grandezza. Oggetto della "Teoria dei segnali" è anche lo studio degli effetti della trasmissione dei segnali attraverso un canale di comunicazione e la degradazione che questo subisce nel passaggio, la misura del loro contenuto informativo e le possibili trasformazioni che questo può subire.

I segnali vengono classificati in varie categorie, a seconda delle loro proprietà. Con riferimento al tempo si definisce:

- Segnale a tempo continuo: la variabile *t* può assumere un qualsiasi valore reale;
- Segnale a tempo discreto: la variabile *t* assume valori appartenenti ad un sottoinsieme discreto dei numeri reali.

Con riferimento alla variabile dipendente *y* si distinguono:

- Segnale ad ampiezza continua: la variabile *y* può assumere qualsiasi valore reale;
- Segnale ad ampiezza quantizzata: i valori assunti dalla variabile *y* valori appartengono ad un sottoinsieme discreto dei numeri reali.

Da queste distinzioni si definiscono:

- Segnale Analogico: segnale a tempo continuo e ad ampiezza continua;
- Segnale Digitale (o Numerico): segnale a tempo discreto e ad ampiezza quantizzata.

1.2 Rappresentazione dei segnali analogici nei sistemi digitali

Per la sua flessibilità nelle applicazioni, la tecnologia digitale è oggigiorno usata per l'elaborazione dei segnali analogici. Nasce immediatamente un problema specifico, cioè quello di interfacciare l'ambiente digitale con il mondo analogico. Purtroppo essi sono distanti tra loro: un elaboratore lavora su una scala di tempi discreta, e i valori che può assumere una grandezza trattata digitalmente sono finiti. Segue che convertire un segnale analogico in uno digitale comporta di per sé una perdita d'informazione.
Il processo di conversione analogico digitale consiste essenzialmente in due fasi distinte: il campionamento e la quantizzazione.

1.2.1 Campionamento

Partendo dal segnale analogico il campionamento consente di ottenere un segnale a tempo discreto, cioè una sequenza di numeri *x[n]*, rappresentabile con una funzione di variabile intera, avente valori reali o complessi.
Campionare un segnale *x(t)* significa "estrarre" dal segnale stesso i valori che esso assume a istanti temporali equispaziati, cioè multipli di un intervallo *T* detto periodo di campionamento. Così facendo si crea una sequenza il cui valore n-esimo *x[n]* è il valore assunto dal segnale a tempo continuo all'istante *nT*.

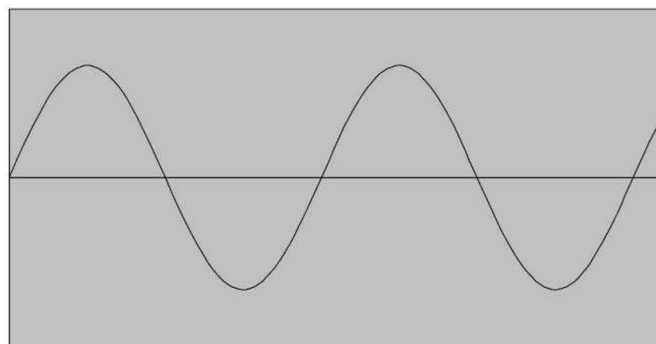

Figura 1: Esempio di un segnale analogico periodico

Un generico segnale analogico periodico è riportato in Figura 1. Per periodico s'intende un segnale per cui vale $x(t + \tau) = x(t)$ con τ periodo del segnale.

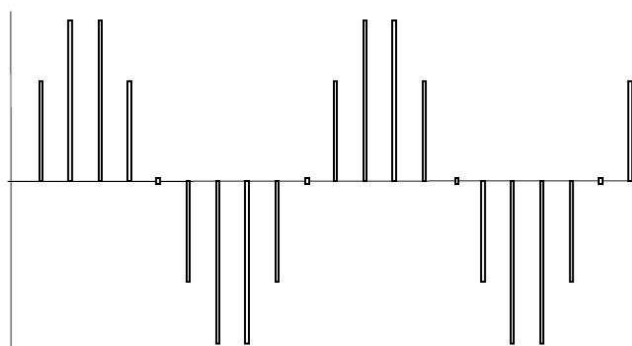

Figura 2: Esempio di un segnale periodico campionato

Dopo il campionamento il segnale si presenta come in Figura 2. Si vede limpidamente che abbiamo ottenuto una sequenza discreta di valori che distano tra loro un tempo *T* pari al periodo di campionamento.

Teorema di Shannon

A questo punto ci si chiede quale frequenza usare per campionare un segnale. Per i segnali a banda limitata ci viene in aiuto il teorema di Shannon, che fornisce la condizione necessaria affinché un segnale dopo il campionamento possa nuovamente essere ritrasformato in analogico ottenendo il segnale di partenza.

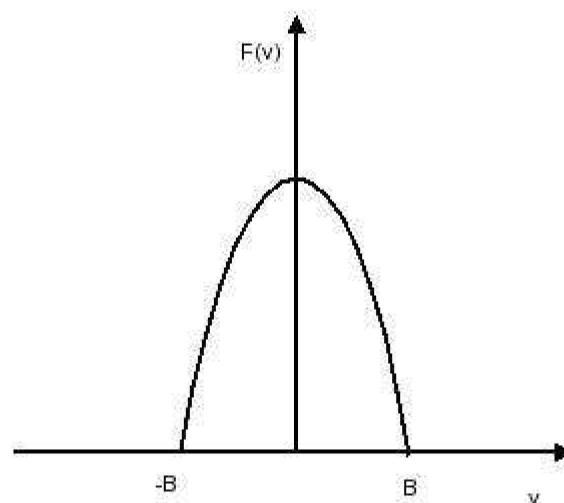

Figura 3: Segnale a banda limitata da B Hz

Il teorema afferma che:
"*un segnale a banda limitata da B Hz* (come quello in Figura 3)*, è ricostruibile univocamente dai suoi campioni purché essi siano campionati con una frequenza* f_c *almeno doppia di quella massima presente nel segnale:* $f_c > 2B$ ".

1.2.2 Quantizzazione

A questo punto, dopo il processo di campionamento, il segnale non è ancora digitale: per memorizzarlo all'interno di un sistema elettronico, le informazioni in esso contenute devono essere convertite in un numeri binari ad *N* bit. Tutti ben sanno che con una parola digitale a *N* bit si possono esprimere 2^N valori diversi, quindi ponendo *N=16* (che è la risoluzione dei dispositivi del nostro sistema) i valori che il segnale può assumere dopo la codifica sono 65535. Segue che ogni combinazione di bit in uscita rappresenta necessariamente tutto un intervallo di valori d'ingresso; l'ampiezza di tale intervallo si dice "intervallo di quantizzazione". L'errore sulla cifra meno significativa rappresenta l'incertezza sul valore di ingresso che ha determinato il dato in uscita; non può essere ridotta se non aumentando la risoluzione dei dati, ossia il numero di bit.
Dopo aver quantizzato il segnale, esso è in formato digitale e si presenta come una sequenza di gradini (Figura 4).

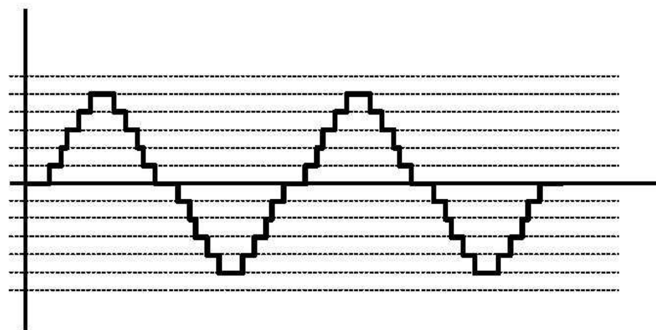

Figura 4: Esempio di segnale digitale

1.3 L'interferometro laser

La forza di gravità è tra tutte le forze della Natura quella che conosciamo da più tempo, ma tuttora disponiamo di pochissimi elementi sulle sue proprietà. Una delle proprietà più elusive della gravitazione, come descritta da Einstein nella Teoria Generale della Relatività, sono le onde gravitazionali, grossomodo delle variazioni del campo

gravitazionale (o meglio dello spazio-tempo) che si propagano alla velocità della luce. Finora non c'è un'osservazione diretta delle onde gravitazionali, ma solo delle prove indirette della loro esistenza. L'esperimento Virgo punta a rilevare le onde gravitazionali mediante l'uso di un interferometro Michelson a laser: una delle grandi sfide della fisica sperimentale.

Per l'altissima sensibilità richiesta, la lunghezza dei bracci deve essere di molti chilometri. Questo non può essere facilmente realizzato sulla Terra (in futuro si prevede la messa in orbita di particolari modelli di interferometri) e si usano riflessioni multiple per aumentare artificialmente la lunghezza dei bracci.

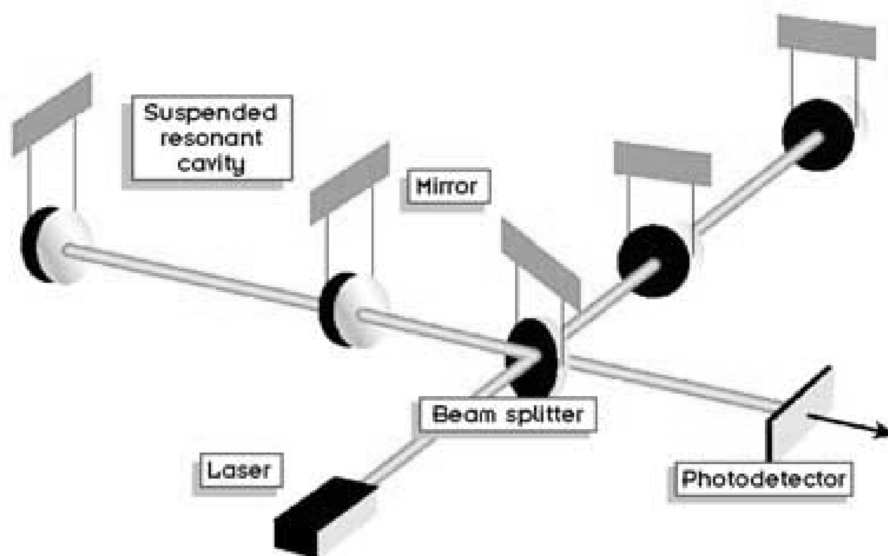

Figura 5: Struttura di un interferometro laser

Virgo è un interferometro laser di tipo Michelson con due bracci di 3 km disposti ad angolo retto. Uno specchio semitrasparente divide il fascio laser incidente in due componenti uguali mandate nei due bracci dell'interferometro. In ciascun braccio una cavità risonante Fabry-Perot formata da due specchi estende la lunghezza ottica da 3 a circa 100 chilometri per via delle riflessioni multiple della luce e pertanto amplifica la piccola variazione di distanza causata dal passaggio dell'onda gravitazionale. I due fasci di

luce laser, provenienti dai due bracci, vengono ricombinati in opposizione di fase su un rivelatore di luce in maniera che, normalmente, non arrivi luce sul rivelatore. La variazione del cammino ottico, causata dalla variazione della distanza tra gli specchi indotta dall'onda gravitazionale, produce un piccolissimo sfasamento tra i fasci e quindi un'alterazione dell'intensità luminosa osservata, proporzionale all'ampiezza dell'onda gravitazionale.
Gli specchi dell'interferometro sono sollevati e isolati dal suolo tramite un complesso sistema di sospensioni con lo scopo di isolarli quanto più è possibile dalle vibrazioni, e c'è bisogno di un sistema elettromeccanico che li controlli automaticamente.
Questo sistema già esiste in Virgo ed è gestito da delle DSP (Date Signal Processors) che forniscono un controllo del sistema con un loop chiuso a 20kHz.

1.4 Presentazione del lavoro svolto

Si è progettato e realizzato un sistema digitale in tempo reale stretto capace di acquisire segnali elettrici analogici, elaborarli in maniera digitale, e rimandarli in uscita nuovamente in forma analogica. La novità rispetto al metodo DSP è quella di spostare la parte di calcolo dei filtri, che può essere molto onerosa, su un computer o su una rete di computer "remoti" connessi mediante un canale di comunicazione, al prezzo di una minore frequenza del loop, ma con una maggiore capacità di calcolo. Questo sistema potrà essere utilizzato in futuro per il controllo automatico degli specchi di un interferometro laser come quello dell'esperimento VIRGO.
La progettazione del sistema di controllo elettronico ha mirato ad avere efficienza, affidabilità, e soprattutto ad essere privo d'errori, dovuti prevalentemente alla conversione analogico-digitale e viceversa. In effetti, al momento dell'acquisizione, ulteriori segnali di "disturbo" si aggiungono al segnale originale, inquinando così l'informazione alla quale siamo interessati. È necessario applicare al segnale acquisito dei *filtri digitali* (talvolta detti *filtri numerici*) in modo da ridurre, o addirittura eliminare, la parte rumorosa acquisita.
La sequenza d'operazioni che deve svolgere il sistema implementato, è dunque:

- Acquisizione del segnale analogico mediante dispositivi ADC (analog-digital converter);
- Elaborazione del segnale con filtri digitali;
- Attuazione del segnale elaborato mediante dispositivi DAC (digital-analog converter).

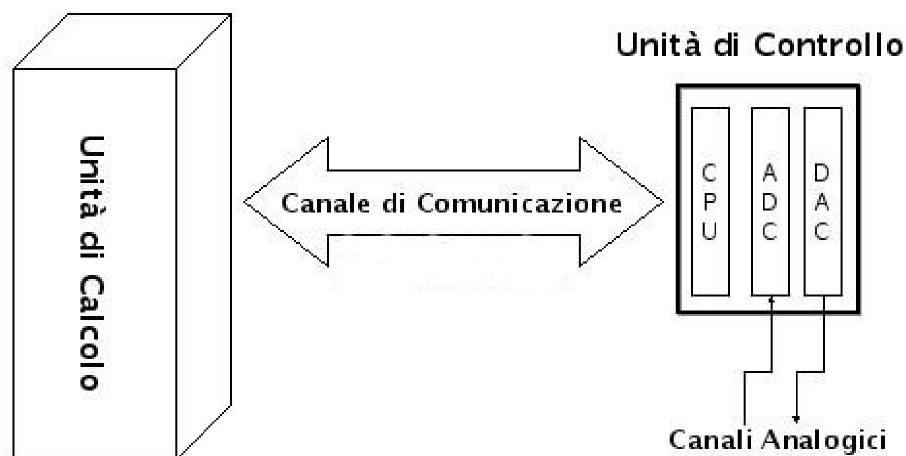

Figura 6: Costituenti del sistema digitale

1.4.1 Costituenti del sistema digitale

L'intero sistema d'acquisizione, elaborazione e attuazione dei dati è diviso principalmente in due moduli connessi tra loro mediante un canale di comunicazione:

- Unità di controllo;
- Unità di calcolo.

Uno schema raffigurante i costituenti del sistema è mostrato in Figura 6.

Unità di controllo

L'unità di controllo è costituita da un elaboratore con una CPU con architettura di tipo RISC (PowerPC), da un dispositivo ADC a 32 canali, e fino a quattro dispositivi DAC da 8 canali. Tutti i dispositivi sono alloggiati in un cestello e connessi mediante bus "VME Standard", presentato nel terzo capitolo.

Il sistema operativo scelto per l'unità di controllo è LynxOS: un ottimo sistema di tipo UNIX in tempo reale stretto. L'introduzione a questo sistema operativo, e il confronto con gli altri sistemi operativi in tempo reale stretto sarà oggetto del secondo capitolo.

L'unità di controllo deve acquisire i dati dal dispositivo ADC, inviarli all'unità di calcolo per la loro elaborazione, e ritrasmetterli sotto forma di segnali elettrici mediante i dispositivi DAC. L'acquisizione dei dati, del sistema, dovrà avvenire con frequenze di campionamento fino a 20KHz, e il tempo d'andata e ritorno (round-trip time) dei dati dalla lettura dell'ADC alle effettive uscite delle tensioni dal DAC, dovrà essere inferiore a 250µs. Inoltre le tensioni di uscita del DAC dovranno avere lo stesso ordinamento temporale dei dati corrispondenti dell'ADC.

All'inizio del lavoro svolto eravamo in possesso solo del driver per il dispositivo ADC, e non di quello per il DAC. Per usare quest'ultimo dispositivo in modo conveniente, è stato necessario scrivere un driver adatto a sfruttare tutte le sue capacità; i dettagli hardware del dispositivo DAC sono illustrati nel quarto capitolo, mentre la progettazione e realizzazione del driver è descritta nel sesto capitolo.

Tutti i programmi e i driver scritti per questa unità, sono stati sviluppati in linguaggio ANSI C, rispettano le specifiche POSIX 1003.1, .1b e .1c, e supportano le schede della Thales Computer VMPC6a.

Unità di calcolo

L'unità di calcolo è un generico calcolatore, o un insieme di generici calcolatori interconnessi, esterno all'unità di controllo e capace di eseguire programma scritto in linguaggio ANSI C, che rispetta le specifiche POSIX 1003.1, .1b e .1c. Il compito dell'unità di calcolo è:

- Ricevere i segnali in formato digitale dall'unità di controllo;
- Applicare un filtro digitale ai dati ricevuti;
- Ritrasmettere i dati filtrati all'unità di controllo per l'attuazione.

Canale di comunicazione

Il canale di comunicazione deve essere comune ad entrambe le unità. Durante il lavoro svolto sono state usate due schede di rete a 100Mbps collegate tra loro mediante un comune cavo di rete di categoria 5e.

Sviluppi futuri del sistema prevedono una connessione tra i due computer mediante una coppia di schede "Reflective Memory", progettate per comunicare tra loro ad alta velocità sfruttando la fibra ottica.

1.4.2 Comunicazione tra le parti del sistema

Allo scopo di far comunicare processi diversi in esecuzione su diverse macchine, viene adottata la programmazione di rete. Considerato l'hardware a disposizione adottiamo per la comunicazione il più comune stack di protocolli per la comunicazione in rete: il TCP/IP, supportato da entrambi i sistemi operativi installati nell'unità di controllo e quella di calcolo.

Lo stack di protocolli TCP/IP è ampiamente descritto nel settimo capitolo, per adesso ci interessa sapere solo che esso è organizzato in quattro livelli, ognuno con un compito ben specifico. Un livello particolarmente importante è quello di trasporto al quale appartengono i protocolli TCP e UDP. A questo livello si regola il flusso delle informazioni, e, secondo il protocollo scelto, si può avere un trasporto affidabile delle informazioni orientato alla connessione con il controllo degli errori (TCP), oppure un trasporto non affidabile non orientato alla connessione ma molto più rapido (UDP).

Nel caso della comunicazione tra l'unità di calcolo e l'unità di controllo, è stato utilizzato il protocollo UDP poiché la velocità di trasmissione delle informazioni è uno degli obiettivi principali del nostro lavoro. Essendo la connessione tra le due unità di tipo punto-punto e

la perdita di pacchetti è in pratica inesistente, quindi non abbiamo bisogno di un protocollo di comunicazione come TCP con un rigido controllo sugli errori.
Sempre inseguendo la velocità dell'intero sistema, e utilizzando le socket (in ambiente UNIX le socket rappresentano in sostanza un canale di comunicazione), le applicazioni su ambedue le unità sono state sviluppate seguendo varie soluzioni delle quali è stata scelta la più conveniente sulla base dei test effettuati. Lo sviluppo, e i test delle soluzioni appena proposte, è trattato in dettaglio nel settimo capitolo.

1.4.3 Interfaccia di comando su rete

L'unità di calcolo e quella di controllo comunicano tra loro mediante i processi appena descritti. Ogni qualvolta si voglia avviare il processo di acquisizione elaborazione e attuazione dati, è necessario mandare in esecuzione i suddetti processi impartendo manualmente i comandi su diversi terminali. Ciò risulta molto scomodo, quindi, si è pensato di introdurre un modo per controllare l'intero sistema da remoto mediante una comoda interfaccia di comando. E' stato pensato di scrivere speciali programmi sempre in esecuzione sull'unità di calcolo e quella di controllo: questo tipo di programma è chiamato *demone*.
Il compito principale dei demoni è quello di accettare ordini da rete, e sono strutturati secondo il modello client server: un concetto fondamentale su cui si basa la programmazione di rete. Questo modello prevede un programma di servizio, il server, che riceve una richiesta di connessione da parte di un altro programma, il client, e risponde fornendo a quest'ultimo un definito insieme di servizi.
Per assicurare la massima portabilità è stato scelto il linguaggio Java per implementare l'interfaccia di comando, e a questo punto nasce il problema di far comunicare processi che usano diverse strutture dati e diversi set di caratteri per rappresentare le stringhe. La soluzione di questo problema, e tutti gli altri dettagli concernenti la struttura dell'interfaccia di comando sono descritti in dettaglio nel capitolo 8.

Capitolo 2

Sistema Operativo

Un sistema operativo è un programma che agisce come intermediario tra l'utente e gli elementi fisici di un calcolatore; il suo scopo è fornire un ambiente nel quale un utente possa eseguire programmi in modo conveniente ed efficiente.
In questo capitolo descriveremo le generalità dei sistemi operativi, in particolare a quelli real-time, come lo è LynxOS: il sistema operativo scelto per il nostro sistema.

2.1 Introduzione ai sistemi operativi

Un sistema operativo è un insieme di programmi (software) che gestisce gli elementi fisici di un calcolatore (hardware); fornisce una piattaforma ai programmi d'applicazione e agisce da intermediario fra l'utente e la struttura fisica del calcolatore.
I sistemi operativi si distinguono, pertanto, in una vasta varietà, da quelli rudimentali, a quelli multiprogrammati, da quelli per PC da casa a quelli per sistemi integrati. In questa sezione si vogliono presentare i vari tipi di sistemi operativi e l'ambiente per il quale sono stati creati.

2.1.1 Sistemi "mainframe"

I sistemi di calcolo ti tipo "mainframe" sono stati i primi calcolatori impiegati per affrontare molte applicazioni scientifiche e commerciali. Inizialmente erano sistemi a lotti (batch) nei quali il sistema operativo eseguiva una, e una sola, applicazione per volta; in

seguito, con l'introduzione della multiprogrammazione, si evolsero in sistemi a partizione del tempo (time sharing).

2.1.2 Sistemi da scrivania

Sono i normalissimi PC domestici. Inizialmente erano dotati di sistemi operativi né multiutente, né a partizione del tempo; oggigiorno ne hanno uno moderno costruito con lo scopo di migliorare la comodità e la prontezza d'uso per l'utente.

2.1.3 Sistemi con più unità d'elaborazione

I sistemi con più unità d'elaborazione, anche chiamati "sistemi paralleli", sono i calcolatori con più di una CPU. In questo modo, nonostante la latenza di tempo dovuta alla comunicazione tra i processori, si ha un aumento del lavoro svolto, e una totale condivisione dei dispositivi periferici. Di conseguenza il sistema operativo per questi sistemi deve mirare a sfruttare quanto più possibile le capacità di tutte le CPU.

2.1.4 Sistemi distribuiti

I sistemi distribuiti si basano sulle reti, sfruttando la loro capacità di comunicazione per far cooperare più calcolatori nella soluzione dei problemi. Un sistema operativo distribuito deve operare in modo tale da rendere trasparente la comunicazione tra le macchine.

2.1.5 Sistemi palmari

Comprendono gli assistenti digitali e i telefoni cellulari d'ultima generazione. I loro sistemi operativi mirano alla facilità d'uso, e a funzionare velocemente poiché il più delle volte sono installati su un hardware limitato.

2.1.6 Sistemi d’elaborazione in tempo reale

Un sistema d’elaborazione in tempo reale (real-time) si usa quando è necessario fissare rigidi vincoli di tempo per le operazioni della CPU o il flusso di dati. Esistono due tipi di sistemi d’elaborazione in tempo reale:

- I sistemi d’elaborazione in tempo reale stretto (hard real-time): assicura che i compiti critici siano completati in un dato intervallo di tempo.
- I sistemi d’elaborazione in tempo reale debole (soft real-time): hanno caratteristiche meno ristrettive; i processi critici hanno priorità sugli altri processi, e la mantengono sino al completamento dell’esecuzione.

Nei sistemi d’elaborazione in tempo reale stretto, solitamente è limitata, o addirittura mancante, una memoria secondaria di qualsiasi tipo. Si preferisce memorizzare i dati in una memoria a breve termine o in una di sola lettura (ROM), dove i tempi di accesso sono decisamente più brevi. Alcuni fra i sistemi operativi classificabili nella categoria dei sistemi in tempo reale stretto sono:

- **RTAI** (*Dipartimento di Informatica e Applicazioni del Politecnico di Milano, www.rtai.org*): Sviluppato inizialmente ambito universitario, adesso RTAI è un progetto internazionale rilasciato con licenza Gnu Public License (GPL) e conforme allo standard POSIX. RTAI non è un vero e proprio sistema operativo: è una modifica al kernel di Linux, in modo da renderlo prelazionabile in ogni punto, e dotarlo di uno scheduler in tempo reale stretto. Le modifiche al kernel Linux mirano a catturare velocemente gli interrupt, e a trattare tutti i processi critici con una priorità diversa da quell’adottata per i rimanenti processi. Un difetto di RTAI, è dato dal fatto che i processi critici operano nello spazio d'indirizzamento del kernel, e non sono soggetti alle protezioni di memoria. Questa limitazione è stata superata da un’estensione di RTAI chiamata “Linux Real Time”. Alcune delle modifiche

fatte al kernel dal gruppo RTAI, e dall'azienda MontaVista (www.mvista.com), sono state implementate nei kernel Linux nella serie 2.6.x.

- **RTLinux** (*FSMLabs, www.fsmlabs.com*): Di questo sistema operativo sono sviluppate due versioni, una commerciale (RTLinuxPro) e una liberamente scaricabile e rilasciata sotto licenza Open Source (RTLinuxFree). Entrambi hanno funzionalità simili, e sono conformi allo standard POSIX. I processi critici girano nel proprio spazio d'indirizzamento, e di conseguenza si hanno a disposizione le protezioni della memoria e tutti gli strumenti per la programmazione di Linux. Questo sistema operativo nasce modificando il kernel di Linux.
- **LynxOS** (*LinuxWorks, www.linuxworks.com*): LynxOS a differenza dei precedenti, non è un sistema operativo derivato da Linux. Essendo stato concepito proprio per il tempo reale stretto, assicura un determinismo completo in ogni parte del sistema operativo. Il gruppo di lavoro che sviluppa LynxOS opera in questo campo da più di 16 anni, questo fa del suo sistema operativo il più utilizzato in ambito embedded. LynxOS è conforme allo standard POSIX, e ha una piena compatibilità con Linux, in modo da mettere in condizione i suoi utenti di usufruire degli strumenti per la programmazione per Linux, come la collezione GNU di strumenti per la programmazione (ad esempio gcc e gdb). L'adattamento di LynxOS alle applicazioni Linux, non ha nessun impatto sul kernel che mantiene la sua stabilità e affidabilità, quindi, per tutte le sue qualità, questo sistema operativo è stato scelto per il nostro sistema.
- **VxWorks** (*WindRiver, www.windriver.com*): è uno dei sistemi operativi real-time più usati in ambito aerospaziale e militare. Conforme agli standard POSIX, dotato d'ottime caratteristiche riguardo al real-time, VxWorks è stato pensato per girare su hardware ridotti e non ha un proprio file system. Distribuito da un'azienda leader del mercato, è una valida alternativa a LynxOS, ma non adatto ai nostri scopi.
- **QNX** (*QNX Software Systems, www.qnx.com*): QNX neutrino, alla conferenza mondiale sui sistemi embedded, è stato nominato come miglior sistema operativo per l'innovazione su dispositivi multiprocessore. Il suo kernel, appunto neutrino,

segue lo standard POSIX ed ha una struttura a microkernel che lo rende molto veloce e competitivo. Pensato e progettato per i dispositivi integrati, QNX è distribuito con il supporto Java preinstallato. Questo sistema operativo sarebbe una buon'alternativa a LynxOS ma non è dotato del "Board Support" per le CPU a nostra disposizione per le quali esiste solo in ambiente LynxOS, Linux e VxWorks.

2.2 Ambienti d'elaborazione embedded

I dispositivi d'elaborazione integrati (embedded) sono il tipo di calcolatori notevolmente più diffuso; incorporano sistemi operativi per l'elaborazione in tempo reale e si trovano quasi ovunque: nei motori d'automobili, nelle macchine industriali, forni a microonde, videoregistratori, eccetera. Si tratta tipicamente di sistemi piuttosto semplici, che offrono una povera o addirittura nessun'interfaccia d'utente, ma che danno la priorità al controllo e alla gestione di dispositivi fisici, come i motori d'automobili, esperimenti scientifici, e le macchine industriali.

2.3 Il sistema operativo LynxOS

LynxOS, in questo periodo disponibile nella versione 4.0.0, è uno dei sistemi operativi in tempo reale stretto più diffusi perché permette lo sviluppo di raffinati sistemi real-time in breve tempo, ottenendo eccellenti prestazioni.
Un fattore decisivo al successo di questo sistema operativo è stato accoppiare la tecnologia hard real-time, con lo standard POSIX, fornendo all'utente un ambiente simile a quello offerto dal sistema operativo unix (unix-like), permettendo in questo modo un facile adattamento ai suoi utenti.
Questo sistema operativo è in grado di operare anche in assenza di un dispositivo di memoria di massa secondario, comportamento tipico dei sistemi embedded, ma in ogni modo supporta i più comuni file system (FAT, ISO 9660, NFS, RAM disk). Ha anche un

suo file system nativo, pensato proprio per gli ambienti in tempo reale stretto (Lynx Fast File System).

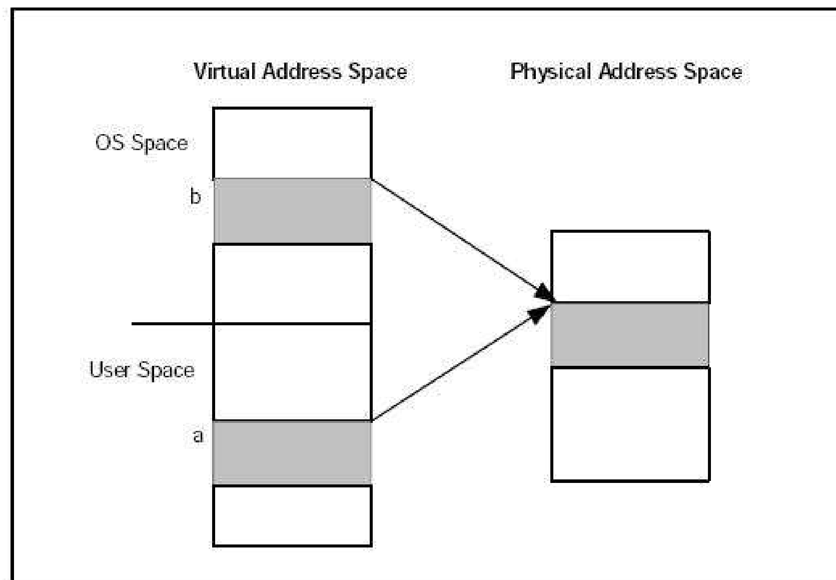

Figura 7: Il kernel ha accesso allo spazio d'indirizzamento dei processi

Di seguito sono riassunte le principali caratteristiche di LynxOS:

- Ambiente multiprocesso e multithreaded.
- Numero di processi potenzialmente senza limite.
- Gestione della memoria attraverso l'hardware MMU (Memory Management Unit).
- Supporta i thread a livello del kernel e gestisce 256 livelli di priorità.
- File system gerarchico, simile a quello UNIX.
- Strumenti d'utilità e shell script UNIX-like.
- Dispone degli strumenti GNU standard.
- Possibilità di usufruire di un'interfaccia grafica.
- Usa il protocollo di rete TCP/IP.
- Semafori con cambio di priorità dinamico (priority inheritance semaphore).
- Può essere usato anche senza dischi (Kernel Downloadable Image).

- Timer real-time POSIX.
- Possibilità di caricare un driver mentre il sistema operativo è in esecuzione.

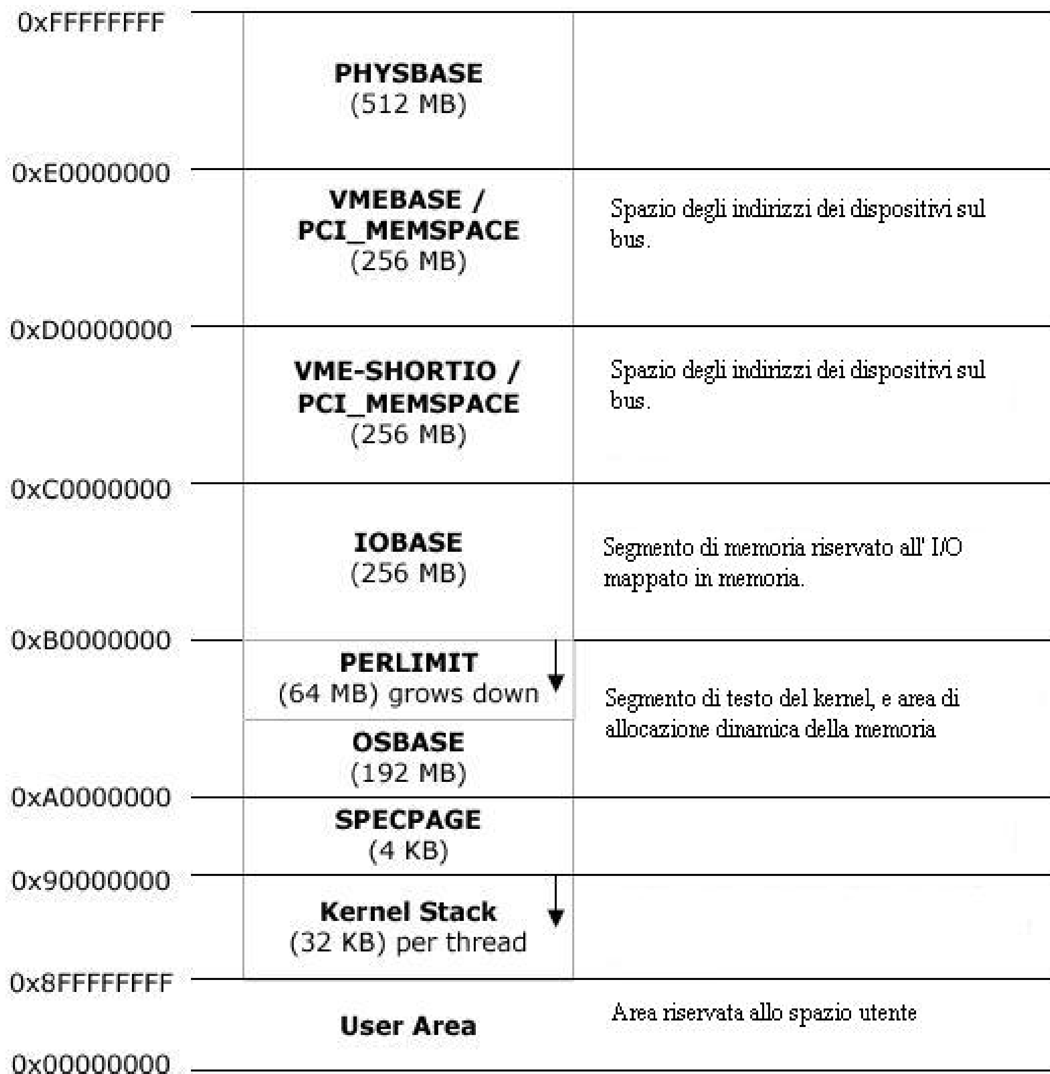

Figura 8: Mappa della memoria virtuale di LynxOS su un processore PowerPC

2.3.1 Gestione della memoria

LynxOS usa un'architettura ad indirizzamento virtuale, servendosi dell'hardware MMU (Memory Management Unit) per tradurre gli indirizzi virtuali in quelli fisici. Ogni

processo ha un proprio spazio virtuale e ciò evita che processi diversi possano interferire tra loro.

Nella Figura 8 è riportato uno schema che rappresenta il modo in cui lo spazio d'indirizzamento virtuale è ripartito dal sistema operativo LynxOS quando gira su un processore PowerPC.

La costante OSBASE definisce il limite superiore accessibile alle applicazioni, mentre tutti gli indirizzi sopra questo limite sono accessibili solo dal kernel. Questo schema di ripartizione della memoria fa sì che durante un cambio di contesto, l'unica porzione della memoria ad essere rimappata è quella associata al task utente.

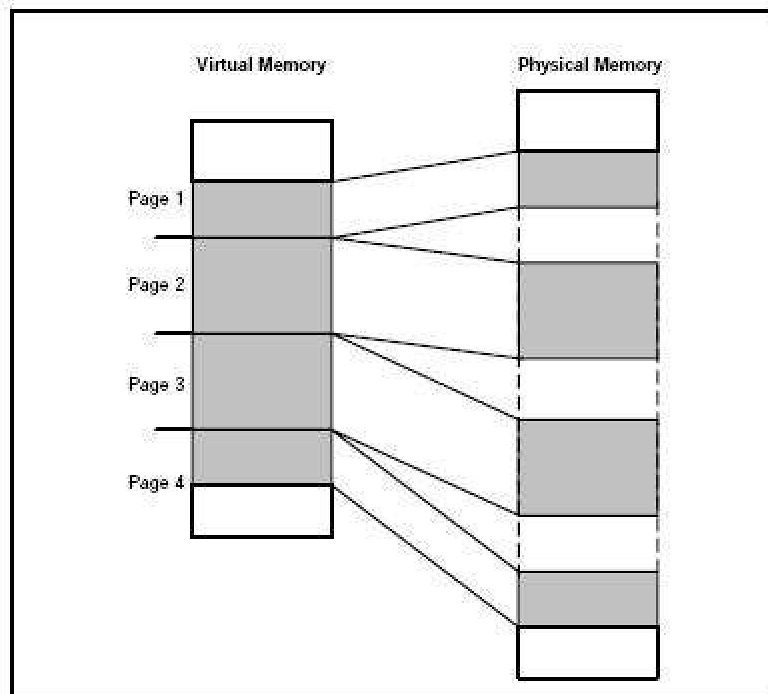

Figura 9: La memoria virtuale viene partizionata in più pagine di memoria fisica

Il codice del kernel ha invece accesso a tutta la memoria, facilitando il passaggio dei dati tra i task utenti e i driver (illustrazione in Figura 7).

Ad esempio consideriamo un processo utente che esegua una chiamata di sistema, supponiamo la read(); la routine associata del driver prende in input un indirizzo di

memoria dello spazio utente e può trasferire dati direttamente dal dispositivo al buffer utente senza l'utilizzo di speciali funzioni.

Quest'architettura ad indirizzamento virtuale permette che i task vedano la memoria come uno spazio contiguo, ma in realtà può essere partizionata in più pagine di memoria come illustrato nella Figura 9. Ciò rende i trasferimenti DMA (Direct Memory Access) molto complicati: i controller DMA presuppongono che la memoria sia allocata in modo contiguo. Per far fronte a quest'inconveniente, LynxOS mette a disposizione agli sviluppatori di driver una vasta gamma di funzioni per gestire il bloccaggio della memoria, e la traduzione di indirizzi virtuali in fisici e viceversa, e delle funzioni per l'allocazione della memoria fisica contigua. Resta in ogni caso compito del programmatore passare correttamente i parametri al controller DMA, assicurandosi che esso esegua il suo compito usando la giusta area di memoria.

2.3.2 Strumenti di sincronizzazione a livello del kernel

La sincronizzazione assicura che determinati eventi accadano in un ordine prestabilito. Questo a livello del kernel, come ad esempio in un driver, significa assicurare che le risorse condivise siano usate in maniera coerente e controllata. LynxOS mette a disposizione dei programmatori tre strumenti per la sincronizzazione a livello del kernel:

- Semafori: classici semafori contatore, che a differenza dei semafori binari, possono assumere valori diversi da 0 ed 1 e risultano, quindi, utili a controllare l'accesso a molti tipi di risorse.
- Disabilitazione della prelazione: Si può evitare che il processo sia prelazionato dal dispatcher. Questo può servire per eseguire regioni di codice critiche.
- Disabilitazione degli interrupt: Lascia gli interrupt pendenti a tempo indefinito. Notiamo che se disabilitiamo gli interrupt, è disabilitata di conseguenza anche la prelazione. Questo meccanismo di sincronizzazione è anch'esso utile in presenza di porzioni di codice critiche.

2.3.3 Gestione degli interrupt

Le eccezioni hardware, consegnate al processore per segnalare un particolare evento, sono chiamate "interrupt". Quest'ultimi sono spesso usati per segnalare eventi come:

- Completamento di un'operazione: ad esempio un trasferimento DMA (Direct Memory Access).
- Nuovi dati presenti nel buffer del dispositivo.
- Entrata in funzione del timer del dispositivo.

Di solito ad ogni segnale d'interrupt è associata una routine di servizio da eseguire chiamata ISR (Interrupt Service Routine). Una routine di servizio è mandata in esecuzione appena il segnale d'interrupt arriva alla CPU; questo può seriamente rallentare i tempi di risposta dei processi in tempo reale stretto. Per evitare che le routine d'interrupt rallentino i processi ad alta priorità, gli sviluppatori possono far uso dei thread a livello del kernel (kernel thread).

2.3.4 Thread a livello del kernel

In un normale sistema, come anche in LynxOS, le routine di servizio degli interrupt hanno la massima priorità e sono gestite prima di qualsiasi task. Ciò significa che se è in esecuzione un processo in tempo reale stretto, esso è sospeso non appena arriva un interrupt alla CPU. Tutto questo introduce un ritardo nel completamento del processo critico; i thread a livello del kernel (kernel thread) sono la soluzione al ritardo introdotto dalle routine di servizio degli "interrupt".

In pratica per ogni tipo d'interrupt è creato un thread in grado di gestirlo. Questo thread è messo subito in attesa su un semaforo e verrà attivato solamente quando effettivamente un interrupt di quel tipo è arrivato alla CPU. Il meccanismo è illustrato nello schema rappresentato in Figura 10.

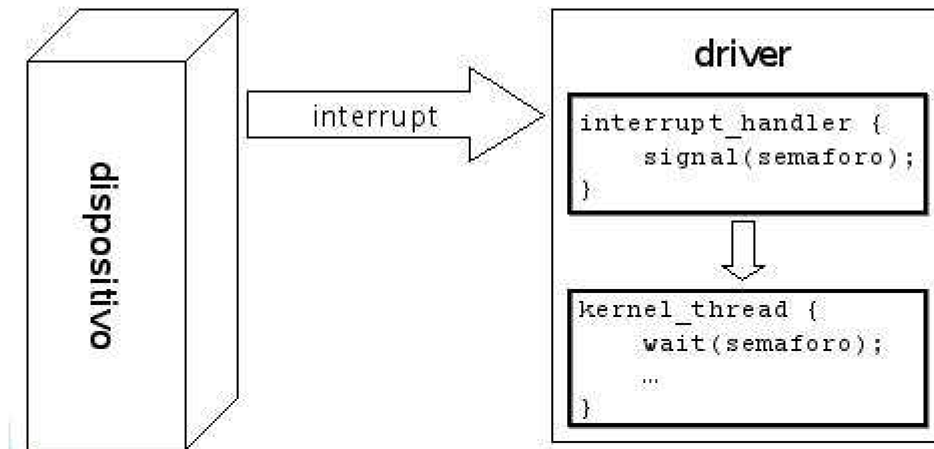

Figura 10: Gestione interrupt con i kernel thread

In questo modo la routine di servizio è ridotta all'osso poiché formata da un'unica istruzione, e non appena arriva il relativo interrupt, il ritardo da lei introdotto sarà minimo. Da notare che il kernel thread non ha la stessa priorità della routine di servizio dell'interrupt, e può essere anche prelazionato e messo in coda.

Supponiamo che arrivi un interrupt durante l'esecuzione di un task in tempo reale stretto: la routine di servizio sarà velocissima a eseguire la sua unica istruzione. A questo punto se il kernel thread ha priorità inferiore rispetto al task critico, sarà sospeso in attesa del suo completamento.

Capitolo 3

Il bus VME

Il modo in cui i vari dispositivi riescono a scambiarsi informazioni tra loro all'interno di una struttura di elaborazione digitale rappresenta una caratteristica fondamentale del sistema.
In questo capitolo saranno esaminati le caratteristiche dei bus in generale, con particolari riferimenti al VMEBus usato nel nostro sistema.

3.1 Generalità dei bus

I più semplici calcolatori sono normalmente costituiti da un BUS comune che collega in parallelo tutte le periferiche. Un calcolatore più complesso può contenere e utilizzare contemporaneamente più BUS, e il principio di funzionamento rimane lo stesso illustrato in Figura 11.

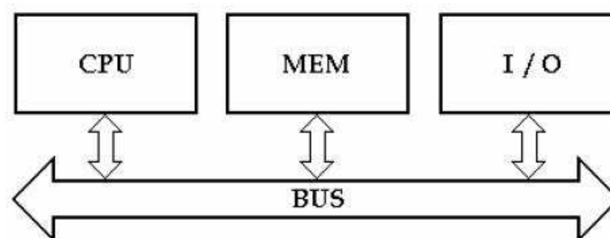

Figura 11: Semplice struttura a BUS

La scelta di questa struttura è dettata da considerazioni di tipo economico e di praticità; se si volesse collegare ogni dispositivo all'altro mediante una connessione punto-punto, si avrebbe un sistema molto costoso e allo stesso tempo più complesso. E' evidente che se due dispositivi comunicano tra loro, stanno, di fatto, occupando il bus, impedendone l'uso alle altre unità presenti nel sistema: il bus è una risorsa condivisa.
Con una struttura a bus ogni dispositivo con un hardware adatto può svolgere uno dei seguenti ruoli in un determinato periodo di tempo:

- *Master* è il dispositivo che in questo periodo detiene le linee del bus; può esistere solo un master per volta ed è denominato "commander".
- *Arbitro* è il dispositivo che decide chi può e chi non può appropriarsi dei fili del bus.
- *Slave* è il dispositivo che in questo periodo sta comunicando con il Master.

L'intero fascio delle linee dal bus (illustrazione in Figura 12), per comodità, è diviso logicamente in tre sotto insiemi di collegamenti: bus dati, bus indirizzi, e bus controlli.

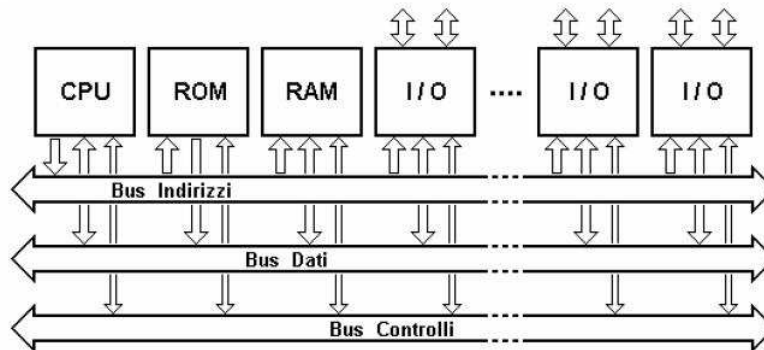

Figura 12: Suddivisione logica delle linee del bus

Il bus dei dati e il bus degli indirizzi sono formati dall'insieme di linee dedicate ad ospitare i dati e i relativi indirizzi. Il bus dei controlli è invece composto da linee con compiti di

vario tipo, anche funzionalmente molto diversi tra loro, ma necessari per il controllo dell'intero sistema e per la gestione (arbitraggio) dello scambio di informazioni.
Il protocollo di comunicazione può essere di tipo sincrono se, una volta avviata la trasmissione dei dati, essa è regolata da un segnale di "clock" a frequenza fissa, o di tipo asincrono nel senso che le sequenze di trasmissione e ricezione dei dati non necessitano di una temporizzazione, ma si basano su un protocollo di "handshaking": in questo modo la frequenza di trasmissione dei dati si adegua automaticamente alle caratteristiche del dispositivo che trasmette e di quello che riceve.

3.2 Storia del VMEBus

Il VME (Versa Module Europe) è l'erede del lavoro fatto dai progettisti della Motorola, a partire dal 1978, per dotare le CPU della serie 68k di un bus per il collegamento dei periferici all'altezza delle prestazioni che queste potevano esprimere. Questo lavoro produsse nel 1979 le prime specifiche di un "crate bus" "proprietario" denominato Versabus, poi accettato come standard (IEEE 970). La Motorola aveva bisogno di promuovere la produzione del suo microprocessore single chip sul quale aveva fatto enormi investimenti, ma il successo dei sistemi basati sul Versabus fu molto condizionato, specialmente in Europa, dall'inadeguatezza degli strumenti software.
In quegli stessi anni un organismo sopranazionale, l'International Electro-technical Commission (IEC) aveva proposto uno standard meccanico modulare per il formato delle schede elettroniche detto comunemente "Eurocard".
La Motorola decise di approfittare dell'incoraggiamento che la Comunità Europea dava in quel periodo a tutte le iniziative di standardizzazione in campo elettronico, aderendo con alcuni progettisti della sede di Monaco di Baviera ad un gruppo di lavoro per la definizione di un sistema d'elaborazione distribuita, al quale diedero supporto anche altri produttori.
Il gruppo di lavoro propose un sistema, in parte logicamente ispirato al Versabus, ma alloggiato in una meccanica Eurocard. Della parentela col Versabus rimase traccia nella sigla VME.

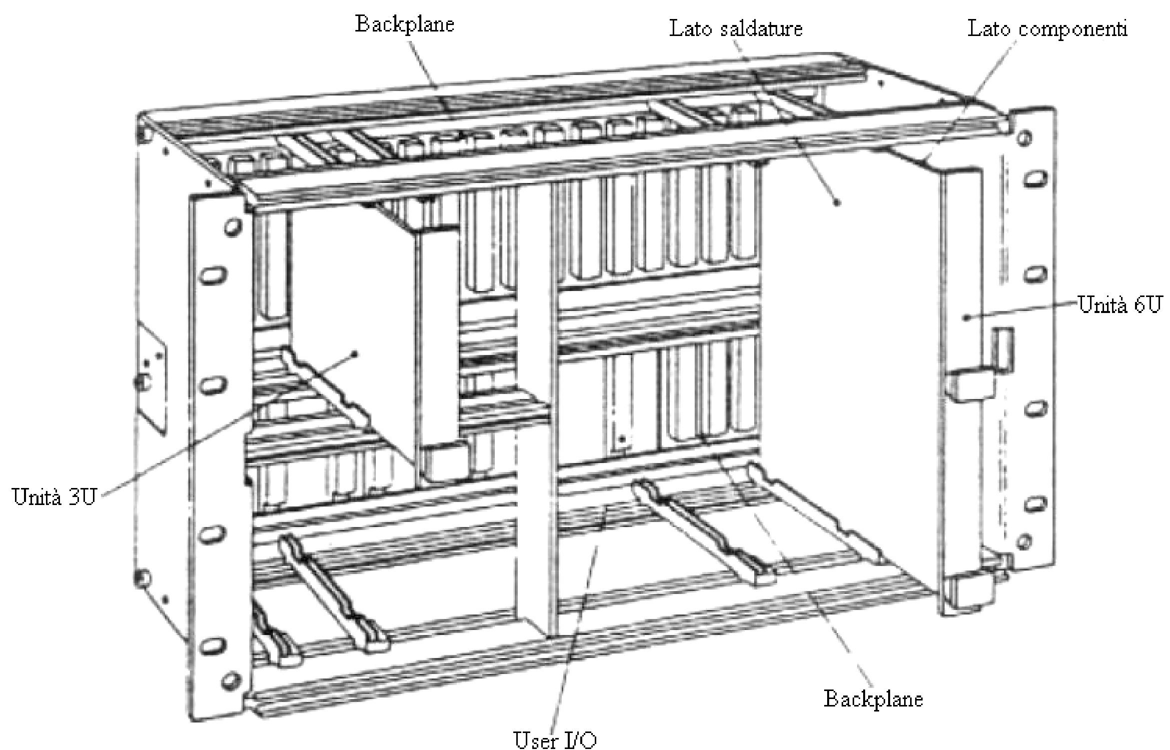

Figura 13: Il VMEBus ha la forma di un cestello (crate)

La presenza nel gruppo di lavoro di progettisti non direttamente legati alla Motorola produsse, però, delle specifiche sufficientemente generali da poter essere interpretate efficacemente anche con CPU di altre marche. La veridicità di quest'affermazione è oggi avvalorata dalla presenza sul mercato di molte schede processore in standard VME non basate su CPU Motorola. Oggigiorno il VMEBus (Figura 13) si è imposto nel mercato mondiale ed è molto utilizzato nel controllo industriale, in ambiti militari e aerospaziali, in sistemi di simulazioni, e in macchine per l'analisi medica.

3.2.1 VMEBus Standard

La prima versione del VME (1987) (D16/A24) prevedeva schede Eurocard di altezza 3U (100 mm) o 6U (233 mm) con un connettore DIN 41612 a 96 pin su tre colonne (Figura 14).

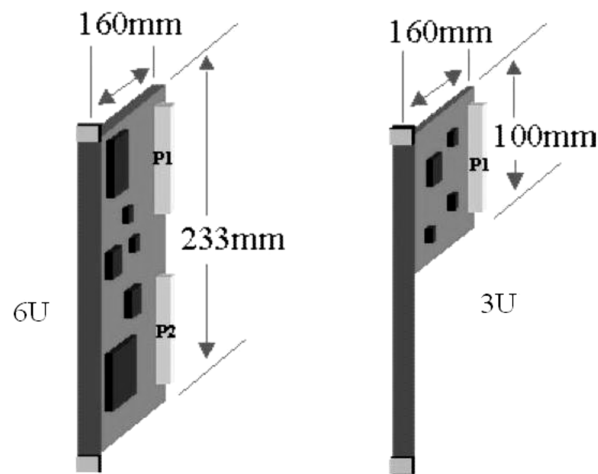

Figura 14: Formato 6U e 3U

Il connettore, che nel formato 6U era montato in alto nella scheda (P1), prevedeva 16 bit per i dati e 24 bit per gli indirizzi (Figura 15).

La versione successiva del VMEBus (1994) aggiunse il connettore P2 al connettore P1, portando così il bus a 32 linee per i dati e 32 linee per gli indirizzi, prevedendo un'ampiezza di banda pari a 40Mbyte/s stimando la frequenza dei trasferimenti a 10MHz[1]. Quest'ultima revisione del VMEBus è stata in uso per molti anni prima di essere sostituita dal VME64.

3.2.2 VME64

Il VME64, prevede 64 bit di dati e 32 bit di indirizzi con connettori a 160 pin su cinque colonne, ed ha, con trasferimenti a 10MHz, ampiezza di banda pari a 80 Mbyte/s. Per quanto riguarda la retrocompatibilità, tutti i dispositivi che possono alloggiare nel VMEBus Standard possono essere usati anche con VME64.

Il VME64 è stato ottimizzato nel 1997 con il nome di VME64x.

[1] La linea di "Address Strobe" (AS*) deve essere asserita almeno 40ns per essere validata e deve essere deasserita per altri 40ns per due cicli successivi. In totale abbiamo un tempo non inferiore a 80ns. Considerando un ritardo di 10ns da parte del dispositivo trasmettente, e un altro ritardo di 10ns da parte del dispositivo ricevente si ha un tempo totale di circa 100ns, per cui si stima la velocità di 10MHz.

PIN NUMBER	ROW A SIGNAL MNEMONIC	ROW B SIGNAL MNEMONIC	ROW C SIGNAL MNEMONIC
1	D00	BBSY*	D08
2	D01	BCLR*	D09
3	D02	ACFAIL*	D10
4	D03	BG0IN*	D11
5	D04	BG0OUT*	D12
6	D05	BG1IN*	D13
7	D06	BG1OUT*	D14
8	D07	BG2IN*	D15
9	GND	BG2OUT*	GND
10	SYSCLK	BG3IN*	SYSFAIL*
11	GND	BG3OUT*	BERR*
12	DS1*	BR0*	SYSRESET*
13	DS0*	BR1*	LWORD*
14	WRITE*	BR2*	AM5
15	GND	BR3*	A23
16	DTACK*	AM0	A22
17	GND	AM1	A21
18	AS*	AM2	A20
19	GND	AM3	A19
20	IACK*	GND	A18
21	IACKIN*	SERCLK (1)	A17
22	IACKOUT*	SERDAT*(1)	A16
23	AM4	GND	A15
24	A07	IRQ7*	A14
25	A06	IRQ6*	A13
26	A05	IRQ5*	A12
27	A04	IRQ4*	A11
28	A03	IRQ3*	A10
29	A02	IRQ2*	A09
30	A01	IRQ1*	A08
31	-12V	+5V STDBY	+12V
32	+5V	+5V	+5V

Figura 15: Assegnazione PIN per il connettore P1

3.2.3 Le evoluzioni di VME: VME64x e VME 320

Evoluzioni della topologia di connessione e del protocollo di comunicazione hanno consentito di portare la velocità di trasferimento dei dati fino ai valori di 160Mbyte/s e 320Mbyte/s consentiti da VME64x e da VME320, quest'ultimo dotato di un limite teorico di 1Gbyte/s.

PIN NUMBER	ROW A SIGNAL MNEMONIC	ROW B SIGNAL MNEMONIC	ROW C SIGNAL MNEMONIC
1	USER I/O	+5 Volts	USER I/O
2	USER I/O	GND	USER I/O
3	USER I/O	RESERVED	USER I/O
4	USER I/O	A24	USER I/O
5	USER I/O	A25	USER I/O
6	USER I/O	A26	USER I/O
7	USER I/O	A27	USER I/O
8	USER I/O	A28	USER I/O
9	USER I/O	A29	USER I/O
10	USER I/O	A30	USER I/O
11	USER I/O	A31	USER I/O
12	USER I/O	GND	USER I/O
13	USER I/O	+5 Volts	USER I/O
14	USER I/O	D16	USER I/O
15	USER I/O	D17	USER I/O
16	USER I/O	D18	USER I/O
17	USER I/O	D19	USER I/O
18	USER I/O	D20	USER I/O
19	USER I/O	D21	USER I/O
20	USER I/O	D22	USER I/O
21	USER I/O	D23	USER I/O
22	USER I/O	GND	USER I/O
23	USER I/O	D24	USER I/O
24	USER I/O	D25	USER I/O
25	USER I/O	D26	USER I/O
26	USER I/O	D27	USER I/O
27	USER I/O	D28	USER I/O
28	USER I/O	D29	USER I/O
29	USER I/O	D30	USER I/O
30	USER I/O	D31	USER I/O
31	USER I/O	GND	USER I/O
32	USER I/O	+5 Volts	USER I/O

Figura 16: Assegnazione PIN per il connettore P2

Lo standard VME64x (1997) nasce dalle ceneri di VME64 attraverso l'adozione del protocollo 2eVME, le cui caratteristiche principali sono:

- impiegare entrambi i fronti del segnale digitale per trasportare le informazioni;
- consentire sia al master che allo slave di terminare il trasferimento.

La compatibilità del protocollo 2eVME è limitata dal fatto che richiede componenti dotati di logica ETL (Enhanced Transceiver Logic).
Una successiva evoluzione del protocollo 2eVME è rappresentata da 2eSST che trasforma il bus VME durante il passaggio dei dati in un bus sincrono alla sorgente (da cui il suffisso SST: Source Synchronous Transfer). Dopo un singolo segnale di strobe i dati sono trasferiti senza attendere il riconoscimento da parte del destinatario, eliminando così la dipendenza della banda ammissibile dal tempo di propagazione sulle linee. VME320 prevede una topologia circuitale a stella in cui tutte le capacità parassite, non essendo più distribuite lungo il backplane, non danno più luogo ai fenomeni di distorsione e riflessione tipici delle linee di trasmissione. Ciò consente di adottare il protocollo 2eSST per ottenere velocità di trasferimento di 320Mbyte/s ed oltre. Lo standard VME320 è compatibile al 100% con le schede VME tradizionali che possono coesistere sul medesimo backplane con le più veloci schede che adottano il protocollo 2eSST. VME320 a differenza degli altri è di tipo proprietario e può essere prodotto esclusivamente dall'Arizona Digital.

3.3 Principali caratteristiche del VMEBus Standard

Le specifiche VME, per favorire una migliore comprensione della struttura del bus, suddividono l'insieme delle linee, dal punto di vista funzionale, in cinque sotto-bus ciascuno in grado di eseguire autonomamente un certo tipo di operazione specifica:

- Bus trasferimento dati: comprende l'insieme di linee dedicate ai dati e ai relativi indirizzi.
- Bus per l'arbitraggio: usato per ottenere il controllo del bus.
- Bus per la gestione delle priorità di interruzione: usato per la gestione delle interruzioni.
- Bus dei servizi: comprende servizi utili come il reset del sistema o la segnalazione di errori.

- Bus seriale (VMSbus): può essere utilizzato per scambiare brevi messaggi urgenti tra moduli o sistemi VME. Utilizza due soli pin.

Il modello di architettura è, a tutti i livelli, quello master/slave, i trasferimenti sul bus sono asincroni e prevedono linee dei dati separate dalle linee dedicate agli indirizzi. Una caratteristica particolare del bus VME sono le 6 linee di "address modifier code" che sono alla base della duttilità di indirizzamento e di gestione del sistema, e implementano alcune caratteristiche del bus come la gestione dinamica dei campi di indirizzamento. Nella tabella, rappresentata in Figura 17, sono illustrati tutti i possibili codici degli address modifier code.

A.M. (hex)	IACK*	Address bits	Tipo di trasferimento
3F	1	24	Standard supervisory block transfer
3E	1	24	Standard supervisory program access
3D	1	24	Standard supervisory data access
3B	1	24	Standard non-privileged block transfer
3A	1	24	Standard non-privileged program access
39	1	24	Standard non-privileged data access
2D	1	16	Short supervisory access
29	1	16	Short non-privileged access
10-1F	1	—	Definibili dall'utente
0F	1	32	Extendend supervisory block transfer
0E	1	32	Extendend supervisory program access
0D	1	32	Extendend supervisory data access
0B	1	32	Extendend non-privileged block transfer
0A	1	32	Extendend non-privileged program access
09	1	32	Extendend non-privileged data access
XX	0	3	Interrupt acknowledge cycle

Figura 17: Codici degli address modifier

Ai codici di "address modifier" è anche affidato il compito di separare trasferimenti dati fatti in modo supervisore (privilegiato) da quelli fatti in modo utente (non privilegiato).
A scopo d'esempio analizziamo un tipico ciclo di lettura sul bus, illustrato nello schema di Figura 18.

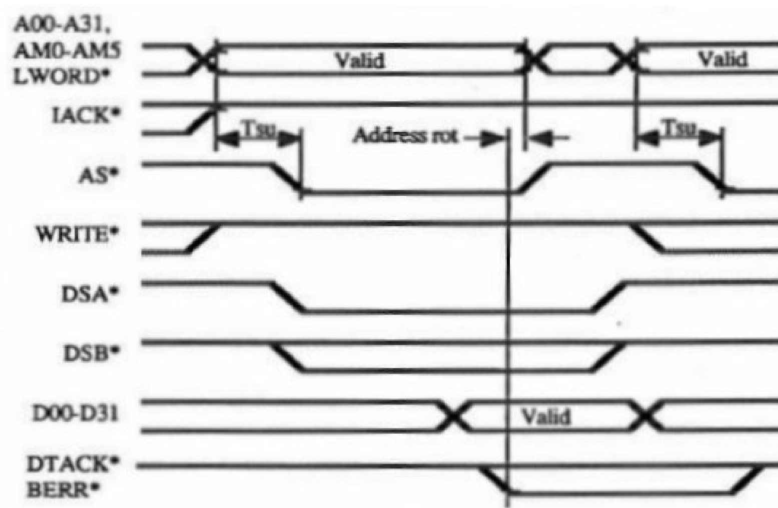

Figura 18: Tipico ciclo di lettura

L'esempio si riferisce ad un caso in cui è adottata la tecnica dello "address pipelining", poiché il nuovo indirizzo è già sulle linee del bus quando il vecchio ciclo di lettura non ancora è terminato. La successione delle operazioni è la seguente:

1. Il master, mediante le linee AM0-AM5, A01-A032, LWORD ed IACK*, indirizza la periferica e nega la linea WRITE* poiché si tratta di una lettura.
2. Di seguito, sempre il master, convalida le linee indirizzi con la transizione verso il livello basso della linea AS. Sono inoltre attivate basse le linee DSA e DSB.
3. Lo slave decodifica l'indirizzo, e avendo già le informazioni necessarie, scrive sul bus il dato richiesto. Alla fine convalida il tutto asserendo la linea DTACK*.
4. Il master legge i dati registrandoli nel proprio buffer, e alla fine di questa operazione attiva le linee DSA e DSB.
5. Lo slave a questo punto, sicuro che il ciclo di lettura è finito correttamente, rilascia le linee dei dati e nega la linea DTACK*. Utilizzando la tecnica dello "address pipelining" in quest'istante di tempo, già è presente sulle linee A01-A032 il nuovo indirizzo (supposto che ci sia un altro ciclo di lettura).

3.4 Gestione degli interrupt del VMEBus Standard

Il sistema degli interrupt è fondamentale in un sistema atto ad eseguire applicazioni real-time. Abbiamo già visto che il VME prevede ben 7 livelli di priorità per il program interrupt. Un ciclo di interruzione comincia con l'attivazione da parte di uno slave di una delle linee di richiesta che, essendo strutturate come linee omnibus, possono essere utilizzate da un numero qualsiasi di dispositivi.

Questo ciclo ha due importanti funzioni, una è l'arbitraggio dell'interrupt e l'altra è il ciclo di lettura del vettore di interrupt. Il sistema di interrupt del VMEBus grazie all'interrupt vettorizzato elimina la necessità del polling per trovare il dispositivo richiedente ed associarlo ad una routine di servizio. Quest'ultimo implica che il modulo di gestione dell'interrupt sia in grado di richiedere ed ottenere il controllo del bus dati attraverso il quale potrà ricevere dal dispositivo il vettore di interruzione che servirà ad individuare in memoria la sua "routine di servizio".

Quando il gestore dell'interrupt avrà ottenuto il bus dati piloterà le linee A01-A03, attivando basse le linee IACK* e AS*. Sulle linee indirizzi viene messo il codice binario del livello di priorità corrispondente alla linea da cui è arrivata la richiesta e viene confermato dalla linea AS*. Gli A.M. non debbono essere usati. Questa operazione serve ai dispositivi che operano con un diverso livello di priorità per non interferire nella fase successiva. Essi, riscontrando che il loro livello di priorità è diverso da quello che sta per essere servito, dovranno rendersi "trasparenti" al segnale che punta ad individuare il dispositivo richiedente, anche se nel frattempo avevano richiesto, a loro volta, l'Interrupt.

Il modulo di gestione (interrupt handler) tiene sotto osservazione le linee di richiesta e, quando rileva una linea attivata, chiede il controllo del bus. Dopo averlo ottenuto, in risposta al segnale, genera un "interrupt acknowledge cycle" illustrato in modo sintetico nella Figura 19.

Nello schema è stato ipotizzato che la richiesta sia venuta sulla linea IRQ3* la che si ottiene impostando i pin A03 A02 e A01 rispettivamente a '0', '1', '1'.

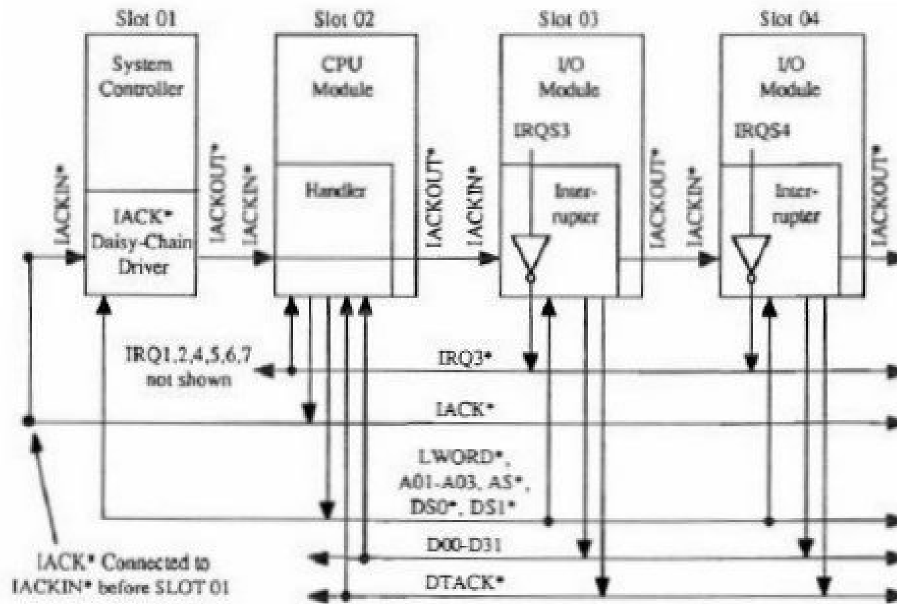

Figura 19: Sistema di interrupt

Il diagramma temporale del ciclo di interrupt per un vettore d'interruzione ad 8 bit è illustrato in Figura 20.

Il segnale IACK* basso indica che è in atto un ciclo di interrupt. L'arbitraggio dell'interrupt è effettuato con la tecnica della daisy-chain che deve essere gestita da un driver che si trova nella slot 1 del bus. Esso, se non è anche gestore dell'interrupt, riceve da quest'ultimo il segnale IACK* sul pin IACKIN* e dopo aver ricevuto basso DS0* o DS1* od entrambi eventualmente accompagnati da LWORD* (a seconda che sia previsto un vettore di interrupt ad 8, 16 o 32 bit) lo propaga attraverso il pin IACKOUT* verso il pin IACKIN* della successiva slot occupata (daisy-chain).

Quando il modulo che ha richiesto l'interrupt (o il primo di quelli che hanno attivata la linea) riceve dal modulo che lo precede lungo il bus il segnale IACKIN*, presenta sulle linee dati il vettore di interrupt e termina il ciclo asserendo bassa la linea DTACK*.

Il segnale DTACK* avvia la procedura di rilascio da parte del gestore che pilota alta le linee di IACK* e di AS* e rilascia le linee A01-A03. A questo punto i dispositivi sul bus attraversati dal segnale della daisy-chain, dopo che il gestore dell'interrupt neghi la linea

AS*, devono, entro un tempo preciso (40 ns), pilotare alte le linee IACKIN*/ IACKOUT* per evitare che s'intreccino due diversi cicli di interrupt.

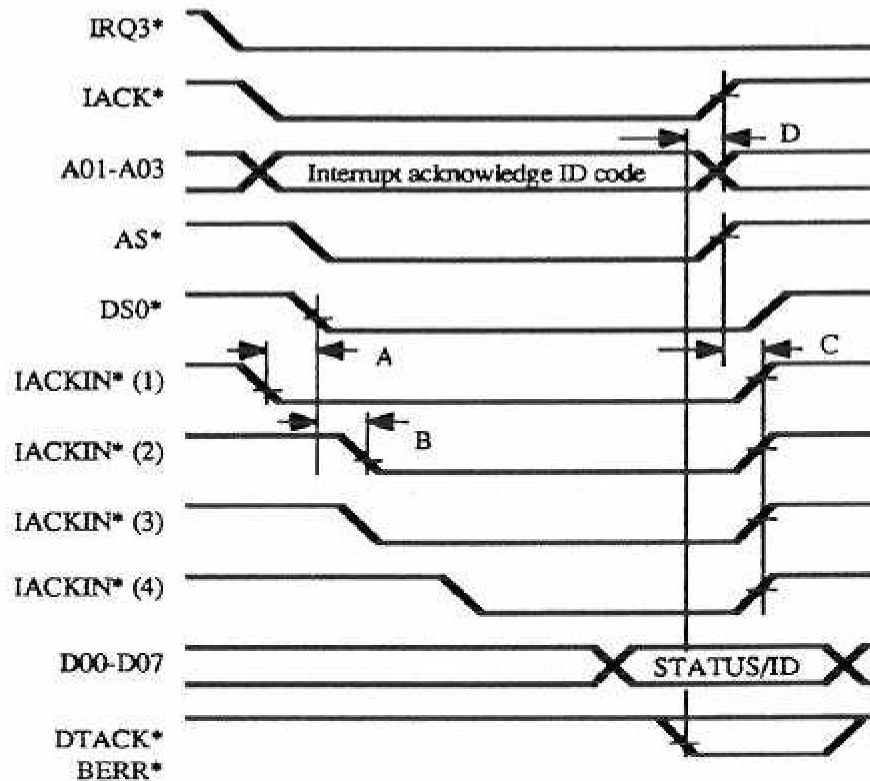

Figura 20: Tipico ciclo di interrupt

3.5 Arbitraggio del VMEBus Standard

L'arbitraggio del bus è un problema cruciale in un bus multi-master. Fortunatamente il VMEBus è dotato di un sistema d'arbitraggio che consente di gestire anche una situazione dove tutti i dispositivi sono master. Non dimentichiamo che il bus rimane una risorsa condivisa: il fatto di avere tutti e 21 gli slot occupati da master non significa che essi lavorano parallelamente.

Le specifiche VME hanno affrontato il problema con estrema attenzione ed hanno operato in due direzioni diverse. Da una parte prevedono l'uso di collegamenti tra schede processori e periferici (in particolare memorie) su bus ausiliari, paralleli, come ad esempio, il VSB o, seriali, come il VMS, per rendere la gestione di ogni sottosistema il meno critica possibile, dall'altra prevedono una struttura hardware per l'arbitraggio del bus molto duttile e potente ma che possa, all'occorrenza, essere facilmente minimizzata per non appesantire i piccoli sistemi che non hanno bisogno di una gestione sofisticata delle priorità per assumere il possesso del bus. La politica d'arbitraggio del VMEBus prevede che il dispositivo situato nel primo slot svolga il ruolo di "arbitro"; questo fa in modo che non ci sia più di un master attivo in un determinato istante di tempo.
Il bus di arbitraggio prevede quattro distinti livelli di richiesta del bus e la priorità d'ogni dispositivo è determinata dal livello scelto e anche dalla distanza dall'arbitro; allo stesso livello di richiesta del bus, ricordando che l'arbitro si trova nel primo slot, è assegnata maggior priorità al dispositivo più vicino allo slot 1. Una volta ottenuto il possesso del bus, un dispositivo può detenerlo a tempo indefinito, a meno che, l'arbitro non attivi la linea BCLR*. Il metodo di arbitraggio può usare la linea BCLR* per favorire alcuni dispositivi (creando così una gerarchia) oppure implementare una tecnica di rotazione (Round-Robin Arbiter) che garantisce a ciascun dispositivo una certa fetta di tempo.

3.6 Sommario delle caratteristiche

SO tipo Unix	SO tipo Wintel	SO real-time
Solaris	DOS	VxWorks
SunOS	OS-2	pSOS
Berkeley	Windows 3.1	LynxOS
AT&T	Windows 95/98	QNX
Linux	Windows NT	RTLinux

Tabella 1: Sistemi operativi che supportano VMEBus.

Per la sua affidabilità, velocità, scalabilità, il VMEBus è utilizzato in una vasta varietà di applicazioni. Di conseguenza il VMEBus è supportato da vari sistemi operativi elencati nella Tabella 1.

Moduli funzionali	Descrizione
Master	Può iniziare cicli bus sul Data Transfer Bus (DTB)
Slave	Può rilevare e partecipare ai cicli di bus
Location Monitor	Monitora il DTB e asserisce dei segnali
Bus Timer	Interrompe un ciclo DTP se troppo lungo
Interrupter	Genera richieste di interrupt
Handler	Risponde alle richieste degli interrupter
IACK* Daisy Chain Driver	Pilota IACK* daisy chain
Requester	Usato per richiedere il possesso del DTB
Arbiter	Gestisce il possesso del bus
System Clock Driver	Fornisce un clock di 16MHz
Power Monitor	Genera i segnali SYSRESET* e ACFAIL*

Tabella 2: Moduli funzionali del VMEBus

Nella Tabella 2 si vuole descrivere l'architettura del bus usando il concetto di "modulo funzionale", mentre nella Tabella 3 si vogliono riassumere le proprietà comuni a tutti i tipi di bus VME:

Elemento	Specifica	Note
Architettura	Master/Slave	
Meccanismo di trasferimento	Asincrono	Nessun clock per sincronizzare le transazioni
Multiplexing delle linee dati	No	
Range di indirizzamento	16 24, 32 bit	L'indirizzo è selezionato dinamicamente
Dimensione del dato	8, 16, 24, 32 bit	Dimensione selezionata dinamicamente
Rilevamento errore	Si	Si usa il segnale BERR*
Interrupt	7 livelli di priorità	
Capacità multiprocessore	1-21 processori	
Numero di slot	21	

Tabella 3: Caratteristiche generali del VMEBus

Capitolo 4

Dispositivi utilizzati: ADC e DAC

I costituenti fondamentali del sistema costruito sono il dispositivo analogico-digitale (ADC) e i dispositivi digitale-analogico (DAC). Entrambi montati sull'unità di controllo, permettono di interfacciare il calcolatore con l'ambiente esterno.
In questo capitolo si descriverà in dettaglio il dispositivo DAC, precisamente il modello MPV955 prodotto da Pentland System; inizialmente non avevamo un driver per LynxOS adatto a questo dispositivo, quindi è stato scritto da zero.
Per il dispositivo ADC, precisamente il modello VGD5 prodotto sempre dalla Pentland System, è stato usato un driver (seppur minimale) già in nostro possesso. Di quest'ultimo dispositivo non faremo una trattazione completa, ma solo breve accenno alle sue caratteristiche.

4.1 Dispositivo MPV955

La scheda MPV955, descritta in maniera dettagliata nel manuale d'uso distribuito dalla casa produttrice (rif. [9]), è meccanicamente ed elettronicamente compatibile con il bus VME. Tra le sue caratteristiche principali notiamo:

- Memoria interna di 16K;
- Possibilità di filtrare l'output per annullare le differenze di tempo introdotte dai latch;

- Range di tensioni in output bipolari oppure unipolari;
- Watchdog timer;
- Risoluzione dei dati a 16 bit;
- Otto canali di output.

Il dispositivo ha, quindi, otto canali di output analogico ad alta velocità, ognuno dotato di un proprio filtro; inoltre è in grado di funzionare in due modi:

- Modalità continua;
- Modalità one-shot.

La scheda mette a disposizione dell'utente tre scelte di triggering (tipo di clock):

- Clock interno;
- Clock esterno;
- Clock ad eventi.

Infine è capace di utilizzare tutte le sette linee d'interruzione del VMEBus, supporta gli "Address Modifier Code" (rif. pagina 39), e dà la possibilità di riprogrammarli tramite PROM.
Tratteremo in dettaglio sia le impostazioni hardware del dispositivo, modificabili a sistema spento, che quelle software, modificabili quando il sistema è in funzione.

4.2 Impostazioni hardware del dispositivo MPV955

Al fine di installare correttamente la scheda nel VMEBus, è possibile eseguire delle impostazioni hardware aprendo, o chiudendo opportuni jumper.

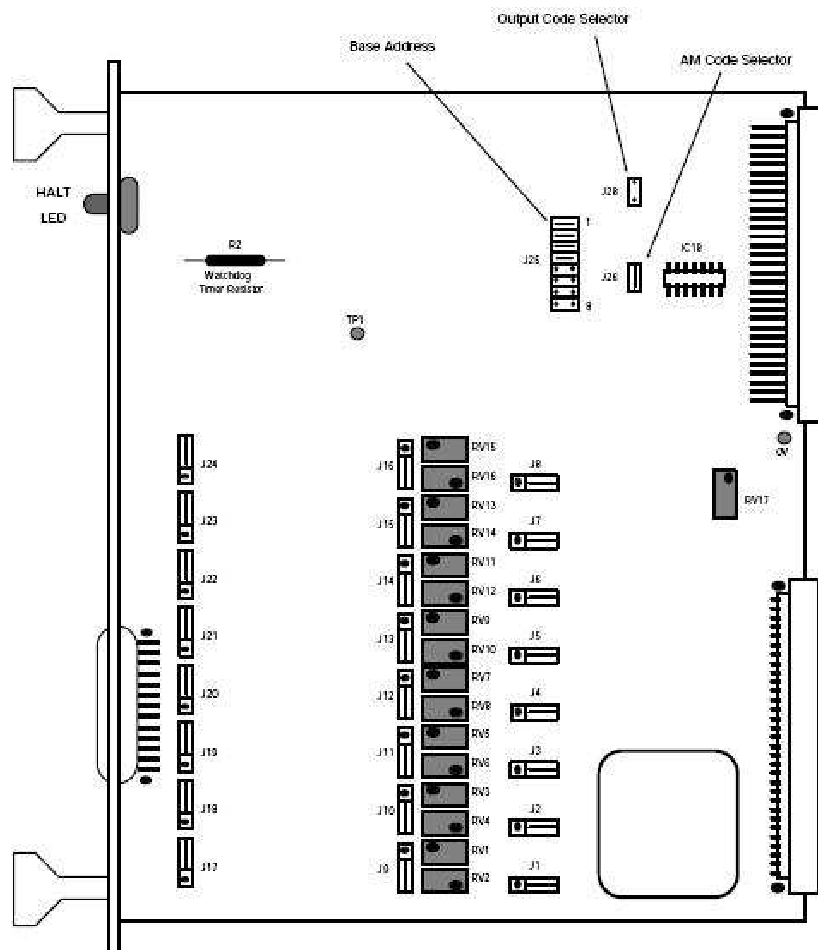

Figura 21: Posizione dei jumper sul dispositivo MPV955

Lo schema rappresentato nella Figura 21 rappresenta la posizione fisica dei jumper sul dispositivo MPV955, ognuno dei quali ha una particolare funzione:

- *Range di tensioni di output* (J1-J8): semplicemente c'è un jumper per ogni canale d'uscita, ed è possibile modificare la sua posizione per avere tensioni nel range +10V/±10V oppure +5V/±5V.
- *Range unipolare/bipolare* (J9-J16): per ogni canale, è possibile decidere se le tensioni in uscita devono essere bipolari (-5÷+5V, -10÷+10V) oppure unipolari (0÷5V, 0÷10V).

- *Base address* (J25): la memoria del dispositivo sul bus parte da un determinato indirizzo chiamato "base address". Questa impostazione è fondamentale poiché fa sì che più dispositivi non collidano usando lo stesso indirizzo. MPV955 permette di impostare il "base address" con un valore compreso tra F00000H e FF0000H.
- *Address modifier code* (J26): è possibile attivare la gestione dei codici "Address Modifier", presentati nel capitolo 3. Sul dispositivo essi sono memorizzati su una PROM e valgono 3DH e 39H (Figura 17).
- *Selezione dei codici di output* (J28): nella memoria interna del dispositivo i segnali digitali sono rappresentati, in maniera discreta, da valori binari a 16 bit. La rappresentazione binaria di un numero intero all'interno di un calcolatore può essere fatta in vari modi; il nostro dispositivo prevede la rappresentazione di questi valori in complemento a uno, in complemento a due, o con un certo offset binario.

4.3 Impostazioni software del dispositivo MPV955

Oltre alle impostazioni hardware, da fare al momento dell'installazione del dispositivo nel VMEBus, è possibile configurare la scheda MPV955 via software manipolando opportunamente il valore dei suoi registri.

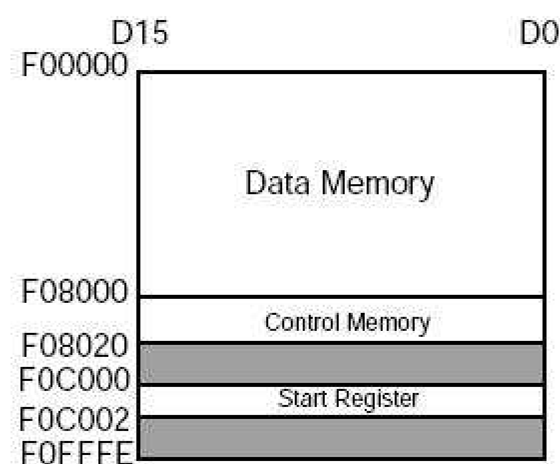

Figura 22: Organizzazione della memoria

4.3.1 Organizzazione della memoria interna

Gli indirizzi della memoria interna e dei registri nel dispositivo sono assegnati come illustrato in Figura 22.

Nel bus VME quest'area di memoria è collocata a partire dal "base address", di default uguale a F00000H. Essa è divisa principalmente in tre parti:

- Memoria dati (data memory): destinata a contenere tutti i dati da convertire in segnali elettrici-analogici.
- Memoria di controllo (control memory): questo segmento di memoria è costituito da tutta una serie di registri utili al controllo del dispositivo.
- Registro di attuazione (Start Register): effettuando un operazione di lettura i di scrittura in questo registro il dispositivo entrerà in funzione cominciando immediatamente la conversione dei dati.

La parte di memoria più interessante da esaminare è quella concernente il controllo del dispositivo (memoria di controllo), della quale vediamo un ingrandimento in Figura 23.

Si nota che ogni registro dell'area di controllo ha due indirizzi, appartenenti rispettivamente all'area 1 e l'area 2. Eseguendo un'operazione di scrittura nell'area 1, l'operazione d'output corrente sarà automaticamente fermata, mentre un accesso in scrittura nell'area 2 non ferma automaticamente l'operazione d'output correntemente in esecuzione. Preferire un'area di memoria, al posto dell'altra, dipende da come l'utente intende utilizzare il dispositivo.

Di seguito analizzeremo i compiti d'ogni registro raffigurato in Figura 23, ricordando che i bit colorati di grigio sono inutilizzati e lasciati a disposizione dell'azienda per sviluppi futuri del dispositivo.

	D15 D14 D13 … D8 D7 … D0	
F08000H	Control Status	F08010H
F08002H	Start Address Register	F08012H
F08004H	Stop Address Register	F08014H
F08006H	Interrupt Control Register	F08016H
F08008H	Rate Timer Control	F08018H
F0800AH	Timeout Control	F0801AH
F0800CH	DAC DIS	F0801CH

Figura 23: Contenuto della memoria di controllo

4.3.2 Registro "Control/Status"

Quasi tutta la configurazione software del dispositivo MPV955 è racchiusa nei 16 bit (raffigurati in Figura 24) costituenti questo registro. Essi sono di due tipi: i primi 8 (D0-D7) sono bit di controllo e sono impostati dall'utente, mentre tutti gli altri (D8-D15) sono bit di stato ed è utile leggerli per ottenere informazioni sullo stato corrente del dispositivo.

D15	D14	D13	D12	D11	D10	D9	D8
N/A				OVER SAMP	CYCFIN	TIMEOUT	HALT

D7	D6	D5	D4	D3	D2	D1	D0
N/A	CHANNEL SELECT			TIMEOUT ENABLE	CONT/ ONE	NORMAL/ EVENT	INT/ EXT

Figura 24: Bit componenti il registro Control/Status

La perfetta conoscenza delle funzionalità di questo registro è d'essenziale importanza per la scrittura di un driver adatto al dispositivo; per questo motivo esamineremo ognuno dei suoi bit:

- D0: se impostato a 1 abilita il clock esterno, altrimenti quello interno;

- D1: se impostato a 1, mentre D0 vale 0, fa partire il clock interno solo quando al dispositivo arriva un segnale proveniente dall'esterno. Non ha senso manipolare questo bit se è abilitato il clock esterno;
- D2: decide se il dispositivo deve convertire i dati in maniera continua (se impostato a 0), oppure in modo one-shot (se impostato a 1);
- D3: abilita il timeout quando è impostato a 0. Chiariremo meglio cosa vuol dire nel corso di questo capitolo;
- D4-D6: abilita o disabilita, in modo contiguo, i canali di uscita. Se i tre bit valgono 000 è abilitato solo il primo canale, mentre se valgono 111 sono attivi tutti i canali;
- D7: bit inutilizzato e lasciato a disposizione per sviluppi futuri; scrivere o leggere questo bit non ha effetti sul dispositivo.

- D8: Questo è il primo dei bit di stato. Esso vale 1 quando il dispositivo è attivo;
- D9: Vale 1 quando è attivo il watchdog timer (caratteristica discussa a pagina 55);
- D10: Vale 1 quando è stato completato un intero ciclo di output, mentre vale 0 durante una conversione;
- D11: Vale 1 quando il dispositivo è messo sotto sforzo, e il clock funziona ad una frequenza troppo elevata;
- D12-D15: bit inutilizzati e lasciati a disposizione per sviluppi futuri; scrivere o leggere questi bit non ha effetti sul dispositivo.

4.3.3 Registri "Start/Stop Address"

"Start Address Register" e "Stop Address Register", entrambi registri da 14 bit, contengono rispettivamente l'indirizzo d'inizio e di fine dai quali prelevare i dati da convertire. All'interno di questi registri vanno memorizzati solo indirizzi relativi e non assoluti: questo è dato dal fatto che la memoria del dispositivo può essere mappata con differenti indirizzi modificando l'impostazione del "base address".

4.3.4 Registro “Interrupt Control”

Il dispositivo MPV955, essendo compatibile elettronicamente e meccanicamente con il bus VME, è in grado di utilizzare tutti e sette i livelli d’interrupt previsti mediante un apposito circuito. Il registro “Interrupt Control” (raffigurato in Figura 25), è utile per configurare gli interrupt.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
	IOS	ICY	ITI		INTERRUPT PRIORITY			STATUS/ID BYTE							

Figura 25: Struttura del registro per gli interrupt

Per la scrittura di un driver adatto al dispositivo è necessario conoscere in modo dettagliato il ruolo di questo registro; per questo motivo esamineremo ognuno dei suoi bit:

- D0-D7: è un byte destinato ad essere inviato alla routine di gestione dell’interrupt del VMEBus;
- D8-D10: seleziona uno dei sette livelli di interrupt;
- D11: bit inutilizzato e lasciato a disposizione per sviluppi futuri; scrivere o leggere questo bit non ha effetti sul dispositivo;
- D12: se impostato ad 1 il dispositivo lancia un interrupt in caso di sovracampionamento;
- D13: se impostato ad 1 il dispositivo lancia un interrupt quando il dispositivo va in timeout;
- D14: se impostato ad 1 il dispositivo lancia un interrupt appena finito un ciclo di conversione;
- D15: bit inutilizzato e lasciato a disposizione per sviluppi futuri; scrivere o leggere questo bit non ha effetti sul dispositivo.

4.3.5 Registro “Rate Timer Control”

Il dispositivo MPV955 ha la capacità di controllare la velocità di conversione dei dati anche quando è abilitato il clock interno. Questo è possibile tramite la manipolazione del registro, a sola scrittura, chiamato “rate timer control”. Esso non è altro che un contatore ad 8 bit capace di emettere un impulso ogni volta che raggiunge il valore FFH; quindi più piccolo è il valore memorizzato all’interno del registro, meno veloce sarà il dispositivo a convertire i dati.

Tramite questo registro si può programmare la velocità del DAC nel range da 0 a 127,5µs con incrementi di 500ns. Banalmente la formula per calcolare la velocità di output, dato un certo valore del registro, è la seguente:

Output = (255 – “valore del registro”) × 500ns

4.3.6 Registro “Timeout Control”

Per tutelare l’integrità di eventuali dispositivi delicati collegati al DAC, abbiamo a disposizione un’interessante funzione della scheda MPV955: il “watchdog timer”. Quest’ultimo è in grado di forzare a 0V tutti i canali analogici del DAC se allo scadere di un certo intervallo di tempo non si è fatto nessun accesso al registro di attuazione (start register).

Per funzionare il “watchdog timer” si appoggia sul registro “timeout control” che dal punto di vista hardware è in sostanza identico al registro “rate timer control”. Quindi il “watchdog timer” entra in funzione quando il valore del registro contatore arriva a FFH.

Questa volta siamo in grado di programmare l’intervallo di tempo a incrementi di 1,3s; di conseguenza la formula per calcolare la velocità dell’intervallo è:

Output = (255 – “valore del registro”) × 1,3s

Il registro “timeout control” è molto legato al registro d’attuazione. In effetti, quando scriviamo un valore in quest’ultimo registro, esso è ricopiato nel registro di controllo del timeout e in seguito viene fatto partire il ciclo d’output.

4.3.7 Registro “DAC Disable”

Questo registro, formato da un solo bit, è il più semplice del dispositivo: se impostato a 0 i canali d’uscita sono abilitati a funzionare, mentre se è impostato a 1 le uscite analogiche del DAC sono tutte disabilitate.

4.4 Esempio d’uso del dispositivo MPV955

Le seguenti sette operazioni sono essenziali per compiere una semplice operazione di pulizia dei canali:

1. Registrare nelle prime 16 locazioni della memoria dati il valore esadecimale pari a 0V;
2. Configurare, tramite il registro di controllo:
   a. modalità one-shot,
   b. clock interno,
   c. abilitare tutti i canali del DAC;
3. Scrivere nello “Start Address Register” e nello “Stop Address Register” rispettivamente i valori 0000H e 000FH;
4. Compiere un accesso al registro di attuazione per far iniziare il ciclo di conversione;
5. Accettarsi, controllando i valori di stato del registro di controllo, che il ciclo di conversione sia completato;
6. Abilitare il DAC.

Quanto appena descritto è solo un semplice esempio, a scopo illustrativo, che non sfrutta a pieno le caratteristiche del dispositivo MPV955.

4.5 Dispositivo VGD5

Il dispositivo VGD5 è un convertitore analogico digitale a 32 canali, utile al nostro sistema per acquisire i dati dall'esterno. Prodotta da Pentland System, non s'interfaccia direttamente sul bus VME, ma si appoggia sulla scheda madre VGX, prodotta dalla stessa azienda. Le principali caratteristiche del dispositivo VGD5 sono:

- Indirizzamento a 16 bit;
- Clock esterno;
- Memoria da 4KByte;
- Risoluzione dei dati a 16 bit;
- Massimo range di input ±20V;
- Sette livelli di "interrupt" gestibili via software;
- Velocità di trasferimento dati sul VMEBus 15MByte/s;
- 32 canali di input.

Inoltre il dispositivo VGD5 è in grado di acquisire i dati in due modi:

- "transient": acquisisce un nuovo campione solo se è stato già convertito il campione precedente;
- "continua": acquisisce continuamente nuovi campioni rispettando la velocità del clock.

Per quanto riguarda il dispositivo VGD5, e la scheda madre VGX, non si scenderà ulteriormente nei dettagli (rif. [7] e [8]).

Capitolo 5

Ingegnerizzazione del sistema

Per introdurre stabilità, controllo e organizzazione in un'attività tendenzialmente caotica è necessario una strategia di sviluppo chiamata "paradigma di ingegneria del software" o "modello di processo" (rif. [10] e [11]).

5.1 Paradigma di ingegneria del software

Lo sviluppo di software è rappresentato come un ciclo di risoluzione di problemi (Figura 26), articolato in quattro fasi distinte: lo status quo, la definizione del problema, lo sviluppo tecnico, e l'integrazione della soluzione.

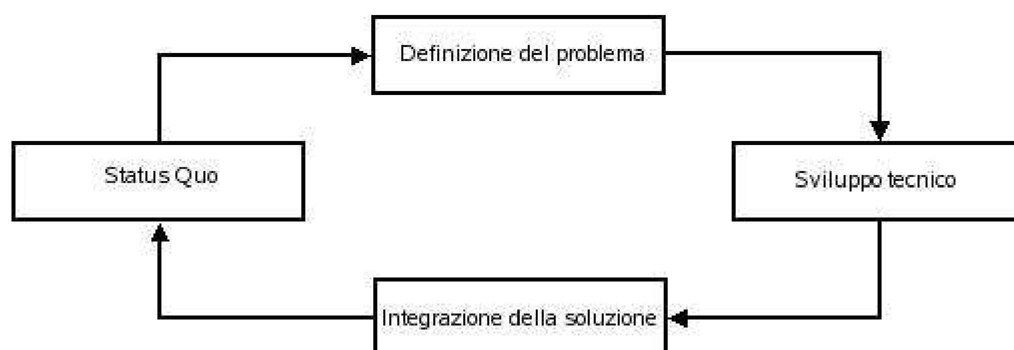

Figura 26: Le fasi di un ciclo di risoluzione

Lo status quo, rappresenta ”lo stato presente”; la definizione del problema individua lo specifico problema da risolvere; lo sviluppo tecnico risolve il problema per mezzo di una certa tecnologia; l’integrazione della soluzione consegna i risultati (ad esempio documenti, programmi, dati) al committente. La scelta del modello o paradigma dipende dalla natura del progetto e dell’applicazione, dai metodi e strumenti che si vogliono utilizzare e dai controlli e prodotti richiesti. Nel nostro caso il modello che meglio si adatta alla progettazione del sistema è quello denominato *modello incrementale*. Questo modello consiste di più stadi, ed ogni stadio è costituito, a sua volta, da un insieme di attività scalate nel tempo (Figura 27).

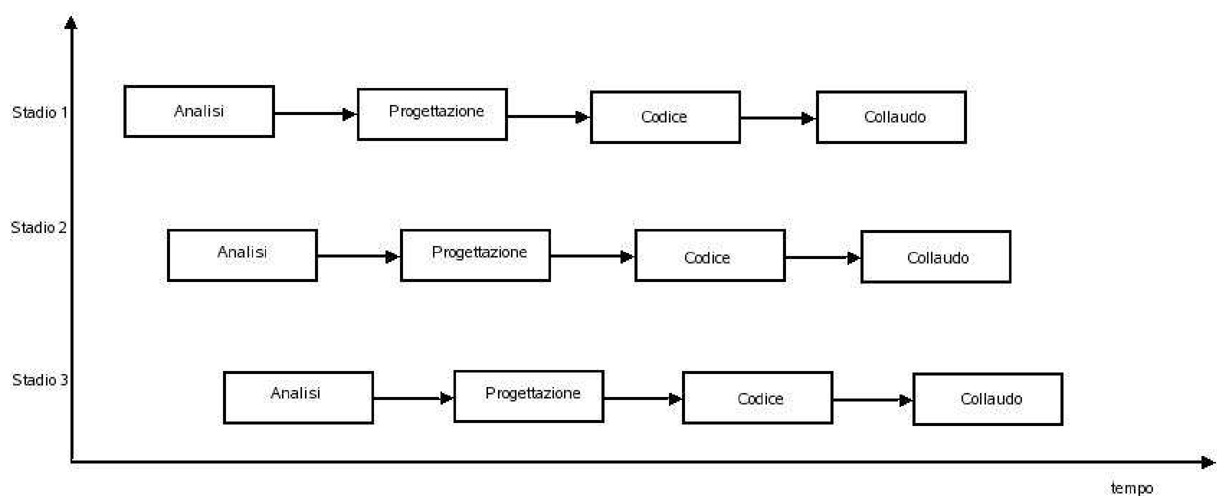

Figura 27: Il modello incrementale

Ogni attività ha un compito mirato e specifico:

- **analisi dei requisiti del software**: la raccolta dei requisiti permette di comprendere il dominio delle informazioni del software, così come la funzionalità, il comportamento, le prestazioni e le interfacce richieste;
- **progettazione**: il processo di progettazione traduce i requisiti in una rappresentazione del software della quale si possa valutare la qualità prima che cominci la stesura del codice;

- **generazione codice**: il progetto deve essere tradotto in codice;
- **collaudo**: una volta generato il codice, comincia il collaudo dei programmi che si concentra essenzialmente, sugli aspetti logici interni del software, al fine di garantire che tutte le istruzioni siano provate, e sulle funzionalità esterne al fine di scoprire eventuali errori, e accertarsi che a fronte di dati di input specifici, vengano prodotti i risultati previsti.

Nel modello incrementale il primo stadio consiste nel rilascio di un prodotto che soddisfa i requisiti fondamentali, tralasciando diverse caratteristiche supplementari. Questo prodotto viene saggiato dal cliente. In seguito alla sua valutazione, si stende un piano per lo stadio successivo. Tale piano deve curarsi di due aspetti: le modifiche al prodotto tese a soddisfare meglio le esigenze del cliente e l'inserimento di nuove funzioni. Il processo si ripete a mano a mano che si completano i vari stadi, fino a giungere al prodotto finale. Siccome i requisiti del sistema da progettare non sono chiari fin dall'inizio, questo modello ben si adatta ai nostri scopi, perché permette di costruire un prodotto funzionante ad ogni stadio, con la possibilità di aggiunte e modifiche in corso d'opera. Nelle sezioni seguenti non saranno mostrati tutti gli stadi necessari al completamento del sistema, ma solo lo stadio finale.

5.2 Raccolta dei requisiti

5.2.1 Requisiti funzionali

Nome requisito	Descrizione
REQ-FUNZ 1	Il sistema dovrà essere composto da un'unità di controllo ed un'unità di calcolo
REQ-FUNZ 2	L'unità di controllo deve essere una CPU su bus VME con sistema operativo LynxOS 4.0 o superiore
REQ-FUNZ 3	L'unità di controllo del sistema deve acquisire dati analogici da un ADC, tramite il VME su 32 canali indipendenti
REQ-FUNZ 4	L'ADC dovrà essere comandato da un driver del sistema operativo

REQ-FUNZ 5	Il sistema deve inviare tali dati ad un'unità esterna per l'elaborazione di filtri digitali; tale unità è parte costituente del sistema e verrà denominata unità di calcolo
REQ-FUNZ 6	La trasmissione dei dati a/e dall'unità di calcolo dovrà essere effettuata tramite un canale di trasmissione dati comune ad entrambe le unità di calcolo e di controllo
REQ-FUNZ 7	Dovrà essere prevista l'utilizzazione di un differente canale di comunicazione tra le unità quando si renderà disponibile
REQ-FUNZ 8	L'unità di controllo del sistema dovrà ricevere i dati elaborati dall'unità di calcolo ed inviarli ad una o più unità di attuazione (DAC)
REQ-FUNZ 9	Ciascun canale acquisito, elaborato e ri-inviato dovrà essere inviato ad un differente canale del DAC
REQ-FUNZ 10	Il DAC dovrà essere comandato dall'unità di controllo tramite un driver del sistema operativo

Tabella 4: Requisiti funzionali

5.2.2 Requisiti di interfaccia

Nome requisito	Descrizione
REQ-INTF 1	I driver di gestione delle periferiche dell'unità di controllo (ADC e DAC) devono fornire al livello applicativo alcune delle interfacce standard di un sistema operativo UNIX (*open, close, ioctl*)
REQ-INTF 2	Ciascun driver deve permettere di accedere alla scheda corrispondente come un normale device di sistema, eseguendo cioè una chiamata alla funzione *open*
REQ-INTF 3	L'interfaccia tra le periferiche dell'unità di controllo e le applicazioni dovrà avvenire tramite chiamate alle funzioni di sistema *read* e *write* per l'acquisizione e l'attuazione
REQ-INTF 4	La gestione delle funzionalità delle periferiche dovrà avvenire tramite chiamate alla funzione di sistema *ioctl* del device corrispondente
REQ-INTF 5	La scelta del canale di trasmissione tra l'unità di controllo e l'unità di calcolo dovrà poter essere impostata dall'utente tramite una flag dell'applicativo di interfaccia su entrambe le unità e dovrà prevedere un canale di default (per es. UDP/IP)
REQ-INTF 6	Deve essere possibile gestire l'intero sistema da remoto da un interfaccia grafica.

Tabella 5: Requisiti di interfaccia

5.2.3 Requisiti di sicurezza

Nome requisito	Descrizione
REQ-SEC 1	Ogni canale di output del DAC deve fornire in uscita una tensione pari a 0V quando non è sotto il controllo di un'applicazione utente.

Tabella 6: Requisiti di sicurezza

5.2.4 Requisiti prestazionali

Nome requisito	Descrizione
REQ-PREST 1	L'acquisizione dei dati dovrà avvenire con frequenze di campionamento fino a 20 kHz
REQ-PREST 2	La latenza tra la lettura dall'ADC e l'effettiva uscita delle tensioni dal DAC dovrà essere limitata dal tempo di round-trip dei dati sul canale di trasmissione
REQ-PREST 3	Il tempo di round-trip del canale di trasmissione dovrà essere minore o uguale a 250μs
REQ-PREST 4	Le tensioni in uscita dal DAC dovranno avere lo stesso ordinamento temporale dei dati corrispondenti dell'ADC

Tabella 7: Requisiti prestazionali

5.2.5 Requisiti implementativi o di contesto

Nome requisito	Descrizione
REQ-CTST 1	I driver per il sistema operativo LynxOS 4.0.0 dovranno supportare le schede Thales Computers VMPC6a o superiore
REQ-CTST 2	I driver, i programmi di interfaccia dovranno essere elaborati in linguaggio ANSI C e rispettare le specifiche POSIX 1003.1, 1b e 1c
REQ-CTST 3	L'installazione e la disinstallazione di ciascun driver per il S.O. LynxOS 4.0 dovrà essere effettuata tramite uno shell script *install.<driver>* ed *uninstall.<driver>*
REQ-CTST 4	Lo script *install.<driver>* richiederà all'utente se il driver debba essere statico o dinamico e, ove necessario eseguirà tutti i passi necessari a partire dai file sorgente per ottenere un S.O. con il driver installato e funzionante
REQ-CTST 5	L'interfaccia grafica di comando su rete deve essere scritta in linguaggio Java, e quindi indipendente dalla piattaforma.

Tabella 8: Requisiti implementativi o di contesto

5.2.6 Requisiti particolari di test e collaudo

Nome requisito	Descrizione
REQ-PTC 1	Il sistema dovrà comprendere una suite di test ed un manuale della stessa contenente la matrice di copertura dei requisiti
REQ-PTC 2	La suite di test dovrà implementare almeno un filtro e controllarne il funzionamento

Tabella 9: Requisiti particolari di test e collaudo

5.3 Analisi architetturale e progettazione del driver

In questa sezione sarà illustrata la collocazione e la progettazione del driver, mentre l'analisi e la progettazione del sistema di comunicazione saranno illustrate nella sezione successiva.

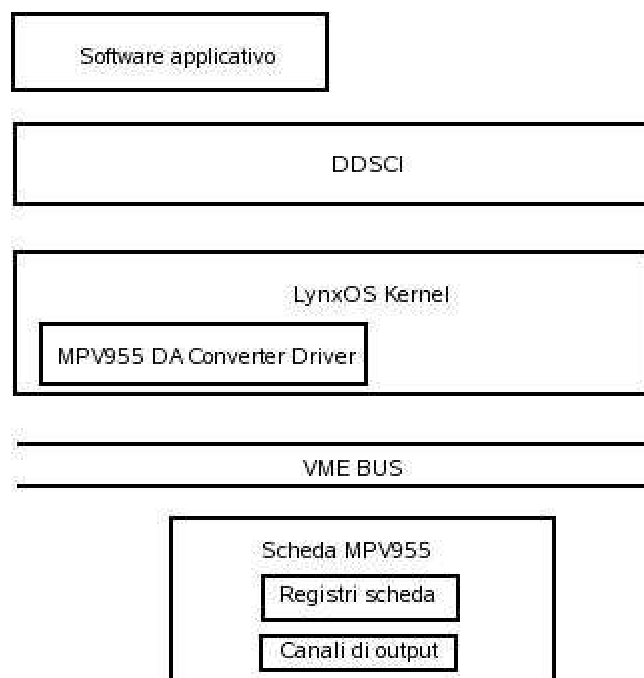

Figura 28: Collocazione operativa del driver

5.3.1 Collocazione operativa

Tramite l'MPV955 Digital-Analogic Converter LynxOS Device Driver, un applicativo può avere accesso alla scheda MPV955 DA Converter secondo lo schema rappresentato in Figura 28.

L'applicativo accede al device driver tramite il sottosistema di interfaccia I/O DDSCI (*Device Driver System Call Interface*) costituita dalle chiamate di sistema *open, ioctl, close*. Il device driver della scheda MPV955 invia, sul Bus VME, i comandi per pilotare opportunamente il dispositivo.

5.3.2 Interfacce esterne

L'interfaccia tra LynxOS Kernel e l'MPV955 DA Converter LynxOS Device Driver avviene tramite le entry point riportate nella Tabella 10.

5.3.3 Moduli del driver

mpv955_install()

L'entry point *mpv955_install()* viene richiamata dal kernel LynxOS durante la fase di bootstrap se l'installazione è statica o in qualsiasi altro momento se l'installazione è dinamica, e provvede all'inizializzazione della MPV955 DA Converter e delle risorse necessarie eseguendo le seguenti azioni:

- allocazione ed inizializzazione della struttura statica del device driver;
- apertura canali VME e mappatura della scheda;
- test di presenza della scheda all'indirizzo selezionato;
- inizializzazione della memoria e dei registri della scheda.

Entry Point	Descrizione
mpv955_install()	Invocata dal kernel (durante il boot-strap se l'installazione è statica o in qualsiasi altro momento se l'installazione è dinamica) per allocare le risorse necessarie all'utilizzo dell'interfaccia e inizializzare la scheda MPV955 DA Converter
mpv955_uninstall()	Invocata dal kernel per rilasciare la scheda MPV955 DA Converter e liberare le risorse necessarie all'utilizzo dell'interfaccia
mpv955_open()	Invocata dal kernel per accedere alla scheda MPV955 DA Converter. Il kernel invoca questa entry point quando un SW applicativo usa la system call *open()* relativamente al device associato alla scheda
mpv955_close()	Invocata dal kernel per chiudere l'accesso alla scheda MPV955 DA Converter. Il kernel invoca questo entry point quando un SW applicativo usa la system call *close()* relativamente al device associato alla MPV955 DA Converter
mpv955_write()	Invocata dal kernel per scrivere nel buffer dati della scheda MPV955 DA Converter. Il kernel invoca questo entry point quando un SW applicativo usa la system call *write()* relativamente al device associato alla MPV955 DA Converter
mpv955_ioctl()	Invocata dal kernel per l'esecuzione di varie operazioni relative alla configurazione della scheda MPV955 DA Converter. Il kernel invoca questo entry point quando il SW applicativo usa la system call *ioctl()* relativamente al device associato alla scheda.

Tabella 10: Entry Point per MPV955 DA Converter

Nel caso di errore riscontrato in uno dei seguenti passi, vengono rilasciate tutte le risorse precedentemente allocate e ritornato un codice d'errore.

mpv955_uninstall()

L'entry point *mpv955_uninstall()* viene invocato dal kernel LynxOS per disinstallare il MPV955 DA Converter Device Driver. L'entry point provvede a liberare tutte le risorse allocate dalla *mpv955_install()*.

mpv955_open()

Entry point per abilitare l'accesso al device relativo alla scheda MPV955 DA Converter: viene invocato dal kernel nel momento in cui un'applicazione usa la system call *open()* relativamente al device della scheda. La funzione abilita l'interazione con la scheda ed esegue le seguenti azioni:

- abilitazione del dispositivo;

Inoltre se è il primo task ad usare il dispositivo vengono eseguite anche le seguenti operazioni:

- inizializzazione del CONTROL-STATUS Register;
- inizializzazione RATE-TIMER-CONTROL Register;
- inizializzazione INTERRUPT-CONTROL Register;
- inizializzazione START e STOP ADDRESS Register;
- abilitazione dell'output mediante il bit DAC DISABLE.

Quando il primo task effettua l'accesso al dispositivo, lo trova con la seguente configurazione di default:

- CONTINUOUS MODE (quindi timeout disabilitato),
- Rate Timer Control inizializzato a 0xFFFB (massimo valore possibile),
- Internal Trigger abilitato,
- Tutti i canali di output selezionati,
- Output abilitato (bit DAC DISABLE),
- Operazioni di output già in corso con valore pari a 0 Volt; non è necessario dare il comando ioctl START per avviare la conversione.

mpv955_close()

Entry point per disabilitare le operazioni e l'accesso al device associato alla scheda MPV955 DA Converter. Viene invocata dal kernel nel momento in cui un'applicazione chiama la system call *close()* relativamente ad uno dei minor device della scheda.

Quando non ci sono più task che usano il dispositivo, questa funzione provvede a disabilitare l'output dei canali. Ricordiamo che anche se viene disabilitato l'output, il

dispositivo tiene memorizzato l'ultimo valore scritto in un determinato canale, quindi se viene riabilitato l'output (con una chiamata a *mpv955_open()*), il canale usato la volta precedente conterrà ancora il vecchio valore. Per evitare questo prima di chiudere il dispositivo è necessario pulire il canale scrivendoci il valore pari a 0 Volt.

mpv955_write()

Entry point per la scrittura di un buffer dati nella memoria del device associato alla scheda MPV955 DA Converter. Viene invocata dal kernel nel momento in cui un'applicazione chiama la system call *write()* relativamente ad uno dei minor device della scheda.

Tutte le scritture sono effettuate in modo atomico, cioè un dato alla volta. Di conseguenza il parametro count della chiamata di sistema *write()* non è utilizzato per specificare la dimensione del buffer, ma per selezionare il numero di canali analogici contigui da usare. Esso può assumere valori da 0 a 7, in base al numero di canali scelti.

mpv955_ioctl()

Entry point per effettuare le operazioni sui registri accessibili della scheda MPV955 DA Converter. Questa funzione ha come parametri di ingresso significativi uno fra i seguenti comandi:

START	ENABLE_EVENT_TRIGGER
STOP	GET_TIMEOUT_STATUS
ENABLE_DAC	GET_OUTPUT_MODE
DISABLE_DAC	GET_TRIGGERING_MODE
ENABLE_TIMEOUT	SET_RATE_TIMER
DISABLE_TIMEOUT	GET_CYCFIN
SET_CONTINUOUS	GET_OVER_SAMP
SET_ONESHOT	GET_NCHAN
ENABLE_INT_TRIGGER	SET_NCHAN
ENABLE_EXT_TRIGGER	SET_ZERO_VOLTS

Tabella 11: Comandi per la system call ioctl()

Di seguito si riporta una spiegazione delle possibili funzioni della chiamata di sistema *ioctl()*:

START: Funzione per avviare un operazione di output. Quando si accede al dispositivo per la prima volta non è necessario utilizzare questo comando perché le operazioni di output già sono abilitate in CONTINUOUS MODE.
Questo comando è fondamentale quando ci si trova nella modalità ONE-SHOT, durante la quale deve esserci un accesso regolare allo START REGISTER.

STOP: Funzione per arrestare le operazioni di output.

DISABLE_DAC: Funzione per disattivare la scheda MPV955 DA Converter in qualsiasi istante. Se già disabilitata rimane in quello stato.

ENABLE_DAC: Funzione per abilitare la scheda MPV955 DA Converter in qualsiasi istante. Se già abilitata rimane in quello stato.

ENABLE_TIMEOUT: Funzione per abilitare il ”time out”. Di default questo valore è 1.3s allo scadere del quale viene disabilitato l’output. L’argomento di questo comando è il valore desiderato del ”time out”.
Non ha senso abilitare il timeout se la scheda non si trova in modalità ONE-SHOT. Per questo se la funzione (ENABLE_TIMEOUT) viene invocata in altre modalità, la *ioctl()* restituirà un indicatore di errore.

DISABLE_TIMEOUT: Funzione per disabilitare il ”time out”.

SET_CONTINUOUS: Funzione per abilitare il modo di output CONTINUOUS.

SET_ONESHOT: Funzione per abilitare il modo di output ONE-SHOT.

ENABLE_INT_TRIGGER: Funzione per abilitare il clock interno.

ENABLE_EXT_TRIGGER: Funzione per abilitare il clock esterno.

ENABLE_EVENT_TRIGGER: Funzione per abilitare l'EVENT TRIGGER. Questa modalità consiste nell'usare il clock interno, combinato al clock esterno, il quale invierà il primo impulso e gli altri saranno gestiti dal clock interno.

GET_TIMEOUT_STATUS: Funzione per leggere il valore del bit TIMEOUT del registro CONTROL/STATUS. Il valore restituito dalla funzione è registrato nel parametro di input/output *unsigned short *arg*. Esso sarà 1 se il dispositivo è andato in timeout, 0 altrimenti.

GET_OUTPUT_MODE: Funzione per ottenere informazioni sulla modalità di output dei dati; CONTINUOUS o ONESHOT MODE. Il valore restituito dalla funzione è registrato nel parametro di input/output *unsigned short *arg*. Esso sarà 1 se il dispositivo si trova in modalità ONESHOT, mentre sarà 0 se è attiva la modalità CONTINUOUS.

GET_TRIGGERING_MODE: Funzione per ottenere informazioni sulla modalità di TRIGGERING. Indica se il clock è esterno, interno oppure ad evento. Il valore restituito dalla funzione è registrato nel parametro di input/output *unsigned short *arg*. Esso sarà:

- 1: Abilitato externel trigger,
- 2: Abilitato internal trigger,
- 3: Abilitato event trigger,
- 4: E' uno stato incoerente; ovvero una situazione che non dovrebbe mai verificarsi, in tal caso conviene riavviare il sistema.

SET_RATE_TIMER: Funzione per modificare la velocità di output (ovvero la velocità del clock interno). L'argomento deve essere un valore di tipo *unsigned short* contenente la velocità di output. Non ha senso settare il rate timer se si sta utilizzando un clock esterno.

GET_CYCFIN: Funzione per leggere il valore del bit CYCFIN del registro CONTROL/STATUS. Indica se un ciclo di output è terminato oppure no. Il valore restituito dalla funzione è registrato nel parametro di input/output *unsigned short *arg*. Esso sarà 1 se il dispositivo ha terminato il ciclo di output, mentre sarà 0 se il ciclo di output è ancora in corso. Se si è in CONTINUOUS
MODE, il ciclo di output non avrà mai termine: in questo caso non ha senso leggere il valore di questo bit.

GET_OVER_SAMP: Funzione che permette di sapere se i dati di input vengono inviati alla scheda più velocemente dell'output rate. Il valore restituito dalla funzione è registrato nel parametro di input/output *unsigned short *arg*. Esso sarà 1 se la scheda non riesce ad effettuare l'output di tutti i dati, mentre sarà 0 se lavora regolarmente.

GET_NCHAN: Funzione per ottenere il numero di canali correntemente attivati. Il valore restituito dalla funzione è registrato nel parametro di input/output *unsigned short *arg*. Esso sarà uguale al numero di canali selezionati; può assumere valori compresi tra 0 e 7. Ad esempio:

- 0: attivo solo il canale 0;
- 1: attivi i canali 0, 1;
- 7: attivi i canali 0, 1, 2, 3, 4, 5, 6, 7.

SET_NCHAN: Funzione per settare il numero di canali di output attivi. L'argomento è un valore numerico di tipo *unsigned short* compreso tra 0 e 7. Ad esempio:

- 0 : attiva solo il canale 0;
- 1 : attiva i canali 0, 1;
- 7 : attiva i canali 0, 1, 2, 3, 4, 5, 6, 7.

SET_ZEROVOLTS: Funzione per ottenere il valore esadecimale corrispondente a ZERO_VOLTS. Il valore restituito dalla funzione è registrato nel parametro di input/output *unsigned short *arg*. Questo comando è utile se si vuole pulire un canale dopo averlo utilizzato; infatti il dispositivo dopo la chiusura non pulisce i canali, demandando tale compito all'applicazione utente. Questa scelta di progettazione consente a più task di accedere al dispositivo senza provocare conflitti tra di loro.

Parametro	**Descrizione**
*info	Puntatore di tipo che contiene l'indirizzo della struttura Device Information del device driver
area_cm	Variabile di tipo *int* contenente un valore 0 oppure 1. Esso è 0 se si sta lavorando nell'area 1 dei registri, mentre è 1 se si sta lavorando nell'area 2 dei registri
*registers[MAX_AREA_CM]	Vettore di due elementi (MAX_AREA_CM=2). Esso contiene i puntatori rispettivamente all'area 1 o all'area 2 dei registri del dispositivo
*start_register	Puntatore contenente l'indirizzo dello START register del dispositivo
*data_memory	Puntatore contenente l'indirizzo base dell'area dati (buffer) presente nel dispositivo
*p_a24_phys_addr	Puntatore che contiene l'indirizzo fisico VME A24. Si conserva questo dato per rilasciare il bus nella funzione entry point *mpv955_uninstall()*
zero_volts	Variabile di tipo contenente il valore corrispondente a zero volt, usato per inizializzare i canali analogici del dispositivo
rate_timer	Variabile contenente il valore del registro RATE TIMER CONTROL
timeout_value	Variabile contenente il valore del registro TIMEOUT CONTROL
primo	Flag usata nella funzione *mpv955_open()* per evitare le race-condition

Tabella 12: Descrizione struttura statica

5.3.4 Interfacce interne

Le interfacce interne del driver sono concentrate nella struttura statica del driver stesso. La struttura, la cui tipologia è definita nel file *mpv955drv.h*, è allocata in fase di installazione del driver e contiene i dati mostrati nella Tabella 12.

5.4 Analisi architetturale e progettazione sistema di trasmissione

In questa sezione sarà illustrata l'analisi e la progettazione del sistema di trasmissione tra l'unità di calcolo e l'unità di controllo. L'interfaccia di comando su rete in grado di controllare l'intero sistema da remoto è introdotta nella prossima sezione.

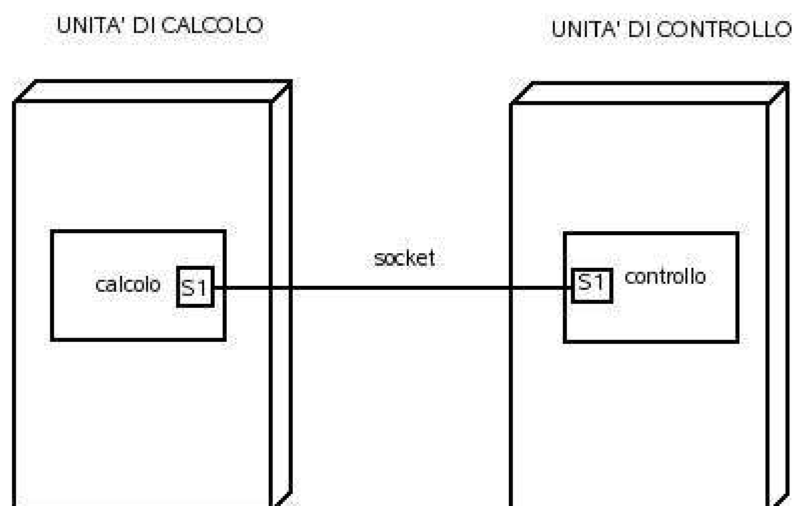

Figura 29: Diagramma dei componenti e dei nodi

5.4.1 Architettura del sistema di comunicazione

Come accennato nel primo capitolo sono state sviluppate diverse soluzioni, per la comunicazione tra le unità di calcolo e di controllo, ma quella effettivamente utilizzata, prevede l'impiego, dal punto di vista implementativo di una socket UDP e di due thread, pertanto si riporta l'ingegnerizzazione di questa soluzione. Le altre differiscono da

quest'ultima per le scelte implementative che saranno brevemente descritte nel capitolo 7, solo per poter giustificare i risultati ottenuti nei test.
In Figura 29 è riportato il diagramma dei nodi del sistema di trasmissione e le applicazioni in esecuzione su di essi, della soluzione ingegnerizzata.

5.4.2 Interfacce interne comuni

La definizione delle strutture dati che i due nodi del sistema hanno in comune è contenuta nel file *trasmissione.h*, e contiene tutto il necessario per implementare la comunicazione tra le due unità. Tutte le funzioni definite in questa sezione prevedono, tramite una flag, la scelta del protocollo di comunicazione (per ciò che riguarda questa tesi è stato implementato solo il protocollo UDP, ma è prevista, la possibilità di aggiungere altri tipi di comunicazione).
Nel caso del protocollo UDP, le funzioni usate dall'unità di controllo differiscono da quelle utilizzate dall'unità di calcolo, per la parte in comune ha solo lo scopo di definire precisamente quali sono le interfacce, e fare in modo che modi di comunicazione possano essere aggiunti senza modifiche al restante codice.

5.4.3 Moduli dell'unità di calcolo

L'unità di calcolo prevede l'implementazione, dal suo lato, delle funzioni per realizzare la comunicazione, nonché la definizione delle strutture dati relative all'implementazione del filtro e del buffer circolare. L'interfaccia per il filtro è stata scelta in modo da rendere semplice l'aggiunta di nuovi filtri senza cambiare la restante parte del programma.
Come mostrato in Figura 29 sull'unità di calcolo è in esecuzione l'applicazione *calcolo*, implementata nel file *calcolo.c*, ed ha i seguenti compiti:

- creare una connessione UDP;
- ricevere i dati dall'unità di controllo;
- filtrare i dati;

- ritrasmettere i dati all'unità di controllo.

Interfaccia utente dal lato unità di calcolo

L'applicazione dal lato dell'unità di calcolo prevede, un'interfaccia utente che permette, da linea di comando le seguenti impostazioni:

- scelta del protocollo di comunicazione;
- scelta della porta su cui mettersi in ascolto;
- scelta dei canali da filtrare;
- modalità verbose, per visualizzare le impostazioni effettuate;
- visualizzare l'help on-line.

5.4.4 Moduli dell'unità di controllo

L'unità di controllo prevede l'implementazione, dal suo lato, delle funzioni relative alla comunicazione UDP, che differiscono da quelle utilizzate nell'unità di calcolo. Inoltre l'unità di controllo è fornita di un modulo che permette al sistema di agire in modo semplice e veloce ai dispositivi ADC e DAC. Quest'ultimo modulo è stato progettato in modo da offrire sempre la stessa interfaccia anche in presenza di forti cambiamenti sul modello e sul numero dei dispositivi.

Come mostrato in Figura 29 sull'unità di controllo è in esecuzione l'applicazione *controllo*, implementato nel file *controllo.c*, ed ha i seguenti compiti:

- connettersi all'unità di calcolo;
- leggere i dati dall'ADC su 32 canali indipendenti;
- inviare i dati letti all'unità di calcolo;
- ricevere i dati inviati dall'unità di calcolo;
- scrive i dati nei DAC, inviandoli su un numero di canali di output definito dall'utente.

Prevede opzionalmente, per il testing, il calcolo del tempo medio di round trip dei dati, ed il numero di pacchetti persi.

Interfaccia utente dal lato unità di controllo

L'applicazione dal lato dell'unità di controllo prevede, un'interfaccia utente che permette, da linea di comando le seguenti impostazioni:

- visualizzare l'help on-line;
- modalità verbose, per visualizzare le impostazioni effettuate;
- indirizzo ip dell'unità di calcolo;
- scelta del protocollo di comunicazione;
- scelta della porta su cui mettersi in ascolto.

5.5 Analisi architetturale e progettazione dell'interfaccia di comando

In questa sezione sarà illustrata la struttura e la progettazione dell'interfaccia di comando su rete.

5.5.1 Architettura del sistema di interfaccia

L'interfaccia di comando su rete non è stata una caratteristica del sistema prevista dall'inizio del lavoro, ma in ogni modo s'integra bene con tutti gli altri elementi. Il problema era quello di riuscire ad avviare processi autonomi su macchine diverse, e per fare ciò ci si è serviti di speciali programmi "demoni" sempre in esecuzione sia sull'unità di calcolo (*cdafd_calcolo*) sia sull'unità di controllo (*cdafd_controllo*). Nella Figura 30 è rappresentato lo schema della struttura realizzata.

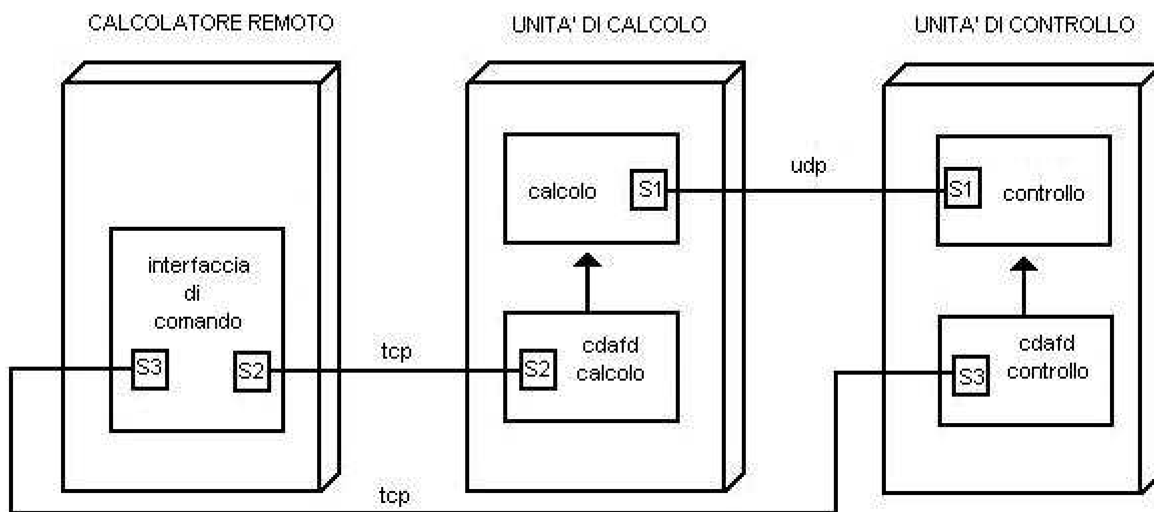

Figura 30: Struttura interfaccia di comando

Ovviamente i demoni devono essere sempre in esecuzione; il loro compito è quello di accettare comandi da rete ed eseguirli avviando o arrestando il rispettivo processo.
La comunicazione tra il calcolatore remoto e una qualsiasi unità si basa sul modello "client server", dove i programmi demoni rappresentano i server (sempre in attesa di connessioni), e l'interfaccia di comando è il client. Per la comunicazione su rete, non avendo questa volta restrizioni di tempo, è stato scelto il protocollo TCP, che ci offre una maggiore affidabilità.
Tutti i processi sono stati scritti in linguaggio ANSI C, e conformi allo standard POSIX, mentre il programma d'interfaccia è stato scritto in linguaggio Java per avere un alto grado di portabilità.

5.5.2 Interfacce interne comuni ai programmi demoni

Le funzioni in comune tra i due demoni riguardano la trasmissione dei dati su socket mediante il protocollo di trasporto TCP, e il loro prototipo è contenuto nel file *socket.h* e sono riassunte nella Tabella 13.

Funzione	Descrizione
avvia_server()	avvia il server
accetta_connessione()	si mette in ascolto di un client
chiudi_connessione ()	chiude una connessione stabilita in precedenza
arresta_server ()	arresta il server
ricevi()	riceve byte grezzi dalla socket
trasmetti()	trasmette byte grezzi sulla socket

Tabella 13: Funzioni comuni ai programmi demoni

Anche se il modulo per la comunicazione permette di accettare richieste di più client, ciò non è previsto perché non ha senso avviare due istanze dello stesso processo su l'unità di calcolo oppure quella di controllo. Le funzioni per la comunicazione su socket non possono essere condivise con il programma client perché è scritto in linguaggio Java. Sarebbe in ogni modo possibile chiamare routine scritte in C da un'applicazione Java tramite la "Java Native Interface", ma questo riduce drasticamente la portabilità del programma.

Nome struttura	Parametro	Descrizione
parametri_filtro_t	*window*	dimensione della finestra
opt_calcolo_t	*command*	permette di avviare o arrestare il processo
	ip	ip dell'unità di controllo
	portnum	porta aperta sull'unità di controllo
	filtro[]	indica su quali canali attivare il filtro
	parametri_filtro[]	contiene i parametri di configurazione dei filtri
	protocollo	scelta del protocollo da utilizzare

Tabella 14: Strutture richiesta al programma demone lato calcolo

5.5.3 Moduli del programma demone lato unità di calcolo

Il demone in esecuzione nell'unità di calcolo, oltre alle interfacce interne in comune appena descritte, prevede l'implementazione delle funzioni *avvia_calcolo()* e *arresta_calcolo()* descritte in questa sezione.

La funzione *main()* del processo non fa altro che attendere richieste da rete, elaborarle, e rispedire al client un indicatore di successo o di fallimento. Ogni richiesta è accompagnata

da parametri di configurazione contenuti nella struttura *opt_calolo_t* descritta in Tabella 14. La differenza sostanziale tra il programma demone in esecuzione sull'unità di calcolo e quello dell'unità di controllo è il formato di questa struttura.

avvia_calcolo()

Questo modulo accetta in ingresso dei parametri di configurazione, utili ad avviare in modo corretto il programma calcolo. Precisamente svolge le seguenti operazioni:

1. Controllare la correttezza dei parametri;
2. Costruire il vettore dei parametri per avviare correttamente il processo calcolo;
3. Avviare mediante la chiamata di sistema *execvp()* il suddetto processo;
4. Conservare il pid del processo appena avviato;
5. Liberare la memoria occupata dal vettore dei parametri.

Solo se tutte queste operazioni hanno buon fine, la funzione ritorna un indicatore di successo.

arresta_calcolo()

La routine *arresta_calcolo()* ha il compito di disfare tutto ciò che è stato fatto dalla routine *avvia_calcolo()*. Le operazioni svolte sono:

1. Recuperare il pid del processo avviato in precedenza;
2. Far terminare il processo all'istante inviandogli un segnale di SIGINT;
3. Controllare lo stato di terminazione mediante la chiamata di sistema *wait()*;
4. Segnalare la terminazione del processo.

La funzione ritornerà un indicatore di successo solo se le operazioni appena elencate vanno a buon fine.

5.5.4 Moduli del programma demone lato unità di controllo

I moduli per il programma demone per l'unità di controllo sono molto simili a quelli per l'unità di calcolo; questa volta sono avvia_controllo() e arresta_controllo(), e svolgono sostanzialmente le stesse operazioni, ma operano sulla struttura dati descritta nella Tabella 15.

Nome struttura	Parametro	Descrizione
opt_controllo_t	*command*	permette di avviare o arrestare il processo
	ip	ip dell'unità di calcolo
	portnum	porta aperta sull'unità di calcolo
	protocollo	scelta del protocollo da utilizzare

Tabella 15: Strutture richiesta al programma demone lato controllo

5.5.5 Struttura interfaccia di comando

L'interfaccia di comando è scritta completamente in linguaggio Java, quindi assicura la massima portabilità. Il suo cuore è rappresentato dal pacchetto *CdafCtrl* il quale contiene le classi *ControlCdaf, OpzioniCdaf, Sender, StringConverter*. Altri due pacchetti scritti sono InterfacciaGrafica e InterfacciaTestuale che si basano sull'uso del pacchetto *CdafCtrl*, e implementano rispettivamente un'interfaccia di comando grafica (GUI), e un'interfaccia di comando testuale. In questa sezione è descritto principalmente il pacchetto *CdafCtrl* poiché motore dell'interfaccia di comando.
Il diagramma delle classi del pacchetto CdafCtrl è illustrato in Figura 31: di seguito presenteremo le funzioni d'ogni classe contenuta al suo interno.

OpzioniCdaf

Un oggetto istanza della classe *OpzioniCdaf* è l'addetto a contenere tutti i parametri di configurazione del sistema realizzato. A questo livello non si fa ancora distinzione tra i parametri da inviare all'unità di calcolo e quelli da inviare all'unità di controllo; inoltre tutti i parametri sono rappresentati da oggetti di tipo *String*, normalmente stringhe rappresentate nel set di caratteri *unicode* a 16 bit.

OpzioniCdaf è una classe visibile dall'esterno del pacchetto.

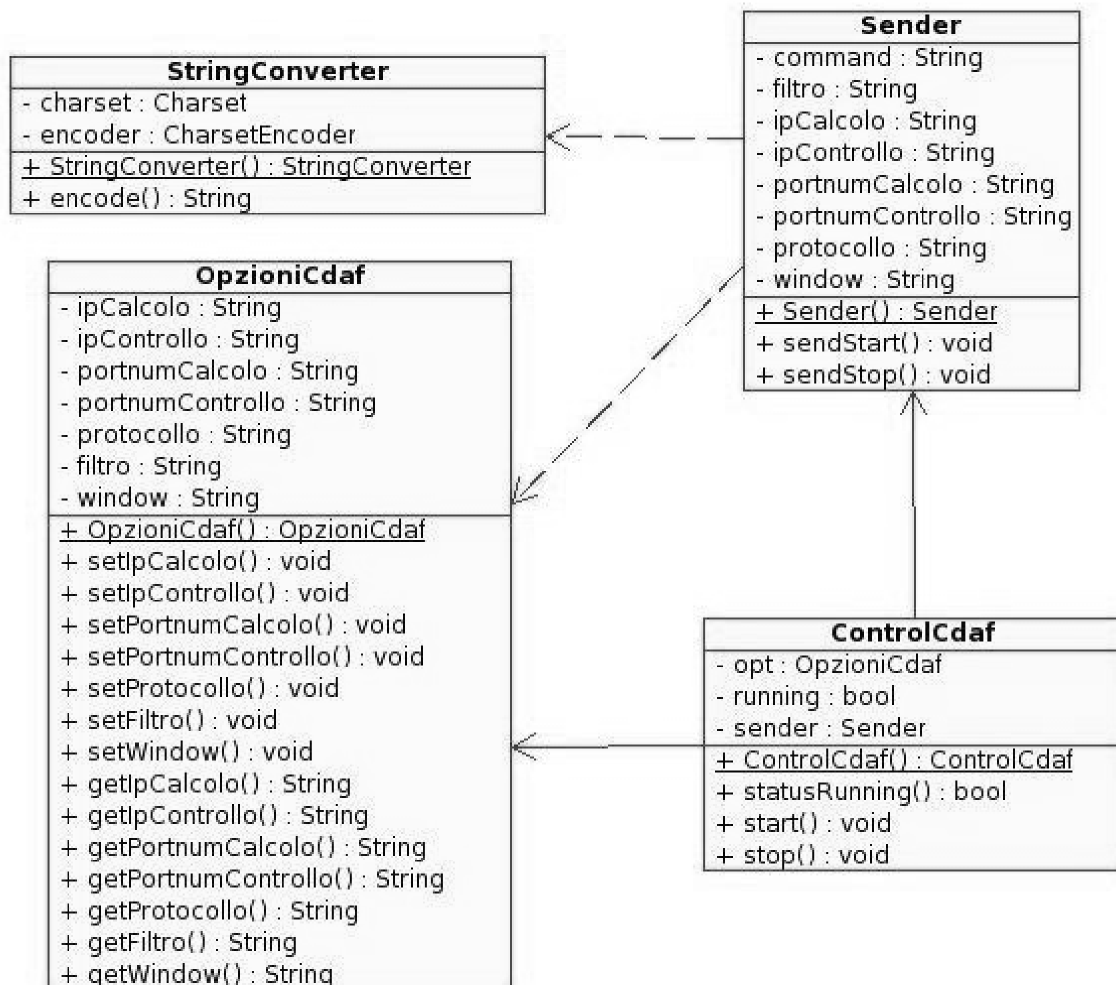

Figura 31: Diagramma delle classi del pacchetto CdafCtrl

ControlCdaf

ControlCdaf è l'altra classe del pacchetto *CdafCtrl* ad essere visibile dall'esterno del pacchetto. Per funzionare correttamente, un oggetto di questa classe, si serve di un oggetto di tipo *OpzioniCdaf* dal quale ricavare la configurazione scelta dall'utente.

Il diagramma di stato di un'istanza della classe *ControlCdaf* è rappresentato in Figura 32.

Un'istanza di questa classe ControlCdaf può trovarsi in due stati:

- Sistema fermo: in questa condizione né l'unità di calcolo, né quella di controllo sono in funzione.
- Sistema avviato: entrambe le unità sono in funzione se ci troviamo in questo stato.

La transizione tra i due stati avviene mediante i metodi *start()* e *stop()*, ed è possibile conoscere lo stato dell'oggetto mediante il metodo *statusRunning()*.

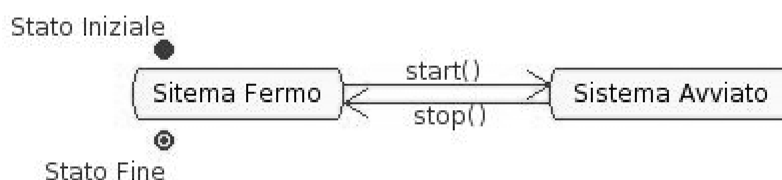

Figura 32: Diagramma di stato per la classe ControlCdaf

StringConverter

A differenza delle classi descritte in precedenza, *StringConverter* non è visibile dall'esterno del pacchetto. Utilizziamo questa classe per convertire stringhe nel formato predefinito di Java, unicode a 16 bit, in stringhe nel formato utilizzato dai demoni in esecuzione nell'unità di calcolo e quella di controllo, UTF a 8 bit.
Il metodo usato per convertire le stringhe è *encode()*, e prevede i seguenti prototipi:

```
String encode(String in_str);
String encode(String in_str, int maxDim);
```

Nel primo caso ricava automaticamente la lunghezza della stringa, nel secondo, invece, chiede anche la lunghezza della stringa di output.
Questo è utile poiché le unità di controllo e di calcolo sono in attesa di stringhe, in formato UTF a 8 bit, di dimensione prestabilita.

Sender

Anche in questo caso la classe *Sender* non è visibile dall'esterno del pacchetto. Un oggetto di tipo *ControlCdaf* utilizza *Sender* per inviare dati su socket mediante il protocollo TCP sia all'unità di calcolo sia all'unità di controllo.

Un'istanza della classe *Sender* si serve della classe *StringConverter* per convertire le stringhe prima di inviarle su rete. Un diagramma di sequenza d'esempio, relativo al metodo sendStart() è illustrato in Figura 33.

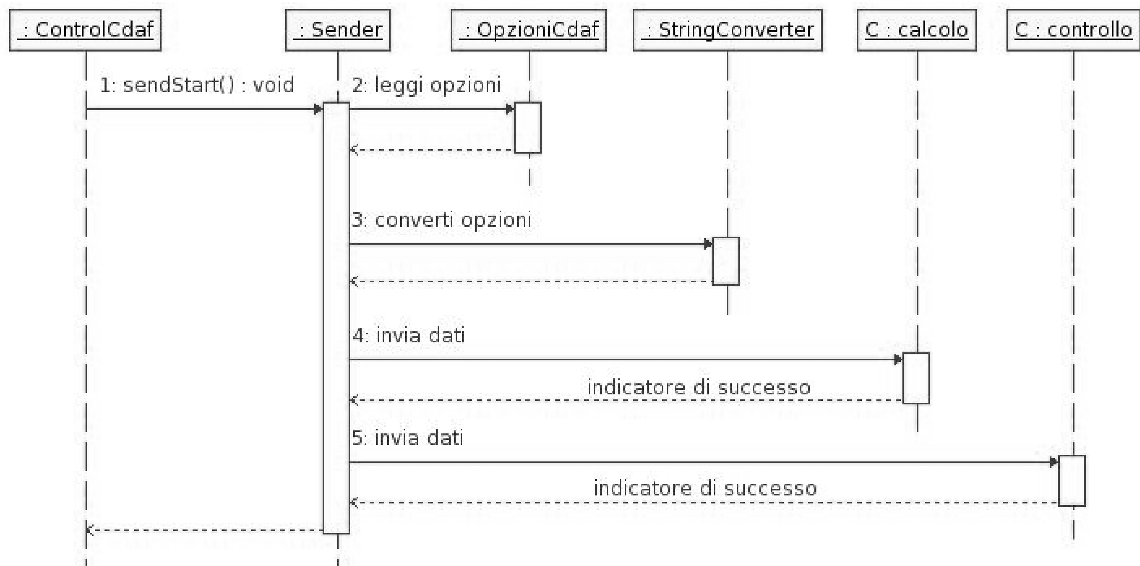

Figura 33: Diagramma di sequenza relativo al metodo sendStart()

5.5.6 Interfaccia grafica (GUI)

Per lo sviluppo di un interfaccia grafica ci si è serviti di un costruttore di GUI poiché la libreria grafica usata (*Swing*, descritta nel capitolo 8) è stata progettata pensando a questi comodi strumenti automatici.

Il pacchetto che implementa l'interfaccia grafica si chiama *InterfacciaGrafica* ed è composto principalmente da quattro classi: *StartCdaf*, *Frame1*, *FilterBox*, *AboutBox*. Il diagramma delle classi di questo pacchetto è mostrato in Figura 34.

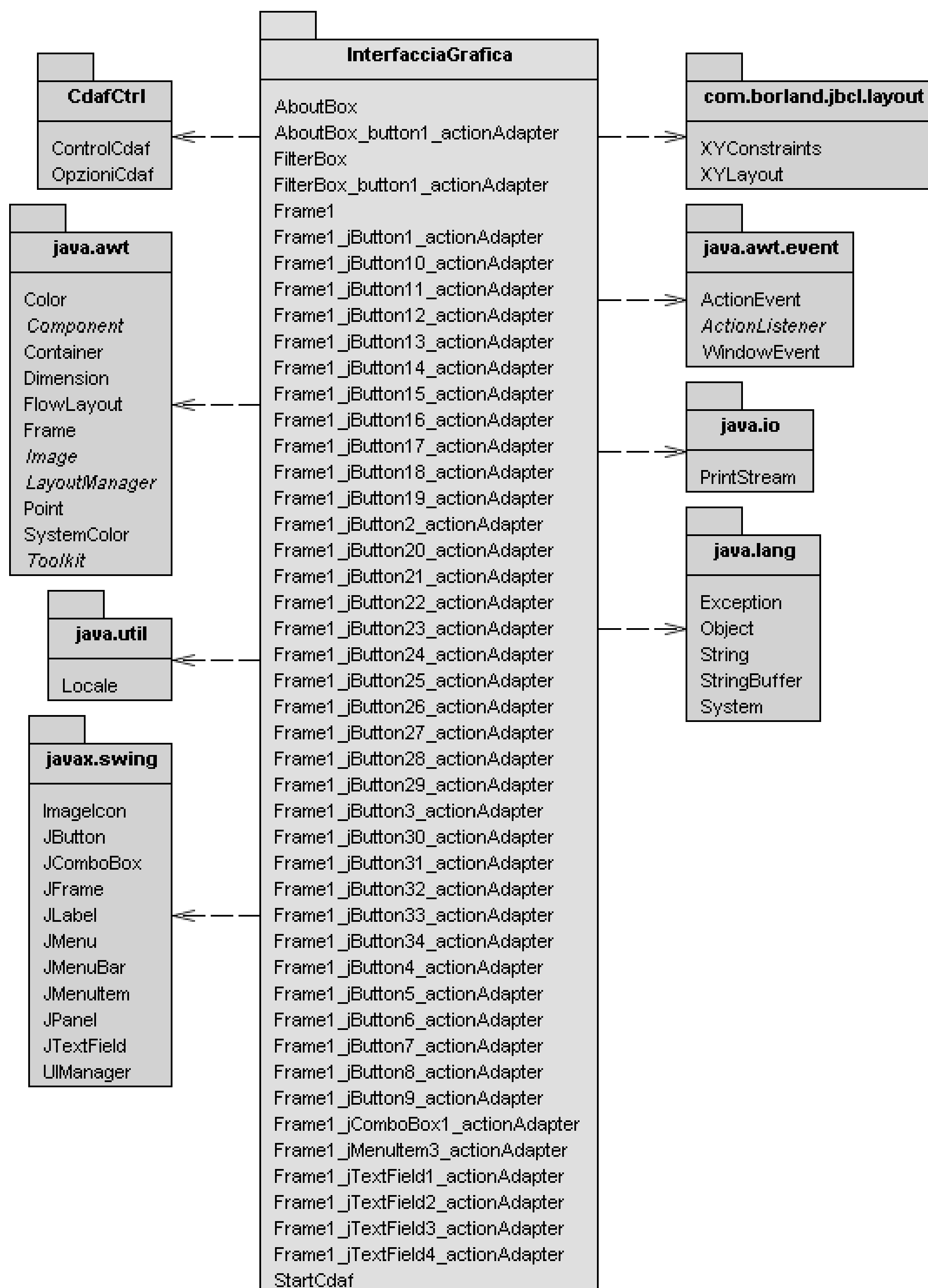

Figura 34: Diagramma delle classi del pacchetto InterfacciaGrafica

La classe che contiene il metodo main() in questo pacchetto è StartCdaf, la quale ha il compito di far visualizzare la prima finestra. Questa, ed altre classi appartenenti a questo pacchetto, sono illustrate dettagliatamente nel capitolo 8.

5.5.7 Interfaccia a linea di comando

Oltre ad un interfaccia grafica, il sistema è dotato anche di un'interfaccia a linea di comando, che implementa le stesse funzioni dell'interfaccia grafica. Il pacchetto che contiene l'interfaccia a linea di comando si chiama InterfacciaTestuale e contiene solo due classi: ModoTesto e ConsoleReader. La Figura 35 mostra il diagramma delle classi del pacchetto *InterfacciaTestuale*.

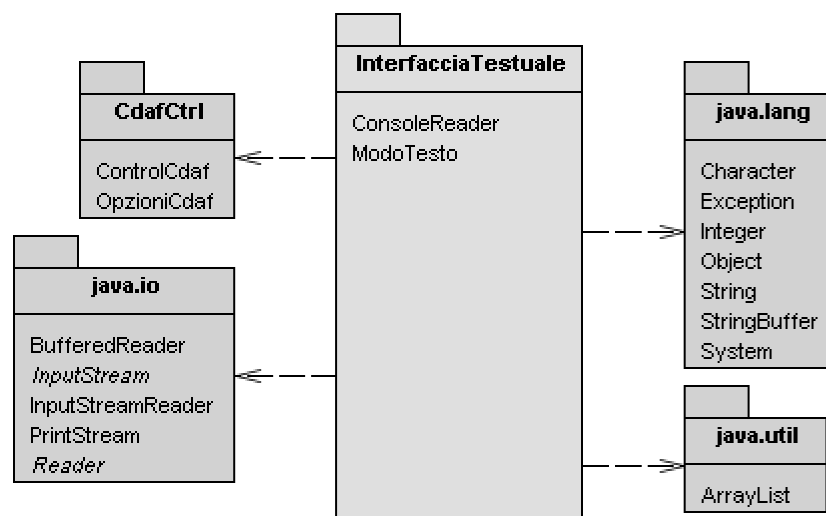

Figura 35: Diagramma delle classi del pacchetto InterfacciaTestuale

La classe principale, *ModoTesto*, si serve della classe *ConsoleReader* per ricevere l'input dall'utente, e poi interagisce con il pacchetto *CdafCtrl* per impartire ordini al sistema.

5.5.8 Funzioni offerte dall'interfaccia di comando

Le funzioni messe a disposizione dall'interfaccia di comando (sia grafica sia testuale), permettono di impostare i seguenti parametri del sistema:

- ip dell'unità di controllo;
- numero di porta usato dall'unità di controllo;
- ip dell'unità di calcolo;
- numero di porta usato dall'unità di calcolo;
- scelta del protocollo di comunicazione;
- scelta dei canali da filtrare.

Inoltre si prevede una sezione dove, una volta implementato un filtro, sarà possibile configurarlo settando opportuni parametri.

5.6 Piano di testing del sistema

Il piano di testing consiste nel testing del driver e del sistema di comunicazione. Solo dopo aver testato opportunamente il driver, ed aver, eventualmente migliorato le sue prestazioni, è possibile utilizzare il driver nel contesto dell'intero sistema di comunicazione, per evitare rallentamenti nelle trasmissioni.
Il testing dei driver prevede:

1. apertura, scrittura e chiusura del DAC tramite le chiamate di sistema UNIX;
2. tutte le impostazioni possibili del dispositivo tramite la chiamata di sistema ioctl();
3. l'apertura del dispositivo da parte di più applicazioni;
4. conversione analogico digitale e digitale analogico con l'utilizzo del driver per il DAC a diverse frequenze di campionamento;
5. eseguire i test descritti prima sia con l'installazione statica sia dinamica.

Il testing per il sistema di comunicazione, per il filtro, e per l'interfaccia di comando prevede:

1. il calcolo del tempo di round-trip per ogni pacchetto, tempo di round-trip medio numero di pacchetti persi;
2. la trasmissione, sia con l'utilizzo del filtro sia senza;
3. l'impostazione da linea di comando dei canali da filtrare;
4. l'impostazione da linea di comando della porta di ascolto, e dell'indirizzo IP;
5. la visualizzazione della guida on line ed impostazione modalità verbose;
6. inviare, in modo ripetuto, ordini al sistema mediante interfaccia di comando;
7. far partire sia l'interfaccia grafica, che quella testuale, da locale e da remoto;
8. provare la portabilità dell'interfaccia grafica su diversi sistemi operativi.

5.7 Configurazione

Lo sviluppo del MPV955 DA Converter Device Driver e dell'applicazione residente sull'unità di controllo, è effettuato avendo come CPU la scheda VMPC6a (o superiore) della Thales Computer con sistema operativo LynxOS 4.0.0.
Lo sviluppo dell'applicazione per l'unità di calcolo è effettuato, invece avendo come CPU un processore Pentium X86 con sistema operativo Linux con kernel versione 2.4 o superiore (possibilmente di un kernel con versione superiore alla 2.6 per trarre profitto dallo scheduler real-time e preemptive presente dalla versione 2.6 in poi).

5.8 Norme per l'implementazione

I driver, e i programmi di interfaccia saranno elaborati in linguaggio ANSI C e rispetteranno le specifiche POSIX 1003.1, 1b, e 1c.

5.9 Matrice di copertura dei requisiti

La matrice di copertura dei requisiti (Tabella 16), indica in quale sottosezione del presente capitolo, sono stati affrontati i requisiti del problema.
Alcuni dei requisiti non riportati in tabella riguardano da vicino l'implementazione, pertanto non sono definiti in questa fase del progetto.

Sottosezione Requisito	3	3.3	4	4.1	4.2	4.3	4.4	4.7	4.8	5	5.5	6	7	8
REQ-FUNZ 1			●											
REQ-FUNZ 2													●	
REQ-FUNZ 3								●	●					
REQ-FUNZ 5				●		●								
REQ-FUNZ 6				●	●									
REQ-FUNZ 7					●									
REQ-FUNZ 8								●	●					
REQ-FUNZ 10	●													
REQ-INTF 1		●												
REQ-INTF 2		●												
REQ-INTF 3		●												
REQ-INTF 4		●												
REQ-INTF 5					●									
REQ-INTF 6										●				
REQ-CTST 1													●	
REQ-CTST 2														●
REQ-CTST 5											●			
REQ-PTC 1												●		
REQ-PTC 2							●							

Tabella 16: Matrice di copertura dei requisiti

Capitolo 6

Sviluppo del driver

Il nostro sistema si basa su due dispositivi: DAC e ADC. Per essere usati in modo conveniente abbiamo bisogno dei rispettivi driver per il sistema operativo LynxOS.
All'inizio del lavoro già eravamo forniti di un driver compatibile con il dispositivo ADC, indi questo capitolo, dopo aver presentato il modo in cui si scrive un driver per LynxOS, analizzerà la scrittura del driver per il dispositivo DAC MPV955.

6.1 Scrivere un driver

Un device driver[2] è un'interfaccia software tra il sistema operativo è l'hardware che cela le specifiche del dispositivo al sistema operativo. Esso fornisce al kernel un meccanismo per comunicare con un tipo specifico di dispositivo. Comunemente queste richieste di comunicazione sono trasferimenti di dati e comandi per pilotare in qualche modo il dispositivo. La Figura 36 mostra la posizione del driver nel sistema operativo LynxOS.
Il device driver è linkato nel kernel e s'interfaccia direttamente con il controller di un particolare dispositivo. Un'applicazione può accedere al dispositivo come se fosse un normale file, attraverso le chiamate di sistema per la gestione dei file. Il kernel invoca la giusta routine dentro il codice del device driver per gestire la richiesta di I/O di una applicazione. Inoltre un driver può essere invocato per far fronte ad interrupt e ad errori del bus. I device driver si distinguono in tipi a blocchi o a caratteri. La differenza sostanziale sta nel fatto che i tipi a blocchi hanno una dimensione fissata del dato da trasferire mentre

[2] Tratteremo i termini "device driver" e "driver" alla stessa maniera.

quelli a caratteri no. I dispositivi ADC e DAC richiedono entrambi di essere gestiti da un driver a caratteri; il presente capitolo non tratta le peculiarità della scrittura dei driver non a caratteri, già trattati in [5].

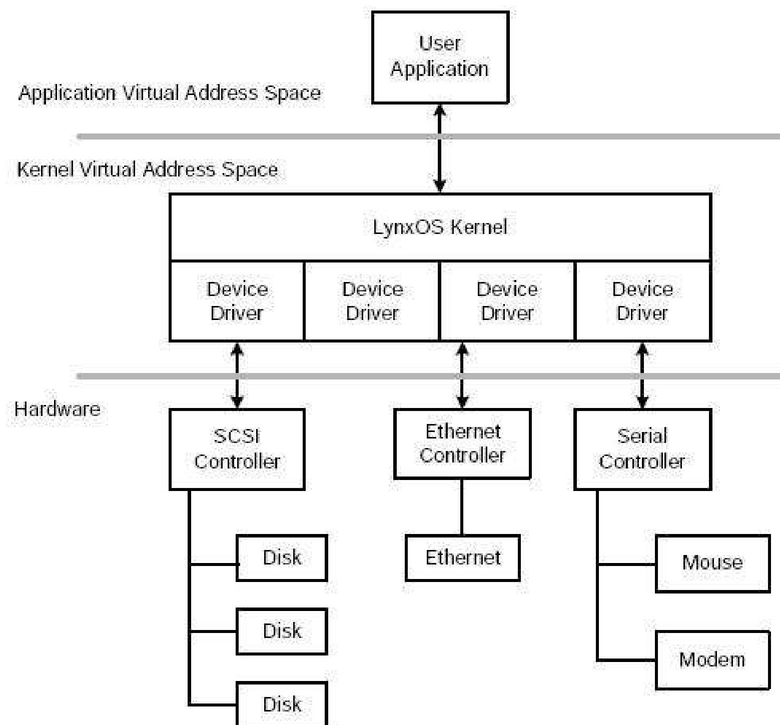

Figura 36: Collocazione di un driver

Un driver per LynxOS consiste di codice e strutture dati di supporto. Quando il driver è installato il kernel passa l'indirizzo della struttura device information al device driver stesso, in modo da identificare il dispositivo, e allocare in modo opportuno la dimensione del canale sul VMEbus e la memoria del dispositivo. Il kernel di LynxOS, come altri sistemi UNIX, ha un insieme di tabelle per tenere traccia dei dispositivi e dei driver installati nel sistema operativo, ed assegna un ID al dispositivo ed un ID unico al driver. Inoltre ogni dispositivo installato ha associato un majorID e un minorID, a cui ci si riferisce come numeri major e minor, rispettivamente. Generalmente il major si riferisce ad un controller e il minor ad un sotto dispositivo di quel controller. Il major è essenzialmente

l'istanziazione del blocco device information del dispositivo ed esiste per ogni dispositivo, mentre il numero minor non sempre è presente. Si può accedere ai dispositivi mediante file speciali presente nella directory "/dev".

```
lynxvirgo_devel# ls -l /dev|grep m|more
crw-r--r--   1 root     28,0    Feb  3 05:57 almavme
crw-rw-rw-   1 root      5,0    Feb  3 05:57 com1
crw-rw-rw-   1 root      6,0    Feb  3 05:57 com2
crw-r--r--   1 root      1,0    Feb  3 05:57 mem
crw-r--r--   1 root     34,0    Feb  8 03:44 mpv955
crw-rw-rw-   1 root     33,0    Feb  3 05:57 pmc_vmi55650
crw-------   1 root      9,208  Feb  3 05:57 rsdncr.0m
crw-------   1 root      9,209  Feb  3 05:57 rsdncr.1m
crw-------   1 root      9,210  Feb  3 05:57 rsdncr.2m
crw-------   1 root      9,211  Feb  3 05:57 rsdncr.3m
crw-------   1 root      9,212  Feb  3 05:57 rsdncr.4m
crw-------   1 root      9,213  Feb  3 05:57 rsdncr.5m
crw-------   1 root      9,214  Feb  3 05:57 rsdncr.6m
brw-------   1 root      1,208  Feb  3 05:57 sdncr.0m
brw-------   1 root      1,209  Feb  3 05:57 sdncr.1m
brw-------   1 root      1,210  Feb  3 05:57 sdncr.2m
brw-------   1 root      1,211  Feb  3 05:57 sdncr.3m
brw-------   1 root      1,212  Feb  3 05:57 sdncr.4m
brw-------   1 root      1,213  Feb  3 05:57 sdncr.5m
brw-------   1 root      1,214  Feb  3 05:57 sdncr.6m
crw-rw-rw-   1 root     31,0    Feb  3 05:57 tim0
crw-rw-rw-   1 root     31,1    Feb  3 05:57 tim1
```

Figura 37: Listato parziale della directory "/dev"

La Figura 37 mostra alcuni file della directory "/dev". Sembrano normali file, ma ognuno è associato a un particolare dispositivo (con un proprio major number), oppure è associato solo a una particolare caratteristica del dispositivo (identificata dal minor number).
Tra le righe si scorge il file speciale associato al dispositivo DAC per il quale è stato scritto il driver. Esso è "/dev/mpv955" e, nel nostro caso, gli è stato assegnato il major number 33.

6.2 Struttura dei driver per LynxOS

Il codice del driver è un modulo incorporato nel kernel e quindi non ha la funzione *main()*, ma possiede le funzioni necessarie ad un'applicazione per interfacciarsi con il dispositivo, dette *Entry Point*, che sono linkate al kernel e usate dallo stesso come parte del meccanismo per tradurre le chiamate di sistema per la gestione dei file in comandi per i

dispositivi. Le "Entry Point" corrispondono, quindi, a livello applicativo alle solite chiamate di un sistema UNIX per la gestione dei file. In seguito si analizzeranno i componenti (illustrati in Figura 38) di un driver, e cioè:

- Entry point: sono tutte le funzioni appartenenti al driver;
- Struttura delle informazioni del dispositivo (device information): utilizzata al momento dell'installazione comprende informazioni concernenti un dispositivo;
- Struttura statica: questa struttura è usata come un contenitore di parametri, e ogni routine del driver può accederci per scambiare informazioni con le altre routine.
- Struttura dldd: è necessaria solo in caso d'installazione dinamica del driver, e fornisce un modo per trovare i nomi delle funzioni entry point.

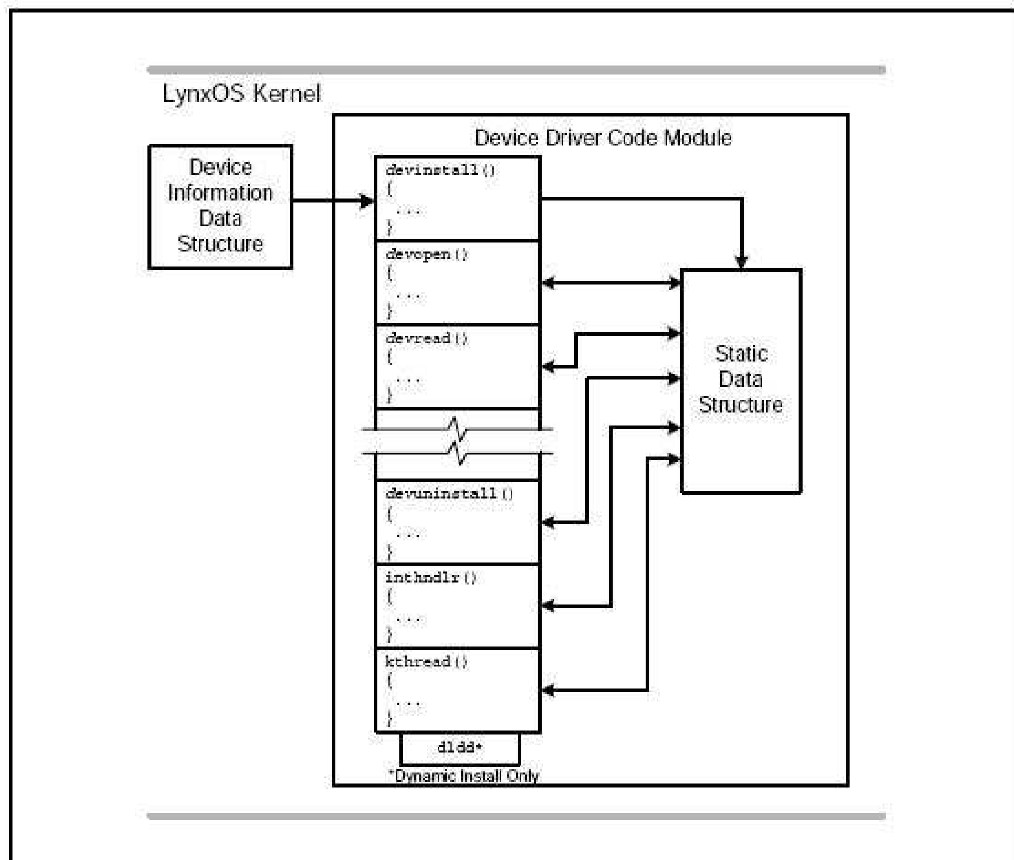

Figura 38: Componenti di un driver

6.2.1 Entry Point

Le "Entry Point" (in italiano "punti d'entrata") non sono altro le funzioni appartenenti al driver; ogni volta che un'applicazione utente effettua chiamate di sistema, il sistema operativo provvede a invocare la funzione del driver associata alla chiamata di sistema effettuata (Figura 39).

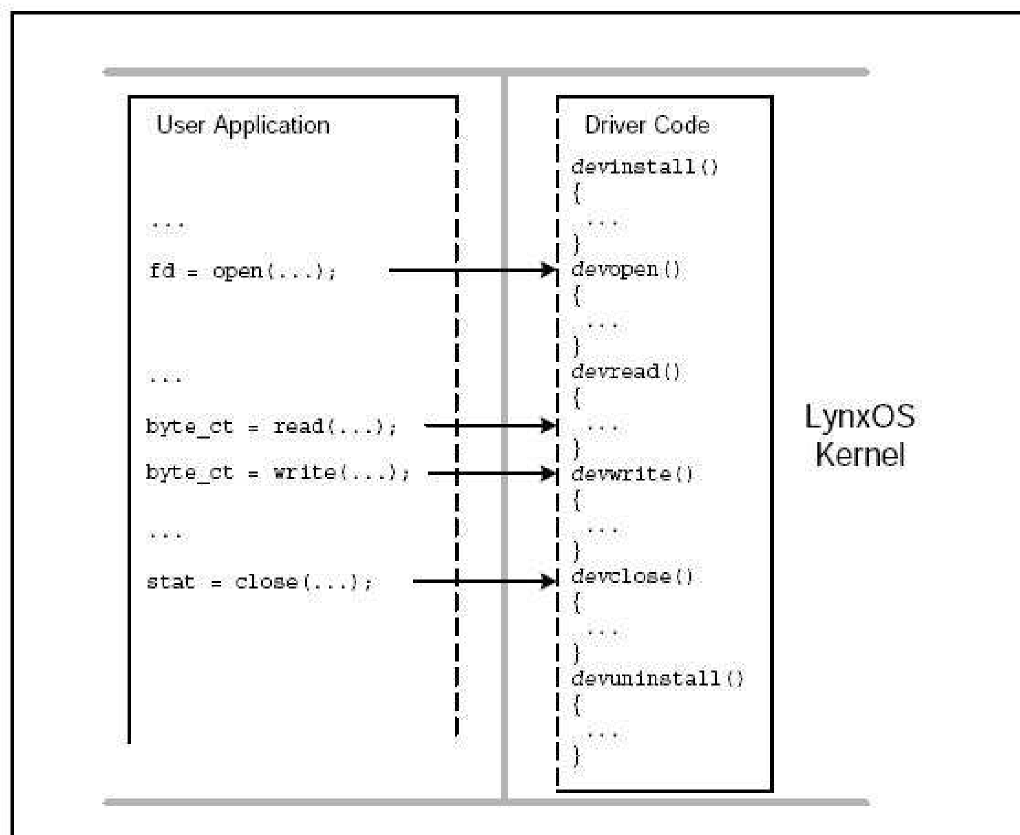

Figura 39: Traduzione delle chiamate di sistema

E' solito dare il nome agli entry point come una combinazione del nome del dispositivo, seguito dal nome della chiamata di sistema. Nel nostro caso sono state implementate le funzioni *mpv955_open(), mpv955_close(), mpv955_write(), mpv955_ioctl(),* corrispondenti rispettivamente alle chiamate di sistema *open(), close(), write(), ioctl()*.

6.2.2 Struttura delle informazioni del dispositivo

La struttura delle informazioni del dispositivo (meglio conosciuta come "device information"), è utilizzata per installare uno specifico dispositivo nel sistema operativo ed è esterna al codice del driver. Di solito è contenuto in questa struttura l'indirizzo del dispositivo sul bus, e informazioni sulla gestione degli interrupt. Nel nostro caso questa struttura è molto piccola poiché contiene solo l'indirizzo base del dispositivo sul bus:

```
typedef struct mpv955_info {
      unsigned short base_address;
} mpv955_info_t;
```

È ovvio che per installare più dispositivi con lo stesso driver è necessario allocare più istanze di questa struttura. I dettagli sull'installazione del driver nel sistema operativo sono trattati più avanti nel corso di questo capitolo, per adesso è necessario sapere che un'istanza di questa struttura è passata alla routine del driver *mpv955_install()* dal kernel.

6.2.3 Struttura statica

La struttura statica, a differenza della struttura delle informazioni del dispositivo, è interna al codice del device driver. L'entry point *mpv955_install()* è incaricata di allocare dinamicamente la struttura statica e di restituire il puntatore al kernel che lo passa come parametro alle routine del driver, e ogni ognuna di esse lo usa per scambiare informazioni con le altre routine, per memorizzare valori utili in futuro, e per accedere alla memoria del dispositivo. Comunemente questa struttura contiene campi per la gestione degli interrupt, campi per implementare la sincronizzazione, la definizione dei registri e della memoria del dispositivo; nel nostro caso è:

```
typedef struct {
      unsigned short control_status;
      unsigned short start_address_register;
      unsigned short stop_address_register;
```

```
	unsigned short interrupt_control_register;
	unsigned short rate_timer_control;
	unsigned short timeout_control;
	unsigned short dac_disable;
} mpv955_registers_t;

typedef struct {
	unsigned short timeout;
} mpv955_start_register_t;

typedef struct {
	mpv955_info_t *info;
	int area_cm;
	mpv955_registers_t *registers[2];
	mpv955_start_register_t *start_register;
	unsigned short *data_memory;
	char *p_a24_phys_addr;
	unsigned short zero_volts;
	unsigned short rate_timer;
	unsigned short timeout_value;
	int primo;
} mpv955_data_t;
```

Questa definizione si trova nel file *mpv955drvr.h*, che è l'header file principale del driver, i suoi campi principali sono analizzati nella Tabella 12. Notiamo che la struttura *mpv955_registers_t* rispecchia in modo esatto l'organizzazione dei registri del dispositivo (rif [9]).

6.3 Analisi del codice

In questa sezione sarà descritta l'implementazione delle entry point appartenenti al driver per il dispositivo MPV955. Per motivi di spazio non è stato possibile inserire il codice sorgente, scaricabile dal sito internet www.studenti.unina.it/~fsterrar/tirocinio.

6.3.1 mpv955_install()

Questa routine è chiamata ogni volta che installiamo nel sistema un dispositivo compatibile con il driver. Permette di associare il particolare dispositivo negli indirizzi del bus VME,

ed ha il compito di allocare dinamicamente la struttura statica. Il prototipo di questa funzione è:

```
char *mpv955_install(mpv955_info_t *p_mpv955_info);
```

Accetta in ingresso il puntatore alla struttura delle informazioni del dispositivo, contenente nel nostro caso l'indirizzo sul bus VME, grazie al quale può allocare correttamente lo spazio necessario. Dopo aver avuto accesso al bus con la funzione *almavme_channel_alloc()*, è effettuato un test d'accesso al dispositivo: in caso di fallimento è restituito il valore speciale SYSERR.

Di seguito viene allocata e inizializzata la struttura statica, vengono puliti i canali del DAC (procedura descritta a pagina 56), e alla fine è ritornato il puntatore della nuova struttura. Il kernel provvederà a passare alle altre entry point il puntatore restituito dalla funzione appena esaminata.

6.3.2 mpv955_uninstall()

Questa funzione ha il compito di rimuovere un dispositivo dal sistema, quindi deve disfare tutto il lavoro svolto dalla routine *mpv955_install()*. Il prototipo della funzione è:

```
int mpv955_uninstall(mpv955_data_t *p_mpv955_data);
```

La funzione ritorna il valore OK se non incontra problemi durante la sua esecuzione, altrimenti è ritornato il valore SYSERR.

La sequenza d'operazioni svolta da questa routine è:

- liberare lo spazio allocato sul bus mediante la funzione *almavme_channel_free()*;
- deallocare lo spazio riservato alla struttura statica.

Non sono effettuate operazioni di chiusura sul dispositivo perché questo compito è lasciato alla routine *mpv955_close()*.

6.3.3 mpv955_open()

La routine *mpv955_open()* è chiamata dal kernel ogni volta che un'applicazione utente apre il dispositivo con la chiamata di sistema *open()*. Il suo prototipo è:

```
int mpv955_open(mpv955_data_t *p_mpv955_data, int devno,
                struct file *f);
```

L'unico compito di questa routine è quello di accedere ai registri della scheda per preparare il dispositivo alla conversione dei dati. Qui c'è un problema di sincronizzazione perché la funzione può essere chiamata da più processi, e quindi il dispositivo può essere inizializzato mentre già un altro processo lo sta usando. Per evitare ciò si è protetta la sezione critica disabilitando temporaneamente la prelazione dei processi.
La prima volta che un processo apre il dispositivo, esso viene inizializzato con le seguenti caratteristiche:

- clock interno con rate time a 2μs;
- interrupt disabilitati;
- modalità di output continua;
- tutti e otto i canali abilitati.

Nel caso un processo apra il dispositivo mentre è già aperto da un altro processo, la routine *mpv955_open()* non ha effetti sul dispositivo. La funzione ritorna il valore OK se non incontra problemi durante la sua esecuzione, in altro modo è ritornato il valore SYSERR.

6.3.4 mpv955_close()

La routine *mpv955_close()* è chiamata dal kernel ogni volta che un'applicazione utente chiude il dispositivo con la chiamata di sistema *close()*. Il suo prototipo è:

```
int mpv955_close(mpv955_data_t *p_mpv955_data, struct file *f);
```

Con questa routine non abbiamo problemi di sincronizzazione perché il kernel di LynxOS provvede a chiamarla solo quando l'ultimo processo che stava utilizzando il dispositivo decide di chiuderlo.
L'unico compito di *mpv955_close()* è quello di disabilitare le uscite analogiche del dispositivo. La funzione ritorna il valore OK se non incontra problemi durante la sua esecuzione, diversamente è ritornato il valore SYSERR.

6.3.5 mpv955_write()

Le due chiamate di sistema più utilizzate, quando si opera su di un dispositivo, sono read() e write(). Nel nostro caso è stata implementata solo *mpv955_write()*, poiché il dispositivo DAC non è fatto per essere letto. Il prototipo della funzione è:

```
int mpv955_write(mpv955_data_t *p_mpv955_data, struct file *f,
                 char *buf, int count);
```

Il modo in cui è stata implementata la funzione rispecchia gli standard POSIX, ma cambia il significato dell'ultimo parametro. In effetti, l'ultimo parametro (*int count*) dovrebbe rappresentare la dimensione del dato (*char *buf*), e noi lo trattiamo come il numero del canale sul quale scrivere; questo è stato possibile supponendo costante la dimensione del dato passato alla routine. Di norma non è buona cosa non seguire lo standard, ma contrariamente ai computer generici, un sistema embedded ha dei compiti conosciuti già a priori, e durante lo sviluppo del sistema si è fatto di tutto per renderlo più veloce possibile. Questa è l'unica eccezione: la parte restante del codice è fedele allo standard POSIX.

6.3.6 mpv955_ioctl()

La routine *mpv955_open()* lascia il dispositivo all'applicazione utente inizializzato in modo generico. Ogni volta che l'utente vuole configurare il dispositivo a suo piacimento, deve ricorrere alla chiamata di sistema *ioctl()* direttamente collegata alla routine del driver *mpv955_ioctl()*.

Il dispositivo MPV955 può funzionare in varie modalità, può avere varie impostazioni, e può essere sincronizzato con diversi tipi di clock. La funzione *mpv955_ioctl()* ha lo scopo di permettere all'utente di configurare il dispositivo a suo piacimento, o di restituire informazioni sullo stato corrente del dispositivo.

```
int mpv955_ioctl(mpv955_data_t *p_mpv955_data, struct file *f,
                 int command, char *arg);
```

La funzione ritorna il valore OK se non incontra problemi durante la sua esecuzione, nel caso contrario è ritornato il valore SYSERR. Un'esaustiva descrizione di questa funzione e delle sue funzionalità è data nella sezione a pagina 64.

6.4 Installazione del driver nel sistema operativo

Il processo di linkaggio delle routine del driver nel sistema operativo, è chiamato "installazione del driver". Ci sono due modi per fare ciò:

- installazione dinamica: fattibile a caldo, quando il sistema operativo è in esecuzione;
- installazione statica: il codice del driver viene incorporato nel kernel e si necessita di riavviare il sistema.

Il nostro driver inizialmente s'installava solo in modo dinamico: questo favoriva lo sviluppo del driver perché eravamo in grado di provarlo senza riavviare la macchina ogni

volta. Adesso, grazie ad uno script d'installazione, è possibile installare il driver nel sistema sia staticamente che dinamicamente.
In entrambi i casi, per ogni dispositivo che utilizza il driver, si deve allocare la "struttura delle informazioni del dispositivo". Ad esempio, quando abbiamo installato il primo dispositivo MPV955, è stata allocata la seguente struttura:

```
mpv955_info_t mpv955info_0={0x00F00000};
```

Con questo valore, la routine *mpv955_install()* può installare correttamente il dispositivo nel sistema. Per installare più dispositivi con utilizzando lo stesso driver, occorre fare altre dichiarazioni come quella appena esposta.
Di seguito sono mostrati i metodi sia per l'installazione dinamica, sia per quella statica.

6.4.1 Installazione dinamica

Nel caso d'installazione dinamica di un driver, agli elementi del driver si aggiunge la struttura dldd (Dynamic Linked Device Driver). Tale struttura è inizializzata nel codice del driver, i suoi campi corrispondono ai nomi delle funzioni entry point, che di conseguenza sono passate al kernel quando il driver è installato. Se una particolare entry point non è presente nel driver (ad esempio nel nostro caso la *read()*), il campo vuoto si riferisce ad una funzione esterna *ionull()*, che è una semplice funzione del kernel, restituisce semplicemente OK. Il metodo dinamico permette di effettuare l'installazione e la disinstallazione del driver velocemente, indi risulta utile durante le fasi di sviluppo. Inoltre permette di usare più driver per lo stesso dispositivo.
È molto semplice installare un driver in modo dinamico, basta eseguire i seguenti passi:

- compilare il codice sorgente del programma;
- creare un modulo oggetto caricabile dinamicamente (con il comando *ld*);
- installare il modulo oggetto creato precedentemente;
- allocare la struttura delle informazioni del dispositivo;

- installare il dispositivo facendo uso della struttura allocata;
- creare il file speciale del dispositivo ella directory “/dev”

Questa procedura è corretta nei sistemi LynxOS con processore PowerPC.

6.4.2 Installazione statica

In questo caso il codice del driver è linkato in modo statico al kernel. Un driver installato in questa maniera è sempre presente nel kernel e, se configuriamo in modo appropriato i dispositivi che lo usano, li troviamo sempre attivi non appena il sistema è avviato, senza aver bisogno di installarli ogni volta. L’installazione statica del device driver non richiede la struttura dldd, ma esige un’organizzazione più rigida del codice sorgente e dei file di configurazione rappresentata in Tabella 17.

Directory	File	Descrizione
/	*lynx.os*	Kernel LynxOS
/sys/lib	*libdrivers.a*	Libreria codice oggetto dei driver
	libdevices.a	Libreria codice oggetto dei dispositivi
/sys/dheaders	*mpv955info.h*	Definizione della device information
/sys/devices.vmpc	*mpv955info.c*	File di configurazione
/sys/drivers,vmpc/MPV955	*mpv955drvr.c*	Codice sorgente del driver
	mpv955drvr.h	File header
	Makefile	Per compilare il codice del driver
/sys/lynx.os	*CONFIG.TBL*	File di configurazione dei driver del sistema
	Makefile	Istruzioni per compilare lynx.os
/etc	*nodetab*	Nodi dei dispositivi
/sys/cfg	*MPV955.cfg*	File di configurazione del driver MPV955
/usr/include	*mpv955ioctl.h*	Header file per le applicazioni
/usr/bin	*Install.MPV955*	Script di installazione del driver
	Uninstall.MPV955	Script di disinstallazione del driver

Tabella 17: Organizzazione del codice sorgente e dei file del driver

Facendo riferimento a tale tabella è semplice capire i seguenti passi utili all'installazione di un driver su un sistema LynxOS con processore PowerPC:

- Aggiornare la libreria dei dispositivi (*libdevices.a*) e dei driver (*libdrivers.a*);
- Aggiornare il file di configurazione del kernel (*CONFIG.TBL*);
- Ricompilare e installare il kernel, quindi riavviare l'intero sistema.

6.5 Collaudo

Il teorema di Shannon (meglio illustrato a pagina 12), riguardante la teoria dei segnali, dice che se un segnale analogico a banda limitata e campionato con frequenza maggiore del doppio della banda, allora il segnale è rappresentato completamente dai propri campioni. Sulla base di questo teorema sono stati effettuati tre test riportati di seguito.

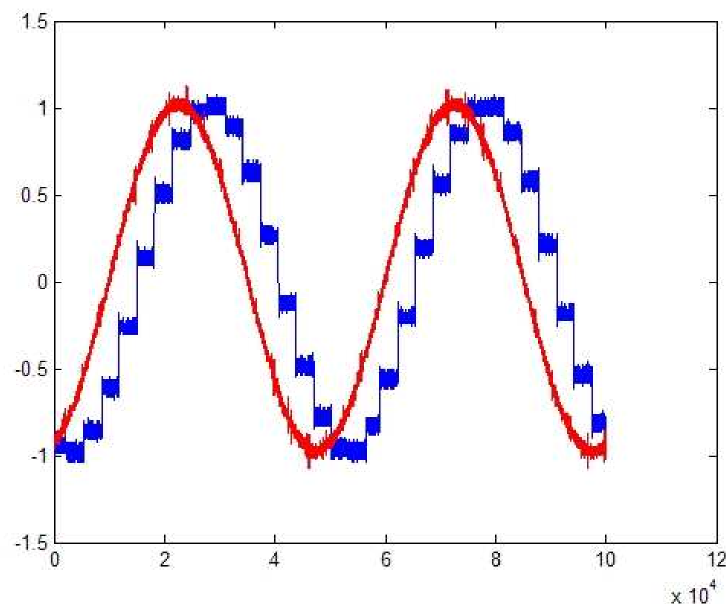

Figura 40: Primo test del driver

6.5.1 Primo test

Il primo test è illustrato in Figura 40. Il segnale in rosso è la sinusoide in ingresso all'ADC, quella in blu è la rispettiva uscita del DAC. La frequenza della sinusoide in ingresso è di 1kHz mentre la frequenza di campionamento è di 16kHz.

Rispettando il teorema di Shannon non si perdono campioni del segnale originale. Tuttavia c'è uno sfasamento tra i due segnali di 11µs, dovuto alla latenza tra l'acquisizione dall'ADC alla conversione del DAC.

6.5.2 Secondo test

Il secondo test è illustrato in Figura 41.

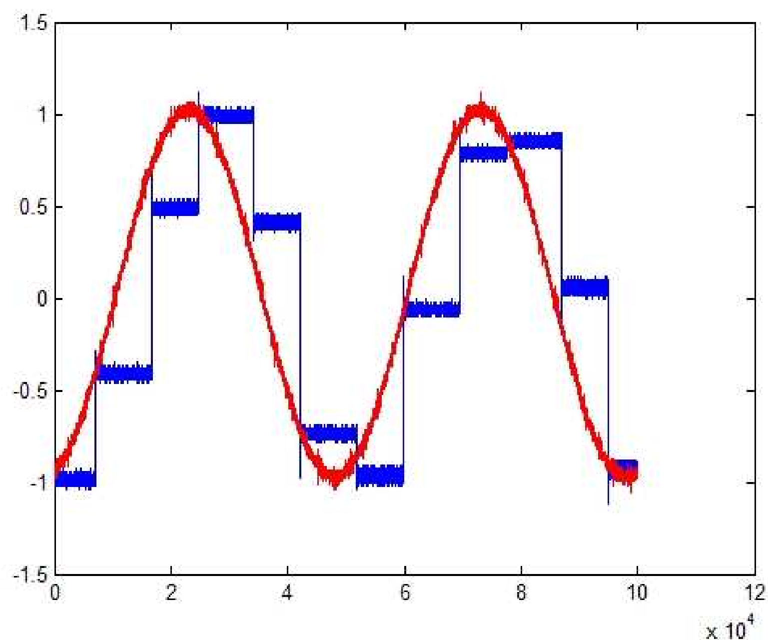

Figura 41: Secondo test del driver

Il segnale in rosso è la sinusoide in ingresso all'ADC, quella in blu è la rispettiva uscita del DAC. La frequenza della sinusoide in ingresso è di 1kHz mentre la frequenza di campionamento è di 20kHz.

Notiamo che nonostante siano state rispettate le ipotesi del teorema di Shannon, abbiamo perdita di campioni. Il motivo è dovuto al fatto che il driver per l'ADC non è in grado di gestire dati ad una frequenza di campionamento troppo elevata.

6.5.3 Terzo test

Il terzo test è illustrato in Figura 42.

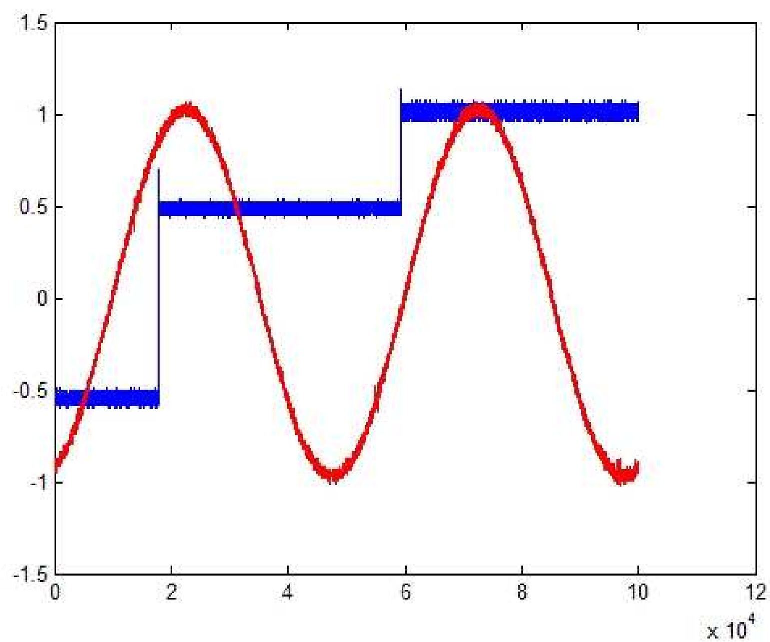

Figura 42: Terzo test del driver

Il segnale in rosso è la sinusoide in ingresso all'ADC, quella in blu è la rispettiva uscita del DAC. La frequenza della sinusoide in ingresso è di 1kHz mentre la frequenza di campionamento è di 1,2kHz.

Le ipotesi del teorema di Shannon in questo caso non sono rispettate e, di conseguenza, abbiamo un segnale in uscita diverso da quello in entrata.

Capitolo 7

Sistema di trasmissione dei dati

Il sistema progettato prevede l'elaborazione distribuita in rete. Quindi, una volta sviluppati i driver per l'unità di controllo, è stato necessario progettare un sistema di comunicazione per connettere le due unità.
Questo capitolo, dopo aver introdotto le reti tra computer e i loro protocolli, presenterà la struttura del sistema di comunicazione, dando una giustificazione delle scelte fatte.

7.1 Modelli di riferimento delle reti

Questa sezione vuole essere un'introduzione alle reti di calcolatori; si tratteranno superficialmente i modelli di riferimento alla base dell'infrastruttura delle reti, cioè il modello OSI e il modello TCP/IP.

7.1.1 Il modello di riferimento OSI

L'architettura di Internet cerca di riprendere molti aspetti del modello OSI (Open System Interconnection), proposto dalla ISO (International Standards Organizzation), ma in realtà esso non è usato e nemmeno implementato poiché ha regole rigide e vincolanti. Tuttavia il modello OSI è un buon modello di riferimento utile allo studio delle reti e per il confronto con altre architetture di rete come il famoso TCP/IP.

Il modello OSI ha sette strati; i principi che sono stati applicati per arrivare ai sette strati si possono riassumere come segue:

- Si deve creare uno strato quando è richiesta un'astrazione diversa;
- Ogni strato deve svolgere una funzione ben definita;
- Evitare la coesistenza di funzioni distinte nello stesso strato;
- I confini degli strati sono stati scelti per minimizzare il flusso d'informazioni tra le interfacce degli stessi.

Gli strati del modello di riferimento OSI sono: Fisico, Data-link, Network, Trasporto, Sessione, Presentazione, Applicazione. Alcuni di essi in comune con l'architettura di rete TCP/IP e saranno descritti nel corso di questa sezione.
Il modello OSI, non essendo mai stato implementato del tutto, non è in sé un'architettura di rete, perché non specifica quali sono esattamente i protocolli e i servizi da usare in ciascuno stato; si limita, infatti, a descrivere ciò che ogni strato deve compiere.

Livello	OSI	TCP/IP
7	Applicazione	Applicazione
6	Presentazione	/
5	Sessione	/
4	Trasporto	Trasporto
3	Network	Internet
2	Data-link	Hosto-to-network
1	Fisico	

Tabella 18: Confronto tra i modelli di riferimento OSI e TCP/IP

7.1.2 Il modello TCP/IP

Il modello TCP/IP, a differenza del modello di riferimento OSI, è anche un architettura di rete poiché per ogni suo strato sono definiti e implementati uno o più protocolli. Questa suite di protocolli è la progenitrice di tutte le reti geografiche tra computer. Iniziò dando vita ad ARPANET (storica rete sperimentale sponsorizzata dal dipartimento della difesa

USA), fino ad arrivare ad Internet. TCP/IP è oggigiorno utilizzato dalla stragrande maggioranza dei computer, e il suo stack di protocolli è supportato da quasi tutti i sistemi operativi.
L'architettura di rete TCP/IP utilizza quattro strati: Host-to-network, Internet, Trasporto, Applicazione. Abbiamo un confronto tra i modelli di riferimento OSI e TCP/IP nella Tabella 18. Di seguito sarà descritta la funzione di ogni strato di TCP/IP.

Lo strato Host-to-network

Questo strato è un gran vuoto nel modello TCP/IP, in quanto l'unica cosa che viene detta a questo livello è che il suo compito è quello di trasportare i pacchetti IP sulla rete mediante un qualsiasi protocollo.
La commissione IEEE 802 LAN/MAN è preposta per sviluppare standard per le reti locali (LAN) e metropolitane (MAN). La suddetta commissione è composta da vari gruppi, alcuni dei quali hanno sviluppato standard molto famosi tra cui Ethernet (802.3), Token Ring (802.5), e WLAN (802.11).
Un protocollo di comunicazione molto diffuso è il protocollo LLC definito nello standard 802.2 che, facendo riferimento al modello OSI, costituisce la parte superiore del livello collegamento data-link. Il livello superiore è quello di rete (chiamato Network nel modello OSI, e Internet nel modello TCP/IP), mentre il livello inferiore non coincide con quello fisico OSI, ma si avvale del MAC (Media Access Control), che costituisce la parte inferiore del livello data-link e contiene funzioni di controllo dell'accesso al mezzo fisico.

Lo strato Internet

Il dipartimento della difesa statunitense aveva il timore che alcuni dei suoi preziosi host, router, e gateway tra reti potessero essere distrutti all'improvviso, chiese di progettare una rete capace di restare in funzione anche in caso di perdita di hardware.
Tenuto conto di questo requisito, TCP/IP implementa una rete a commutazione di pacchetto basata su uno strato Internet senza connessione. Lo strato Internet, molto simile allo strato Network del modello OSI, fornisce un formato per i pacchetti e un protocollo

chiamato IP (Internet Protocol): uno dei protocolli fondamentali dell'intero stack, che assieme al protocollo TCP (descritto più avanti) dà il nome all'architettura di rete TCP/IP.

Lo strato Trasporto

Lo strato Trasporto è progettato come l'omonimo strato del modello OSI. L'architettura di rete TCP/IP non ha gli strati Sessione e Presentazione, quindi uno sviluppatore di software di rete interagisce molto con questo strato, facendo largo uso dei protocolli che esso offre: TCP e UDP.

La scelta di usare un protocollo o l'altro è stata di fondamentale importanza durante lo sviluppo del nostro sistema, per questo la presentazione di questi protocolli è data più avanti.

Lo strato Applicazione

Lo strato Applicazione è direttamente interfacciato con lo strato Trasporto, e contiene i protocolli di rete utilizzati per implementare servizi molto conosciuti ed utilizzati come http, telnet, ftp, smtp, dns, ed altri ancora.

7.2 Protocolli offerti dallo strato Trasporto di TCP/IP

Come appena mostrato l'infrastruttura di rete TCP/IP è un insieme di protocolli diversi, che operano su quattro strati diversi. Lo strato Trasporto è quello più vicino al programmatore, il quale ogni volta che scrive un applicazione di rete, deve decidere se utilizzare il protocollo di trasporto TCP, oppure quello UDP.

In questa sottosezione saranno descritti i protocolli TCP ed UDP; alla fine sarà motivata la scelta fatta nell'implementazione dei programmi di comunicazione tra le unità di controllo e di calcolo.

7.2.1 Il protocollo di trasporto TCP

TCP (Transmission Control Protocol) è il protocollo di trasporto, definito nel RFC[3] 793, su cui si appoggiano gran parte delle applicazioni Internet. I suoi punti di forza sono l'alta affidabilità e robustezza; la sua popolarità si deve anche grazie ad una sua implementazione diffusa dalla "Berkeley University of California" sotto forma di sorgenti di libero uso. Le caratteristiche principali del TCP sono:

- La creazione di una connessione (protocollo orientato alla connessione);
- La gestione di connessioni punto-punto;
- La garanzia che i dati trasmessi giungano a destinazione in ordine e senza perdita d'informazione (tramite il meccanismo di acknowledgment e ritrasmissione);
- Attraverso il meccanismo della finestra scorrevole, offre funzionalità di controllo di flusso e controllo della congestione, vitali per il buon utilizzo della rete IP, che non offre alcuna garanzia in ordine alla consegna dei pacchetti, al ritardo, alla congestione;
- Una funzione di multiplazione delle connessioni ottenuta attraverso il meccanismo delle porte.

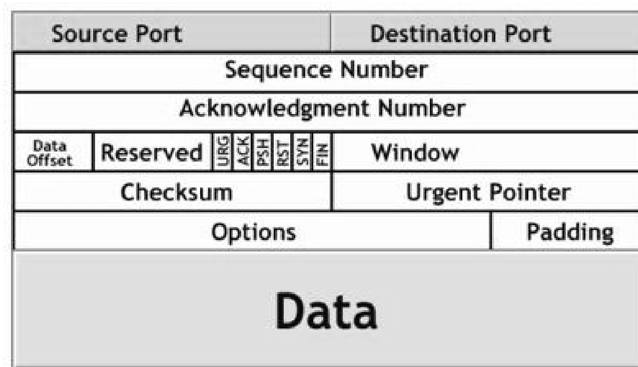

Figura 43: Contenuto di un pacchetto TCP

[3] Un Request for Comments (RFC) è un documento che riporta informazioni o specifiche messe a disposizione della comunità Internet. Gli RFC furono inizialmente pubblicati nel 1969 come parte del progetto ARPANET.

Intestazione di un pacchetto TCP

L'intestazione di un pacchetto TCP (header), illustrata in Figura 43, è così strutturata:

- Porta sorgente (Source port), campo di 16 bit;
- Porta di destinazione (Destination port), campo di 16 bit;
- Numero di sequenza (Sequence number), campo di 32 bit, indica la posizione del primo byte di dati del segmento TCP all'interno del flusso completo; se il flag SYN è impostato, il valore del sequence number corrisponde all'Initial Sequence Number (ISN);
- Numero di acknowledgment (Acknowledgment number), campo di 32 bit, contiene il valore del prossimo sequence number che la sorgente del segmento TCP è in attesa di ricevere ed è utilizzato congiuntamente al flag ACK;
- Data offset, campo di 4 bit, indica la lunghezza (in word da 32 bit) dell'header del segmento TCP;
- 6 bit riservati (Reserved), non utilizzati e predisposti per sviluppi futuri del protocollo;
- 6 bit di controllo (Control bits), possono essere impostati ad 1 o 0 e indicano:
  - URG: il valore dell'urgent pointer è valido;
  - ACK: il valore dell'acknowledgment number è valido;
  - PSH: l'host che riceve il segmento TCP deve provvedere a trasferire i dati allo strato Applicazione il più velocemente possibile;
  - RST: reset della connessione;
  - SYN: se impostato, indica che si tratta del primo segmento della connessione;
  - FIN: se impostato, indica che si tratta dell'ultimo segmento della connessione;
- Finestra (Window), campo di 16 bit, indica il numero di byte che il destinatario è in grado di accettare a partire dal byte indicato dall'acknowledgment number;
- Checksum, campo di 16 bit, utilizzato per il controllo della validità del segmento;

- Urgent pointer, campo di 16 bit, puntatore al sequence number di dati con priorità di trasferimento;
- Opzioni (facoltative)
- Padding, campo dinamico utilizzato per completare i bit non utilizzati dalle opzioni

Instaurazione della connessione

La procedura utilizzata per instaurare in modo affidabile una connessione TCP tra due host è chiamata three-way handshake (triplice stretta di mano), ad indicare la necessità di scambiare tre messaggi per garantire la corretta creazione della connessione.

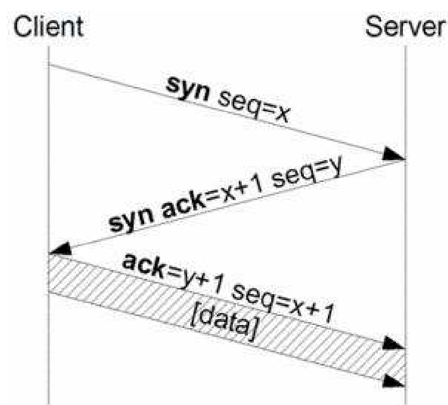

Figura 44: Procedura three-way handshake

Seguendo lo schema in Figura 44, supponiamo, per esemplificare, che l'host A (il client) intenda instaurare una comunicazione TCP con l'host B (il server); i passi indicati dalla tecnica three-way handshake sono:

1. A invia un segmento SYN a B, contenente il suo sequence number x;
2. B invia un segmento SYN/ACK ad A, contenente il suo sequence number y e l'acknowledgment del sequence number x di A;
3. A invia un segmento ACK a B con l'acknowldedgment del sequence number y di B.

La procedura three-way handshake è una delle differenze fondamentali tra il protocollo TCP e quello UDP illustrato di seguito.

7.2.2 Il protocollo di trasporto UDP

L'UDP (User Datagram Protocol) è un protocollo di trasporto a pacchetto (non orientato alla connessione), definito nel RFC 768.

Figura 45: Contenuto di un pacchetto UDP

Come TCP, UDP possiede una funzione di multiplazione delle connessioni ottenuta attraverso il meccanismo delle porte. A differenza del TCP, non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi. L'UDP ha come caratteristica di essere un protocollo di rete molto efficiente ma poco affidabile. È usato spesso per la trasmissione d'informazioni audio o video.
Infine, siccome UDP è un protocollo non orientato alla connessione, si hanno situazioni in cui un client può scrivere pacchetti destinati a server diversi sulla stessa socket, e dall'altro lato un server può ricevere su una socket pacchetti provenienti da client diversi.

7.2.3 Protocollo scelto per il nostro sistema

Il sistema progettato, per la comunicazione tra l'unità di calcolo e quella di controllo, è interamente fondato sul protocollo UDP. Questo è dovuto a due fattori fondamentali:

1. Il nostro sistema mira alla velocità di trasmissione dei dati, e UDP ha prestazioni molto superiori a TCP su quest'aspetto;

2. Un altro scopo è quello di ridurre al minimo gli errori, e su questo UDP non ci è di aiuto. Visto che l'unità di controllo è collegata all'unità di calcolo mediante un solo cavo di rete, quindi siamo in presenza di un collegamento punto-punto, il numero di errori su rete è praticamente nullo.

I test svolti, commentati verso la fine di questo capitolo, confermano quanto appena detto, per questo il protocollo UDP è stata una buona scelta.

7.3 Struttura dei programmi di comunicazione

Si sono scritti due programmi, uno per l'unità di controllo (chiamato, appunto, *controllo*), e l'altro per l'unità di calcolo (chiamato *calcolo*). Il processo sull'unità di controllo ha il compito di acquisire i segnali elettrici analogici da 32 canali mediante il dispositivo ADC, inviarli all'unità di calcolo per l'elaborazione distribuita in rete, quindi dopo l'eventuale applicazione di un filtro digitale da parte di quest'ultima, ricevere i dati elaborati da rete, e, infine, riconvertire i dati in segnali elettrici analogici mediante i dispositivi DAC.
Adottando la programmazione di rete in ambiente UNIX, siamo in grado di far comunicare processi diversi in esecuzione su diverse macchine. Qui nasce il problema di scrivere entrambi i programmi (uno per l'unità di calcolo e l'altro per l'unità di controllo) in modo che essi comunichino nel modo più veloce possibile. Le soluzioni proposte sono quattro e si distinguono in base al numero di socket utilizzate, e al numero di thread o processi.

Soluzione	Processi unità di calcolo	Processi unità di controllo	Thread unità di controllo	Socket utilizzate
1	1	2	-	2
2	1	1	-	2
3	1	1	2	1
4	1	1	-	1

Tabella 19: Soluzioni sviluppare proposte il sistema

In tutte le soluzioni è stato utilizzato il protocollo UDP poiché la velocità di trasmissione delle informazioni è uno degli obiettivi principali del nostro lavoro. Essendo la connessione tra le due unità di tipo punto-punto, la perdita di pacchetti è in pratica inesistente, quindi non abbiamo bisogno di un protocollo di comunicazione come TCP con un rigido controllo sugli errori. Inoltre, siccome il DAC mantiene l'ultimo valore finchè questo non viene cambiato, possiamo permetterci di perdere qualche pacchetto con il risultato di avere un limite alla frequenza di attuazione che è determinato dal tempo di round-trip dei pacchetti.
Guardando la Tabella 19, notiamo che le soluzioni differiscono per il numero di socket utilizzate, e per il numero di task in esecuzione sull'unità di controllo. Dopo aver implementato e testato le quattro soluzioni, si è scelto di adottare la terza; di seguito saranno analizzate le quattro soluzioni, ed infine sarà chiarito il motivo della scelta fatta.

7.3.1 Prima soluzione

Per compiere il test, si è implementata la comunicazione tra le unità come previsto nella prima soluzione, e, sono stati inviati dati generati in modo casuale su rete. Nel pacchetto trasferito, di dimensione fissa, sono stati inseriti due campi addizionali: il tempo e il numero del pacchetto. In questo modo l'unità di controllo può sapere quanti pacchetti sono stati persi e con quale velocità sono stati trasmessi e ricevuti (tempo di round-trip). Alla fine, dopo aver collezionato i dati sui tempi di round-trip e sui pacchetti persi, è stata effettuata la statistica riportata in Tabella 20.

Pacchetti spediti	10000
Pacchetti persi	0
Tempo medio	334μs
Scarto	5μs

Tabella 20: Risultati del test con la prima soluzione

7.3.2 Seconda soluzione

Unificando i due processi in esecuzione sull'unità di controllo della prima soluzione, abbiamo risultati molto migliori (Tabella 21). Resta lo stesso l'utilizzo di due socket diverse, cosa alquanto inutile giacché ogni socket di per sé rappresenta un canale bidirezionale.

Pacchetti spediti	10000
Pacchetti persi	0
Tempo medio	120μs
Scarto	4μs

Tabella 21: Risultati del test con la seconda soluzione

I risultati ottenuti sono migliori poiché non è più presente la perdita di tempo introdotta dal cambio di contesto tra i due processi in esecuzione sull'unità di controllo.

7.3.3 Terza soluzione

La terza soluzione è la prima ad utilizzare una sola socket. Introduce l'uso di thread nell'unità di controllo, e, per questo motivo, abbiamo un tempo medio solo leggermente superiore a quello avuto con un unico processo: il tempo di cambio di tra thread è quindi trascurabile.

Pacchetti spediti	10000
Pacchetti persi	0
Tempo medio	126μs
Scarto	5μs

Tabella 22: Risultati del test con la terza soluzione

7.3.4 Quarta soluzione

Questa soluzione è molto simile alla seconda, con l'unica differenza che utilizza un'unica socket. Il tempo medio di trasmissione è pressoché lo stesso, con un piccolo miglioramento

riguardante lo scarto quadratico medio. Questo ci fa capire che l'utilizzo di una oppure due socket non incide molto sui tempi di trasmissione dei dati.

Pacchetti spediti	10000
Pacchetti persi	0
Tempo medio	120μs
Scarto	4μs

Tabella 23: Risultati del test con la quarta soluzione

7.3.5 Soluzione adottata per l'implementazione del sistema

La seconda e la quarta soluzione ci offrono il miglior tempo medio di trasmissione dei dati, quindi si è tentati a scegliere una delle due. Abbiamo invece scelto la terza soluzione che offre un tempo medio di trasmissione dei dati superiore di *6μs* (cioè del 5%). In compenso otteniamo un ottimo grado di flessibilità: si può far aumentare il numero di compiti da svolgere all'unità di controllo semplicemente aggiungendo un thread.

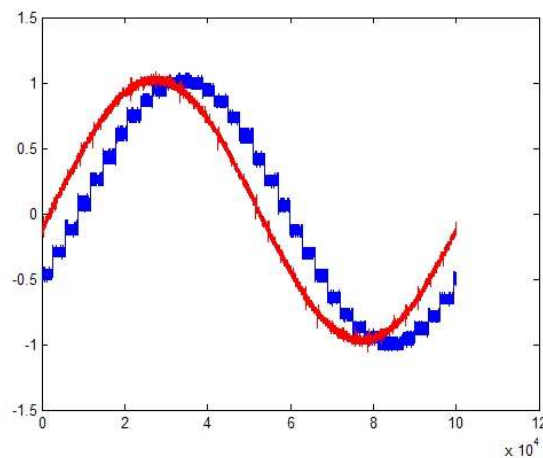

Figura 46: Test del sistema di acquisizione
Sull'asse delle ascisse troviamo i numeri dei campioni dell'oscilloscopio, mentre sull'asse delle ordinate sono riportati i volts

Adottando la terza soluzione è stato fatto un semplice test di trasmissione dati per porre sotto stress il sistema creato. Il test consiste nella sola acquisizione dei dati mediante il dispositivo ADC, trasmissione e ricezione dei dati su socket, ed attuazione degli stessi mediante i dispositivi DAC, tutto senza applicare nessun filtro ai dati acquisiti. Nell'immagine in Figura 46 vediamo in rosso la sinusoide in ingresso all'ADC, quella in blu in uscita dal DAC. La frequenza del segnale in rosso è di 200Hz mentre la frequenza di campionamento è di 6,2kHz. Questo limite della frequenza di campionamento è da imputarsi grosso modo al tempo di round-trip sul collegamento UDP. La larghezza dei gradini della curva in uscita è 126µs, consistente con il tempo di round-trip misurato con il programma di test descritto nel paragrafo 7.3.3.

7.4 Descrizione della soluzione scelta

Di seguito è descritta l'implementazione della soluzione scelta per il nostro sistema, cioè quella che prevede l'utilizzo di un'unica socket dal lato controllo, e l'uso di due thread distinti: uno per scrivere i dati sulla socket e l'altro per leggerli.

7.4.1 Interfacce interne comuni

Le strutture dati e le interfacce in comune tra i task in esecuzione sull'unità di controllo e il processo in esecuzione sull'unità di calcolo riguardano la trasmissione delle informazioni su socket e il loro formato.

I dati sono rappresentati usando la seguente struttura:

```
typedef struct dati {
      unsigned short vett[NUMCHAN];
      #ifdef STAT
      struct timeval time;
      long num_pkg;
      #endif
} dati_t;
```

I campi *time* e *num_pkg* sono opzionali: sono stati utilizzati per svolgere i test sul sistema, ed è possibile accedervi solo se compiliamo il codice sorgente definendo la macro *STAT*.
La struttura appena presentata ha il compito di contenere un campione di dati per ogni canale nel vettore *vett*. L'unità di calcolo accetterà i dati in questo formato, applicherà ad essi un filtro digitale, e ritrasmetterà la struttura all'unità di controllo.
Per implementare la comunicazione mediante socket con protocollo UDP ci serviamo della seguente struttura dati:

```
typedef struct comunicazione {
     int                  sock_fd;
     struct sockaddr_in   servaddr;
     struct sockaddr_in   cliaddr;
     char                 ip[IP_LEN];
     int                  portnum;
} comunicazione_t;
```

Dalla specifica fornita durante l'ingegnerizzazione del processo, vediamo che quest'ultima struttura è utilizzata dalle funzioni per la comunicazione tra le unità: di seguito sono riportati i prototipi:

```
int connetti(comunicazione_t *connessione, int mode);
int disconnetti(comunicazione_t *connessione, int mode);
int trasmetti(comunicazione_t *connessione, void *buffer,
              size_t len, int mode);
int ricevi(comunicazione_t *connessione, void *buffer,
           size_t len, int mode);
```

Esse prevedono il parametro mode, il quale prevede che l'utilizzatore possa scegliere il tipo di protocollo da usare per la comunicazione tra le unità. Ad esempio esaminiamo il codice sorgente della routine *connetti()*:

```
int connetti(comunicazione_t *connessione, int mode) {

     switch (mode) {
          case UDP :
               return connetti_udp(connessione);
               break;
```

```
        /* Supporto per le reflective memory
        case RM :
            return connetti_rm(connessione);
            break;
        */
    }

    return -1; // Modo di comunicazione non implementato
}
```

Finora è implementato solo il supporto per il protocollo UDP, ma con questa struttura sarà molto semplice inserire un nuovo modo di comunicazione tra le due unità, come ad esempio le reflective memory recentemente acquistate per migliorare il sistema.
Le funzioni che implementano la comunicazione usando il protocollo UDP sono:

```
int connetti_udp(comunicazione_t *connessione);
int disconnetti_udp(comunicazione_t *connessione);
int trasmetti_udp(comunicazione_t *connessione, void *buffer,
                  size_t len);
int ricevi_udp(comunicazione_t *connessione, void *buffer,
               size_t len);
```

Queste funzioni non sono implementate nella parte comune poiché il codice dell'unità di controllo differisce dal codice dell'unità di calcolo. In effetti, siccome il protocollo UDP non è orientato alla connessione, ogni pacchetto spedito dall'unità di controllo prevede le seguenti operazioni:

- collega la socket;
- spedisci il primo pacchetto;
- disconnetti la socket.

In questo modo si perde molto tempo prezioso. La soluzione a questo problema prevede si usare la funzione *connect()*, la quale collega la socket ad un determinato indirizzo e poi la utilizza fin quando non si decide di sconnetterla:

- collega la socket;
- spedisci tutti i pacchetti necessari;
- disconnetti la socket;

Utilizzando la funzione *connect()*, abbiamo un ulteriore incremento delle prestazioni.

7.4.2 Moduli per l'unità di calcolo

I moduli principali per l'unità di calcolo sono quello riguardante la comunicazione con il protocollo UDP, e quello riguardante il filtro digitale.

Modulo per la comunicazione

Il modulo per la comunicazione prevede l'uso delle routine *connetti()*, *ricevi()*, *trasmetti()*, *disconnetti(),* che a loro volta richiameranno le routine *connetti_udp()*, *ricevi_udp()*, *trasmetti_udp()*, *disconnetti_udp()*, tutte implementate nel file *trasmissione_calcolo.c*.

connetti_udp()

Questa funzione ha lo scopo di mettere l'unità di calcolo su una determinata porta in attesa di richieste proveniente dall'unità di controllo.
L'unico parametro che richiede in ingresso è un puntatore alla struttura *comunicazione_t* descritta in precedenza, e ritorna un valore negativo in caso d'errore.

disconnetti_udp()

Poiché il protocollo UDP non è un servizio orientato alla connessione, questa funzione non ha nessun compito: è stata inserita per permettere la coesistenza di più tipi di protocollo.

ricevi_udp()

La routine *ricevi_udp()* ha lo scopo di ricevere un certo numero di byte (stabilito dal parametro *len*) dalla rete, e restituirli in output mediante un buffer. La funzione ritorna il numero di byte effettivamente letti.

Per funzionare questa routine si serve della funzione *recvfrom()*.

trasmetti_udp()

Simile alla funzione *ricevi_udp()*, la routine *trasmetti_udp()* ha il compito di trasmettere un certo numero di byte su rete. In questo caso ci serviamo della funzione *sendto()*, ed è ritornato in output il numero di byte effettivamente trasmessisi rete.

Modulo per il filtro digitale

In questa sottosezione sarà illustrata la specifica di un esempio di filtro, per consentire, in futuro, l'inserimento in modo semplice, del codice per un filtro più complesso. Per l'implementazione del filtro in questione, è necessario un buffer circolare, la cui struttura dati (Tabella 24) e prototipi delle funzioni sono definite nel file *buffer.h*.

Nome struttura	Parametro	Descrizione
belement_t	**array*	puntatore al buffer
	dim	dimensione buffer
	in	prima posizione in cui inserire un nuovo elemento
	out	posizione del primo elemento in uscita dal buffer

Tabella 24: Struttura dati buffer circolare

Le funzioni per la gestione del buffer sono riportate in Tabella 25. Se ne riporta solo un elenco non si scende ulteriormente nei dettagli di ognuna, data la loro semplicità.

Funzione	Descrizione
adtb_get_buffer()	crea un buffer
adtb_free_buffer()	elimina il buffer creato
adtb_empty()	indica se il buffer è vuoto
adtb_max_dim()	indica la dimensione del buffer allocato
adtb_current_dim()	indica il numero di elementi correnti nel buffer
adtb_insert()	inserisce un elemento nel buffer
adtb_read()	estrae un elemento dal buffer
adtb_update()	modifica un elemento del buffer

Tabella 25: Funzioni per la gestione del buffer

La funzione *filtro_ma()*, implementata nel file *filtro.c* utilizza le funzioni per la gestione del buffer e deve:

- inserire un dato nel buffer;
- eseguire la media aritmetica degli elementi presenti nel buffer;
- ritornare la media calcolata.

7.4.3 Moduli per l'unità di controllo

I soli due moduli principali per l'unità di controllo sono quello riguardante la comunicazione con il protocollo UDP, e quello riguardante l'acquisizione e l'attuazione dei dati mediante i dispositivi ADC e DAC.

Modulo per la comunicazione

Il modulo principale per la comunicazione di questa unità prevede anch'esso l'uso delle routine *connetti()*, *ricevi()*, *trasmetti()*, *disconnetti(),* che a loro volta richiameranno le routine *connetti_udp()*, *ricevi_udp()*, *trasmetti_udp()*, *disconnetti_udp()*, diverse da quelle implementate nell'unità di calcolo e tutte implementate nel file *trasmissione_controllo.c*.

connetti_udp()

Questa routine ha lo scopo di associare una socket in modo permanente alla socket aperta sull'unità di calcolo per i motivi discussi in precedenza in questo capitolo. Si fa uso della funzione *connect()*, ed è ritornato 0 se l'operazione è andata a buon fine.

disconnetti_udp()

Ancora una volta la funzione *disconnetti_udp()* non ha nessun compito. La routine *connect()* usata in precedenza, infatti, non stabilisce una vera e propria connessione con l'unità di calcolo poiché il protocollo UDP non è orientato alla connessione, quindi non è necessario disconnettere la socket.

ricevi_udp()

In questo caso, siccome la socket è collegata esplicitamente all'unità di calcolo grazie alla routine *connect()*, non è utilizzata *recvfrom()* per ricevere i dati, ma bensì la funzione *read()*. La routine *ricevi_udp()* restituisce il numero di byte effettivamente ricevuti.

trasmetti_udp()

Simile alla funzione ricevi_udp(), anch'essa differisce dalla versione implementata lato calcolo. Si serve questa volta della funzione *write()* al posto di *sendto()* per trasmettere i dati. La routine *trasmetti_udp()* restituisce il numero di byte effettivamente trasmessi.

Moduli riguardanti l'acquisizione e l'attuazione dei dati

Per rendere indipendente il processo sull'unità di controllo, dal tipo e dal numero dei dispositivi fisici, si è creato un modulo separato. Le strutture dati usate in questo caso sono implementate nel file adcdac.h; le più importanti sono riportate di seguito:

```
typedef struct adcfd {
     int fd;
}adcfd_t;
```

```
typedef struct dacfd {
     int fd[NUMDAC];
}dacfd_t;
```

La struttura *adcfd_t* contiene il file descriptor del dispositivo ADC, che nel costro caso è unico ed ha 32 canali. La struttura *dacfd_t*, invece, contiene quattro file descriptor (NUMDAC vale 4) poiché i nostri dispositivi DAC MPV955 hanno ognuno 8 canali.

Funzione	Descrizione
adc_open()	apre il dispositivo ADC
adc_close()	chiude il dispositivo ADC aperto
adc_read()	legge i canali del dispositivo ADC aperto
dac_open()	apre il dispositivo DAC
dac_close()	chiude il dispositivo DAC aperto
dac_write()	scrive nel dispositivo DAC i dati da convertire

Tabella 26: Funzioni per l'utilizzo dei dispositivi ADC e DAC

Prevedendo le strutture per i file descriptor si può facilmente cambiare l'implementazione delle funzioni per accedere ai dispositivi senza cambiare la loro interfaccia.
Le funzioni offerte da questo modulo sono descritte in Tabella 26.

Capitolo 8

Controllo remoto del sistema

Impartire ogni volta i comandi su terminali diversi per mandare in esecuzione i processi sull'unità di calcolo e quella di controllo necessari per l'acquisizione, elaborazione, ed attuazione dei dati è molto scomodo e frustante.
Si è pensato di introdurre un modo per controllare l'intero sistema da remoto, mediante una comoda interfaccia di comando su rete. Questo capitolo descrive ampiamente l'interfaccia di comando e l'infrastruttura adottata per la realizzazione.

8.1 Struttura del sistema di interfaccia

Per controllare l'unità di calcolo e quella di controllo da remoto è stato necessario scrivere due speciali programmi sempre in esecuzione su di esse: questo tipo di programma è chiamato "demone", e sono stati già introdotti nel capitolo 1 e 5. In questa sezione si provvederà a descrivere in modo dettagliato il funzionamento di questi programmi.
I demoni sono essenzialmente dei server; normalmente si dividono i server in due categorie:

- *Iterativi*: che rispondono alla richiesta e restano in attesa fintanto non hanno concluso la richiesta;
- *Concorrenti*: che al momento di trattare una richiesta creano un nuovo thread (o processo) e si mettono in attesa d'altre richieste. In questo modo sono in grado di soddisfare più richieste contemporaneamente.

I programmi demoni implementati sono server di tipo iterativo, poiché la risorsa è unica e non è possibile soddisfare più richieste nello stesso momento. L'interfaccia di comando è il client capace di fare richieste ai demoni in esecuzione su ciascuna delle due unità (Figura 47).

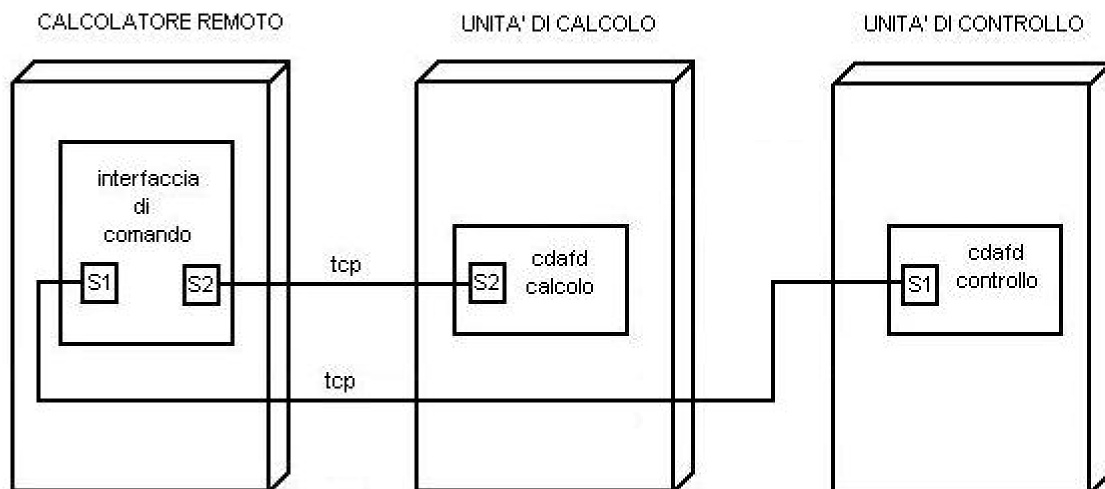

Figura 47: Struttura del sistema di comando

I demoni sono stati scritti in linguaggio ANSI C, rispettando le specifiche POSIX 1003.1, .1b e .1c, mentre l'interfaccia di comando, per assicurare la massima portabilità, è stata scritta in linguaggio Java. Nel seguito di questo capitolo si presenteranno le caratteristiche dei programmi "demoni", il linguaggio per l'interfaccia di comando, e infine si esamineranno i problemi riscontrati nel far comunicare tra loro due processi scritti in diversi linguaggi di programmazione.

8.2 Programmi demoni

Nei sistemi di tipo Unix, e più in generale nei sistemi operativi multitasking, un demone (daemon in inglese) è un programma eseguito in background, senza che sia sotto il

controllo diretto dell'utente. Di solito i demoni hanno nomi che finiscono per "d": per esempio, nel nostro caso *cdafd* è il demone in esecuzione su entrambe le unità.
Spesso i demoni vengono avviati al boot del sistema: in generale hanno la funzione di rispondere a determinate richieste, che siano di rete, hardware, e di altro tipo.
Il termine demone è stato coniato facendo riferimento a personaggi della mitologia greca, appunto i demoni, i quali eseguivano compiti di cui gli dei non potevano occuparsi, esattamente come i demoni eseguono compiti in background di cui l'utente non vuole o non può occuparsi.
In senso strettamente tecnico, Unix considera come demone qualsiasi processo che abbia come genitore il processo numero 1 (cioè *init*). Poiché *init* adotta ogni processo il cui genitore termini senza aspettare lo stato del processo figlio, il metodo comune per lanciare un demone è eseguire una chiamata di sistema *fork()* due volte e poi far terminare il processo genitore, mentre il figlio continua normalmente l'esecuzione. Questo idioma di programmazione è talvolta descritto con l'espressione inglese “fork off and die”.
In ambiente DOS, l'analogo dei demoni erano i programmi TSR (Terminate and Stay Resident, ovvero “termina e resta in memoria”), mentre nel sistema operativo Microsoft Windows, i programmi che svolgono le funzioni di demone sono chiamati “servizi”.

8.3 Linguaggio di programmazione Java

Per implementare l’interfaccia di comando è stato scelto il linguaggio di programmazione Java, il quale è molto conosciuto per la sua portabilità su varie piattaforme. In questa sezione si descriveranno le caratteristiche del linguaggio, facendo un accenno al paradigma di programmazione a oggetti, un concetto sul quale si basa il linguaggio Java.

8.3.1 Programmazione orientata agli oggetti

La programmazione orientata agli oggetti (*OOP*, in inglese Object Oriented Programming) è un paradigma di programmazione, che prevede di raggruppare in un'unica entità (la

classe) sia le strutture dati sia le procedure che operano su di esse, creando per l'appunto un "oggetto" software dotato di proprietà (dati) e metodi (procedure) che operano sui dati dell'oggetto stesso.

Adottando questo paradigma, la modularizzazione di un programma è realizzata nel momento in cui progettiamo il programma stesso sotto forma di classi che interagiscono tra loro.

Per la sua natura, la programmazione orientata agli oggetti è particolarmente adatta a realizzare interfacce grafiche, nella quale ogni oggetto rappresentato sullo schermo rappresenta, di fatto, un'*istanza* di una *classe* descritta nel programma, appunto un *oggetto*.

La *classe* può essere considerata l'erede del tipo di dato astratto, una tendenza che si è sviluppata all'interno del paradigma della programmazione procedurale, secondo la quale un modulo dovrebbe implementare un tipo di dato definito dall'utente, con cui si possa interagire solo attraverso un'interfaccia ben definita, che nasconda agli altri moduli i dettagli dell'implementazione, in modo che sia possibile modificarli contenendo gli effetti della modifica sul resto del programma. La classe può essere vista come il costrutto che permette di realizzare quest'astrazione con un supporto strutturato da parte del linguaggio di programmazione stesso.

Il primo linguaggio di programmazione orientato agli oggetti fu il Simula (1967), il quale ha avuto molti successori e negli anni '90 il paradigma ad oggetti è diventato quello più utilizzato, per cui gran parte dei linguaggi di programmazione erano nativamente orientati agli oggetti o avevano una estensione in tal senso. Tra i linguaggi orientati agli oggetti troviamo, appunto, il linguaggio Java usato per l'implementazione dell'interfaccia di comando del nostro sistema.

Un linguaggio ad oggetti deve avere le tre seguenti proprietà: incapsulamento, ereditarietà, polimorfismo. Parlare in modo approfondito di questi tre concetti richiede un volume intero (rif. [15]), quindi di seguito si farà solo una breve presentazione.

Incapsulamento

L'incapsulamento è la proprietà per la quale un oggetto contiene ("incapsula") al suo interno i dati e i metodi che accedono ai dati stessi. Lo scopo principale

dell'incapsulamento è appunto dare accesso ai dati incapsulati solo attraverso i metodi definiti, nell'interfaccia, come accessibili dall'esterno. Gestito in maniera intelligente, l'incapsulamento permette di vedere l'oggetto come una *black-box*, appunto una scatola nera di cui, attraverso l'interfaccia definita dai metodi, sappiamo cosa fa e come interagisce con l'esterno, ma non conosciamo nulla riguardo l'implementazione interna. I vantaggi principali portati dall'incapsulamento sono: robustezza, indipendenza, e l'estrema riusabilità del codice scritto.

Ereditarietà

La programmazione orientata agli oggetti prevede un meccanismo molto importante, l'ereditarietà, il quale permette di derivare nuovi tipi di dati a partire da classi già definite aggiungendo membri e modificando il comportamento di quelli esistenti. L'ereditarietà è un meccanismo importantissimo per gestire l'evoluzione ed il riuso del software, e su di essa si basano concetti importanti come il *polimorfismo*.

Polimorfismo

La possibilità che una classe derivata ridefinisca i metodi e le proprietà dei suoi antenati rende possibile che gli oggetti appartenenti ad una classe che ha delle sottoclassi rispondano diversamente alle stesse istruzioni: quanto appena detto è il concetto fondamentale sul quale si basa il polimorfismo. I metodi ridefiniti in una sottoclasse sono detti "polimorfi", in quanto lo stesso metodo si comporta diversamente a seconda del tipo di oggetto su cui è invocato. Le buone regole di programmazione ad oggetti prevedono che quando una classe derivata ridefinisce un metodo, il nuovo metodo deve avere, dal punto di vista degli utenti della classe, la stessa semantica di quello ridefinito.

Il polimorfismo è particolarmente utile quando la versione del metodo da eseguire viene scelta sulla base del tipo di oggetto effettivamente contenuto in una variabile a runtime, invece che al momento della compilazione. Questa funzionalità è detta *binding dinamico* (o *late-binding*), e richiede un grosso sforzo di supporto da parte della libreria run-time del linguaggio. Il binding dinamico è supportato dai più diffusi linguaggi di programmazione ad oggetti come Java e C++; c'è però da sottolineare che in Java il binding dinamico è

implicitamente usato come comportamento predefinito nelle classi polimorfe, mentre il C++ non usa il binding dinamico se non dichiarato esplicitamente dal programmatore nella segnatura del metodo interessato con la parola chiave *virtual*.

8.3.2 Caratteristiche del linguaggio Java

Il linguaggio Java è un linguaggio di programmazione orientato agli oggetti, creato da James Gosling ed altri ingegneri di Sun Microsystems. Il gruppo iniziò a lavorare nel 1991 e Java fu annunciato ufficialmente il 23 maggio 1995. La piattaforma di programmazione Java è fondata sul linguaggio stesso, sulla Java Virtual Machine (detta anche JVM, ed esaminata meglio nel seguito di questo capitolo) e sulle API.
Java venne creato per soddisfare quattro scopi:

1. Essere orientato agli oggetti;
2. Essere indipendente dalla piattaforma;
3. Contenere strumenti e librerie per il networking;
4. Essere progettato per eseguire codice da sorgenti remote in modo sicuro.

La prima caratteristica è l'orientamento agli oggetti, il moderno metodo di programmazione e progettazione appena descritto. La seconda caratteristica, l'indipendenza dalla piattaforma, significa che l'esecuzione di programmi scritti in Java deve avere un comportamento simile su hardware diverso. Si dovrebbe essere in grado di scrivere il programma una volta e farlo eseguire dovunque. Questo è possibile con la compilazione del codice di Java in un linguaggio intermedio, il bytecode, basato su istruzioni semplificate che ricalcano il linguaggio macchina. Esso è eseguito da una virtual machine, cioè da un interprete: Java è quindi, in linea di massima, un linguaggio interpretato. Inoltre, sono fornite librerie standardizzate per permettere l'accesso alle caratteristiche della macchina in modo unificato. Il linguaggio Java include anche il supporto per i programmi con multithread, necessario per molte applicazioni che usano la rete. La portabilità è un obiettivo tecnicamente difficile da raggiungere, e il successo di Java in quest'ambito è

materia d’alcune controversie. Sebbene sia, in effetti, possibile scrivere in Java programmi che si comportano in modo consistente attraverso molte piattaforme diverse, bisogna tenere presente che questi poi dipendono dalle virtual machine, che sono programmi a sé e che hanno inevitabilmente i loro malfunzionamenti, diversi dall'una all'altra.

Le prime implementazioni del linguaggio usavano una virtual machine che intepretava il bytecode per ottenere la massima portabilità. Questa soluzione si è però rivelata poco efficiente, poiché i programmi interpretati erano molto lenti. Per questo, tutte le implementazioni recenti di macchine virtuali Java hanno incorporato un compilatore JIT (just in time), ossia un compilatore interno, che al momento del lancio traduce al volo il programma bytecode Java in un normale programma nel linguaggio macchina del computer ospite. Questi accorgimenti, a prezzo di una piccola attesa in fase di lancio del programma, permettono di avere delle applicazioni Java decisamente più veloci e leggere. Tuttavia programmi scritti in linguaggio Java restano tra i più lenti, in ogni caso, se in un determinato punto del programma c’è necessità di eseguire calcoli in maniera veloce, Java offre la “Java Native Interface” (JNI), un modo per chiamare, da codice Java, codice nativo scritto in altri linguaggi di programmazione come C o C++.

Rispetto alla tradizione dei linguaggi ad oggetti da cui deriva (e in particolare rispetto al suo diretto progenitore, il C++), Java ha introdotto una serie di notevoli novità rispetto all'estensione della sua semantica. Fra le più significative si possono citare probabilmente la possibilità di costruire GUI (o Graphical User Interface: GUI) con strumenti standard e non proprietari (per il C++ e altri linguaggi analoghi solitamente le GUI non fanno parte del linguaggio, ma sono delegate a librerie esterne), la possibilità di creare applicazioni multi-thread, ovvero che svolgono in modo concorrente molteplici attività, e il supporto per la riflessione, ovvero la capacità di un programma di agire sulla propria struttura e di utilizzare classi caricate dinamicamente dall'esterno.

Java è sia un linguaggio di programmazione, che una piattaforma, ossia un ambiente hardware/software dove un programma va in esecuzione. La piattaforma Java è una piattaforma solo software che gira su una piattaforma hardware di base che può essere un computer, una tv, un telefono cellulare, una smart card, o un altro dispositivo. La

piattaforma Java è composta da due blocchi: la Java Virtual Machine (JVM) e la Java Application Program Interface (API).
La JVM è la base della piattaforma Java, mentre la Java API è una collezione di componenti software pronti all'uso per lo svolgimento dei più disparati compiti. La macchina virtuale Java (JVM), è la macchina virtuale che esegue i programmi in linguaggio bytecode, ovvero i prodotti della compilazione dei sorgenti Java. La JVM è formalmente una specifica, e qualsiasi sistema che si comporti in modo coerente con tale specifica sarà quindi da considerarsi una particolare implementazione della JVM. Esistono implementazioni software, sia gratuite sia commerciali, della JVM per i moderni sistemi operativi più diffusi. È possibile eseguire un programma scritto in Java anche senza una JVM, questo richiede la compilazione del codice sorgente Java direttamente in un codice nativo anziché in bytecode, usando un apposito compilatore.

8.3.3 Unified Modeling Language: UML

In ingegneria del software, UML (Unified Modeling Language, "linguaggio di modellazione unificato") è un linguaggio di modellazione e specifica standard basato su concetti orientati agli oggetti. Il linguaggio nacque con l'intento di raccogliere una summa delle migliori pratiche nel campo della modellazione a oggetti che potesse facilmente diventare un punto di riferimento nel campo della programmazione orientata agli oggetti.
Il linguaggio UML consente di rappresentare in modello un sistema software orientato agli oggetti secondo numerosi aspetti (funzionali, strutturali, dinamici) e a diversi livelli di dettaglio, con una flessibilità sufficiente a garantire la realizzabilità di modelli accurati sia nella fase di analisi sia nelle varie fasi di progetto a diversi livelli di raffinamento, mantenendo la tracciabilità dei concetti impiegati per modellare il sistema in queste varie fasi. UML ha una semantica molto precisa e un gran potere descrittivo; caratteristiche che ne hanno decretato il successo anche al di fuori dell'ambito in cui esso era nato.
Lo standard UML definisce una sintassi e delle regole di interpretazione; non si tratta quindi di una metodologia di progettazione e per questo motivo può essere adottato con diverse metodologie o in ambiti diversi da quello informatico; attualmente non sono rari i

casi di sistemi software non orientati agli oggetti e modellati in UML, o addirittura modelli UML di sistemi o oggetti di altra natura come sistemi hardware, strutture organizzative aziendali, processi di business e così via.

Attualmente, l'uso di UML è estremamente diffuso. La maggior parte dei testi che trattano d'analisi e progettazione ad oggetti utilizzano UML per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico. Molti ambienti integrati di sviluppo per linguaggi ad oggetti come Java o C++ comprendono strumenti di modellazione in UML, eventualmente con meccanismi automatici di traduzione parziale dei diagrammi UML in codice. Sono anche disponibili ambienti software sofisticati dedicati alla modellazione in UML che consentono di generare codice in diversi linguaggi.

UML consente di descrivere un sistema secondo tre aspetti principali, per ciascuno dei quali si utilizzano un insieme di tipi di diagrammi specifici che possono poi essere messi in relazione fra loro:

- *Modello funzionale*: rappresenta il sistema dal punto di vista dell'utente, ovvero ne descrive il suo comportamento così come esso è percepito all'esterno, prescindendo dal suo funzionamento interno. Questo tipo di modellazione corrisponde, in ingegneria del software, all'analisi dei requisiti. La modellazione funzionale utilizza i diagrammi dei casi d'uso;
- *Modello ad oggetti*: rappresenta la struttura e sottostruttura del sistema utilizzando i concetti di classe e oggetto, evidenziando le relazioni esistenti tra loro. In ingegneria del software, questo tipo di modellazione può essere utilizzata sia nella fase di analisi del dominio che nelle varie fasi di progetto a diversi livelli di dettaglio. Il modello ad oggetti utilizza i diagrammi delle classi, i diagrammi degli oggetti, e i diagrammi di deployment;
- *Modello dinamico*: rappresenta il comportamento degli oggetti del sistema, ovvero la loro evoluzione nel tempo e le dinamiche delle loro interazioni. È strettamente legato al modello ad oggetti ed è impiegato negli stessi casi. Utilizza i diagrammi di sequenza, i diagrammi delle attività, e i diagrammi degli stati.

UML include tre meccanismi che consentono l'estensione della sua sintassi e della sua semantica da parte dell'utente: stereotipi, “tagged values”, e “constraints”. Questi strumenti possono essere usati nel contesto di un modello per esprimere concetti altrimenti non rappresentabili in UML, o non rappresentabili in modo chiaro e sufficientemente astratto. I profili UML sono collezioni di stereotipi, tagged values, e constraints, che specializzano il linguaggio per particolari domini applicativi o per l'uso di UML in congiunzione con particolari tecnologie. Fra i profili riconosciuti ufficialmente da OMG (l’associazione che gestisce lo standard UML) si trovano profili per i sistemi distribuiti, per sistemi con vincoli di QoS (qualità del servizio), e per sistemi real-time.
Alcuni dei diagrammi UML che descrivono l’interfaccia di comando implementata sono riportati in Figura 31, Figura 32, e Figura 33.

8.3.4 La libreria grafica Swing

L’obiettivo originale del progetto dell’interfaccia utente grafica in Java 1.0 era quello di consentire al programmatore di costruire una GUI che si presentasse bene su tutte le piattaforme. L’obiettivo fu raggiunto con L’Abstract Window Toolkit (AWT) che produceva una GUI con aspetto mediocre su tutti i sistemi. Purtroppo AWT era limitativo: erano disponibili soltanto quattro caratteri tipografici e non si poteva accedere ad alcuno degli elementi GUI più raffinati che pure esistono nel sistema operativo. Il modello di programmazione dell’AWT di Java 1.0 era per di più goffo e non orientato agli oggetti.
La situazione è migliorata col modello a eventi AWT di Java 1.1, che assume un approccio molto più nitido, orientato agli oggetti e aggiunge inoltre i JavaBeans, un modello di programmazione dei componenti che è orientato verso la facile creazione di ambienti ambienti di programmazione visivi. Java 2, usato per implementare l’interfaccia di comando per il nostro sistema, completa il distacco dal vecchio AWT sostituendo essenzialmente tutto con le Java Foundation Class (JFC), la cui componente grafica si chiama “Swing”. Si tratta di un ricco insieme di JavaBeans facili da utilizzare e da imparare, che possono essere utilizzati per creare una GUI con molta semplicità.

La libreria grafica predefinita di Java 2 è, quindi, Swing, la quale e molto vasta e per questo motivo tratteremo solo i suoi componenti utilizzati per lo sviluppo dell'interfaccia grafica di comando. Swing è un modello di programmazione decisamente migliore rispetto ai suoi concorrenti; inoltre questa libreria è stata fatta in modo da essere usata con molta facilità dai costruttori automatici di GUI, che ormai sono un aspetto obbligato di un ambiente di sviluppo Java che voglia definirsi completo. JavaBeans e Swing consentono a un costruttore GUI di scrivere il codice per conto del programmatore in modo automatico e nello stesso tempo il codice generato sarà molto leggibile e comprensibile. La libreria Swing contiene tutti i componenti che ci aspettiamo di trovare in un'interfaccia grafica moderna, dai pulsanti che contengono immagini, agli alberi, e alle tabelle. È una libreria di grandi dimensioni, ma è stata concepita per avere la complessità adeguata al compito da affrontare: se c'è da fare qualcosa di semplice, non si scrive molto codice, però se c'imbattiamo in progetti complessi, il codice diventerà proporzionalmente più complesso. Una caratteristica particolare di Swing è l'ortogonalità dell'utilizzo. Vale a dire, una volta capiti i concetti base della libreria, essi si applicano in modo uniforme a tutti gli oggetti. In questo modo i nomi delle classi e dei metodi sono simili e molto intuitivi, e si può programmare usando di meno il manuale.

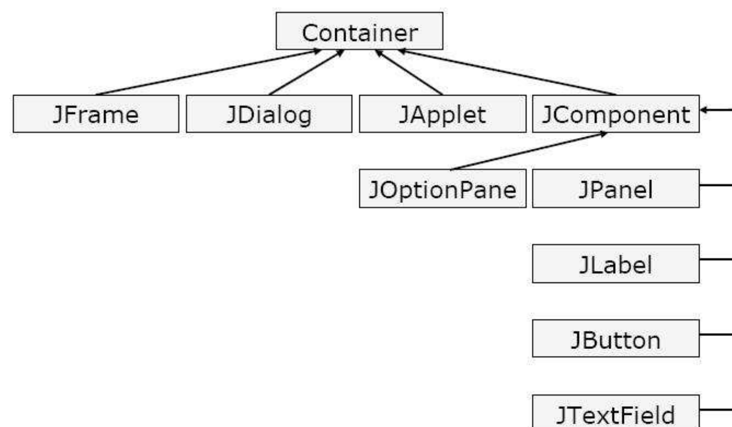

Figura 48: Panoramica della libreria Swing di Java 2

Nella Figura 48 è mostrata una panoramica degli elementi principali di Swing; di seguito descriveremo i più importanti:

- *JFrame* è una classe che implementa un contenitore, cioè una finestra dotata di cornice, barra del titolo con pulsanti tipici, e un pannello. Per inserire oggetti grafici al suo interno dobbiamo ottenere un riferimento al suo pannello mediante il metodo *getContentPane()* della classe *Container*.
- JApplet, usata per la programmazione web, è una classe che implementa una finestra integrata all'interno di un browser. Non esamineremo questo aspetto poiché l'argomento di questa tesi non tocca la programmazione web.
- JComponent, è la superclasse di tutti i componenti che possiamo inserire in un'applet o in un frame Java:
    - JPanel, implementa un contenitore senza cornice. Si usa per contenere al suo interno qualsiasi altro componente.
    - JLabel, implementa un stringa non selezionabile.
    - JButton implementa un pulsante con etichetta o immagine.
    - JTextField, implementa una casella di testo in cui è possibile far inserire del testo all'utente.
    - JComboBox, non presente nella figura, è un altro componente di Swing utilizzato per l'implementazione dell'interfaccia grafica di comando, che implementa un elenco a tendina dal quale l'utente può selezionare una voce.

Ad ogni azione che l'utente esegue su una GUI Java può essere associato un evento, ossia un codice che viene eseguito appena l'utente esegue la suddetta azione. Ciascun componente Swing può riferire tutti gli eventi che gli potrebbero accadere, e può riferire individualmente ciascun tipo di evento. Quindi, se non siamo interessati ad un determinato evento, ad esempio il fatto che il mouse passi sopra un pulsante, non registreremo il nostro interesse verso questo evento. È un modo molto diretto ed elegante per gestire la programmazione pilotata agli eventi. Per manifestare l'interesse verso un particolare

evento, bisogna dichiarare un *ActionListener* per questo evento, previsti dal pacchetto *java.awt.event*. Quest'ultimo particolare ci fa capire che Swing fa ancora uso del vecchio sistema grafico AWT, anche se con la sua introduzione molte classi di AWT sono diventate obsolete.

8.4 Interfaccia di comando

Per implementare l'interfaccia di comando è stato scelto, quindi, il linguaggio di programmazione orientato agli oggetti Java per due principali motivi:

- Estrema portabilità del software. Trattando un'interfaccia di comando su rete, è possibile impartire ordini da remoto, quindi da calcolatori potenzialmente con diversi sistemi operativi.
- Il linguaggio Java è particolarmente adatto per realizzare interfacce grafiche poiché fa uso della libreria standard "Swing" e non si appoggia su librerie esterne come, ad esempio, il linguaggio C++.

È stato inoltre preferito il linguaggio Java ad una piattaforma web basata su HTML abbinata ad un linguaggio di scripting, come ad esempio php, poiché in futuro potrà essere implementata nello stesso linguaggio una rappresentazione on-line grafica dei dati in ingresso e dei dati filtrati, molto semplice da realizzare in Java. Inoltre la piattaforma web implementa un modo di comunicare più complesso in quanto si serve di un calcolatore "server" per processare le richieste HTML.
Tutto il codice scritto è contenuto in tre pacchetti:

1. *CdafCtrl*: rappresenta il motore dell'interfaccia di comando. Contiene classi capaci di impartire ordini sia al demone in esecuzione sull'unità di calcolo, sia a quello in esecuzione sull'unita di controllo;

2. *InterfacciaGrafica*: contiene l'implementazione di un'interfaccia grafica (GUI), che permette di utilizzare tutte le funzionalità del pacchetto *CdafCtrl* in modo semplice ed intuitivo. La finestra principale dell'interfaccia grafica di comando è rappresentata in Figura 49;
3. *InterfacciaTestuale*: sistemi operativi rudimentali spesse volte non sono dotati di un motore grafico; altre volte invece sorge la necessità di controllare un sistema da remoto con strumenti come *telnet* o *ssh*. Per entrambi i casi il pacchetto *InterfacciaTestuale* implementa le stesse funzioni del pacchetto *InterfacciaGrafica*, ed è gestibile da un'interfaccia testuale.

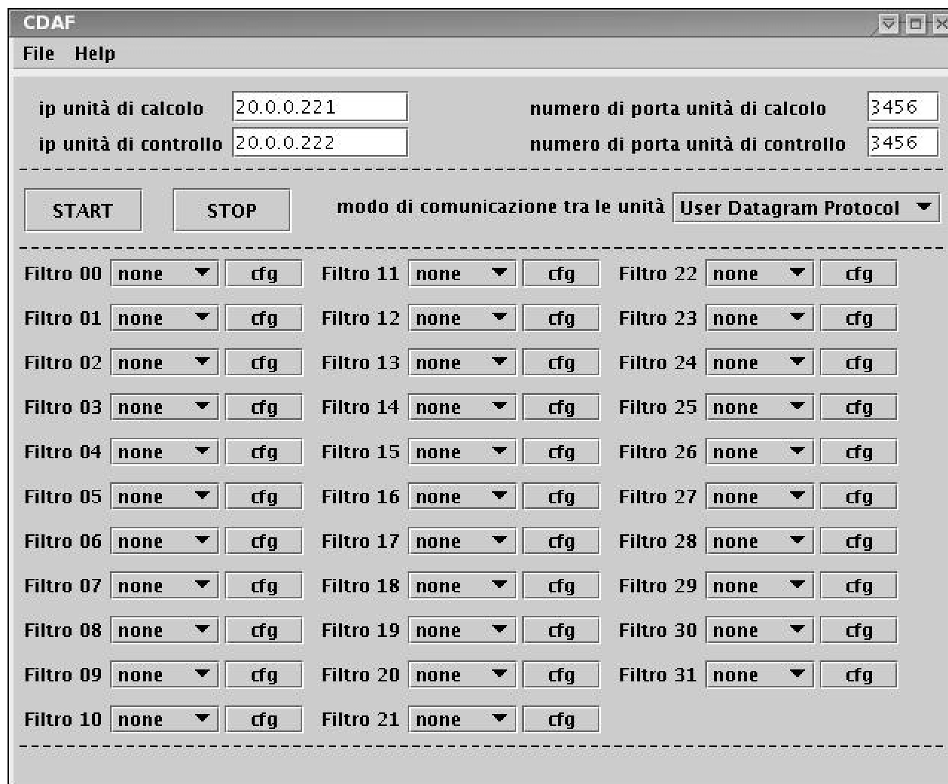

Figura 49: Finestra principale dell'interfaccia grafica sotto Linux

Il pacchetto più interessante è CdafCtrl, descritto ampiamente nel capitolo 5, il quale riesce a comunicare con processi scritti in linguaggio C mediante una connessione di rete.

8.4.1 Comunicazione mediante socket TCP

Essendo scritta l'interfaccia di comando in linguaggio Java, e i demoni in esecuzione sull'unità di calcolo e quella di controllo in linguaggio C, nasce il problema di implementare la comunicazione tra processi scritti in linguaggi di programmazione diversi, i quali offrono diverse strutture dati e diversi set di caratteri per rappresentare le stringhe.
La prima soluzione al problema prevedeva di implementare una libreria C capace di impartire ordini ai demoni, evitando così la comunicazione diretta come illustrato in Figura 50.

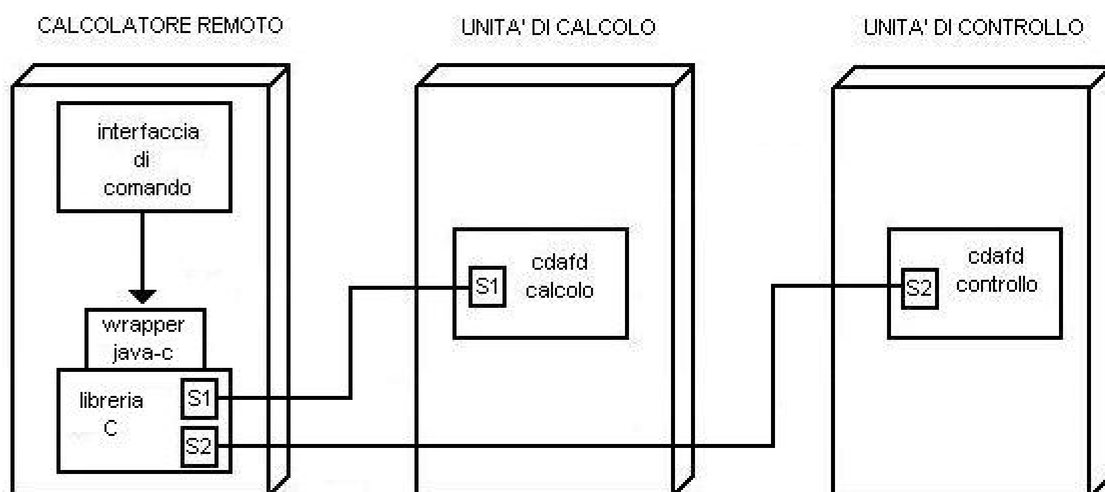

Figura 50: Soluzione che utilizza JNI

Per chiamare routine appartenenti alla libreria scritta nel linguaggio C da codice Java si usa la JNI (Java Native Interface) introdotta precedentemente in questo capitolo. Tuttavia questa soluzione ha forti ripercussioni sulla portabilità dell'interfaccia di comando, quindi è stata scartata.
L'altra soluzione è quella di comunicare con i programmi demoni direttamente dal processo scritto in Java (Figura 47). Ognuno dei due demoni è sempre in esecuzione attendendo una richiesta da parte di un client, contenuta in una struttura; ad esempio il demone in esecuzione sull'unità di controllo accetta una struttura di questo tipo:

```
typedef struct opt_controllo {
      char command;
      char ip[LEN_IP];
      char portnum[LEN_PORTNUM];
      char protocollo[LEN_PROTOCOLLO];
} opt_controllo_t;
```

Un struttura simile è accettata dal demone in esecuzione sull'unità di calcolo. Il linguaggio di programmazione Java non prevede le strutture, ma è possibile simulare la struttura appena riportata con una classe contenente quattro stringhe di lunghezza fissata. Una differenza tra i due linguaggi è che Java rappresenta le stringhe in formato unicode a 16 bit, mentre in C sono rappresentate con il set di caratteri UTF ad 8 bit. Per superare questa differenza nel pacchetto CdafCtrl è stata inserita la classe StringConverter non visibile dall'esterno, che ha appunto il compito di convertire le stringhe da unicode a UTF.
La classe appena citata offre il seguente metodo per convertire le stringhe:

```
public String encode(String in_str);
```

Chiamando questo metodo, la stringa *in_str* data in ingresso è convertita in un differente formato ed è restituita in output. Nel linguaggio C le stringhe hanno una dimensione fissata, quindi necessitiamo anche di fissare il numero di caratteri appartenenti alla stringa. La classe *StringConverter* offre anche il seguente metodo:

```
public String encode(String in_str, int maxDim);
```

Questo metodo è simile al precedente, solo che fa decidere all'utente della classe la dimensione della stringa da restituire in output.
Notiamo che la classe *StringConverter* offre due funzioni differenti che hanno lo stesso nome: questa particolarità, prevista anche dal linguaggio Java, è chiamata *overloading*.
In programmazione, parliamo di *overloading*, quando ci troviamo di fronte ad una famiglia di funzioni aventi lo stesso nome, ma con la possibilità di accettare un diverso insieme di argomenti, ed eventualmente restituire un diverso valore di ritorno.

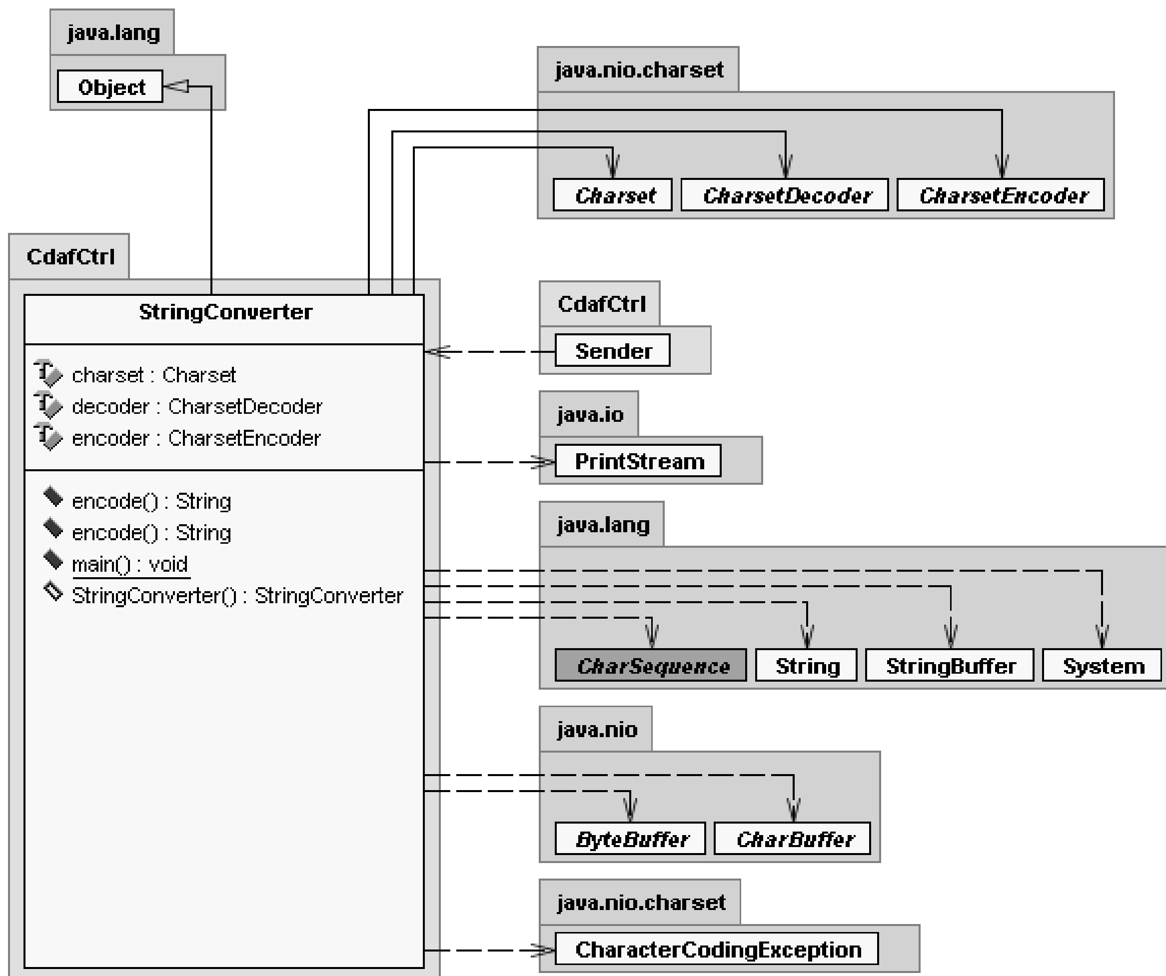

Figura 51: Diagramma della classe StringConverter

Come si nota dal diagramma della classe StringConverter (Figura 51), per convertire una stringa da un set di carattere ad un altro, sono usate le classi Charset, CharsetDecoder, e CharsetEncoder di java.nio. La classe StringConverter non è stata pensata per convertire le stringhe solo nel formato UTF a 8 bit, ma può usare qualsiasi set di caratteri supportato dal linguaggio Java. Ogni volta che si crea un'istanza di questa classe si deve, quindi, decidere quale set di carattere adottare, passando un opportuno parametro al costruttore:

```
StringConverter converter = new StringConverter("UTF-8");
```

Tornando alla comunicazione tra l'interfaccia di comando e il demone sull'unità di controllo, una volta convertite le stringhe non ci resta altro che spedirle mediante socket nell'ordine in cui sono state definite nella struttura riportata in precedenza:

```
PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream(), "UTF-8")), true);

out.print(command);
out.print(ipCalcolo);
out.print(portnumCalcolo);
out.println(protocollo);
```

In modo simile è implementata la comunicazione con l'unità di calcolo.

8.4.2 Pacchetto CdafCtrl

Il pacchetto CdafCtrl rappresenta il cuore dell'interfaccia di comando; esso, oltre a contenere la classe StringConverter appena descritta, contiene le classi visibili dall'esterno OpzioniCdaf e ControlCdaf che fanno da interfaccia al pacchetto stesso. In effetti l'utente di questo pacchetto deve seguire i seguenti passi:

1. Creare un'istanza della classe OpzioniCdaf;
2. Impostare tutti i parametri dell'istanza creata;
3. Creare un'istanza della classe ControlCdaf collegandola alle opzioni precedentemente definite;
4. Impartire comandi al sistema mediante i metodi *start()* e *stop()* offerti dall'oggetto appena creato.

In codice questo si traduce grosso modo in:

```
OpzioniCdaf opt = new OpzioniCdaf();
ControlCdaf ctrl = new ControlCdaf(opt);
ctrl.start();
```

La classe ControlCdaf, per riuscire ad inviare dati ai demoni in esecuzione sulle due unità, si serve a sua volta della classe Sender la quale offre i metodi sendStart() e sendStop() per impartire rispettivamente ordini di avvio o di arresto del processo di acquisizione, elaborazione, ed attuazione (Figura 52).

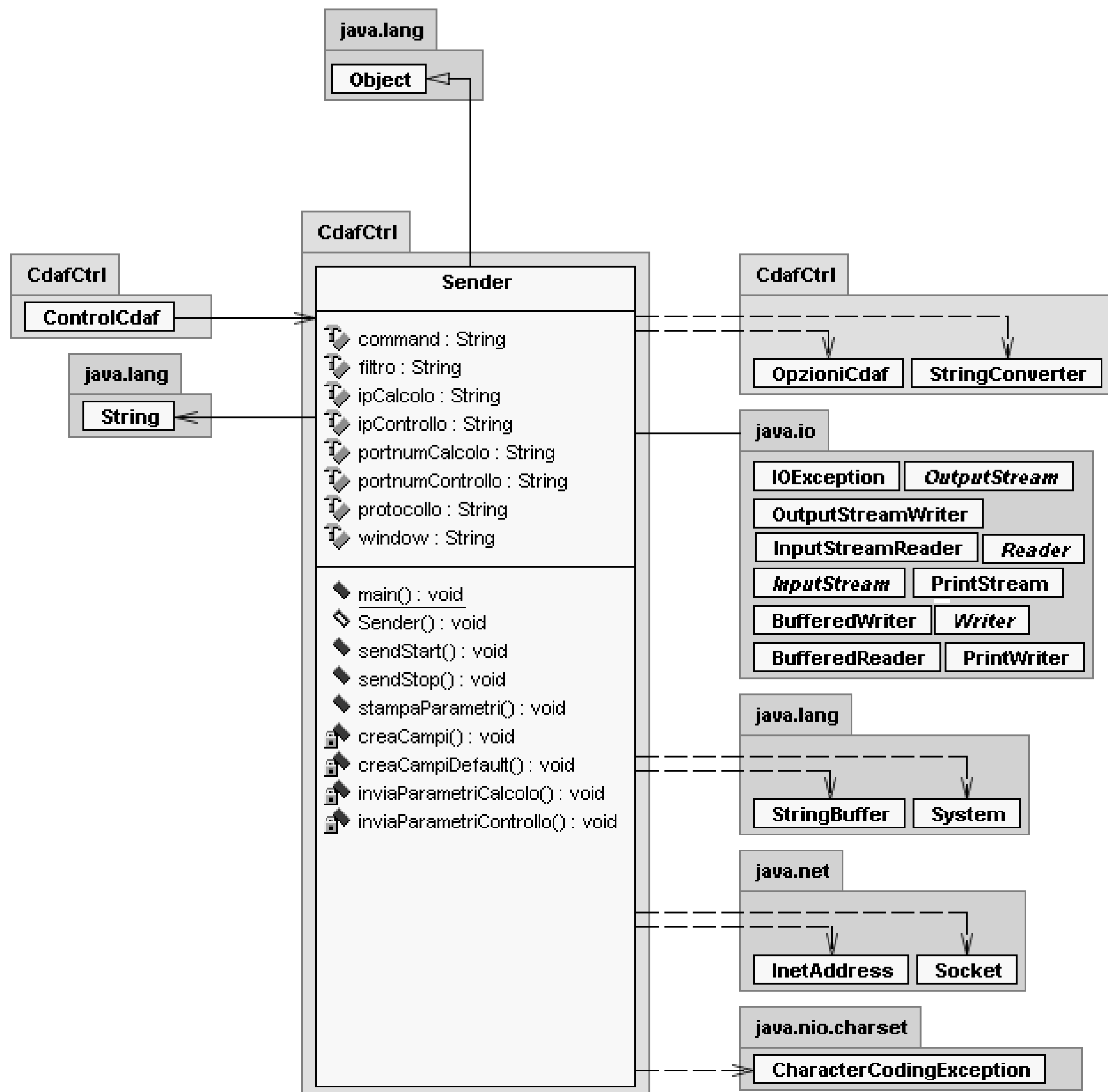

Figura 52: Diagramma della classe Sender

Non scenderemo in dettaglio nell'analisi di questo pacchetto poiché già descritto ampiamente nel capitolo 5.

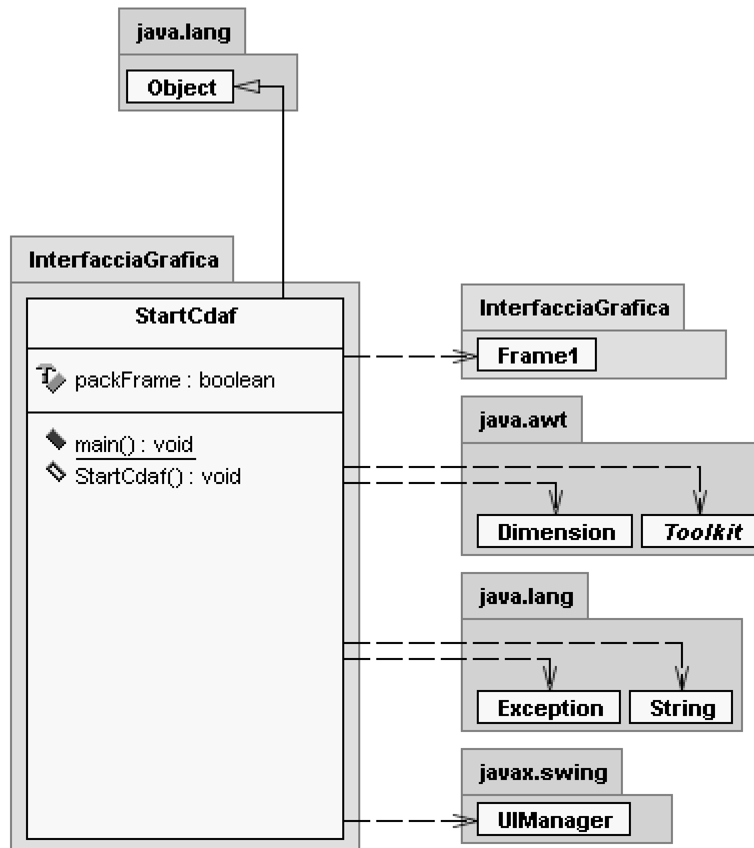

Figura 53: Diagramma della classe StartCdaf

8.4.3 Pacchetto InterfacciaGrafica

L'architettura del pacchetto *InterfacciaGrafica* è descritta nel capitolo 5, in questa sezione sarà presentata ogni sua classe guardando l'aspetto implementativo. Le classi contenute nel pacchetto sono *StartCdaf*, *Frame1*, *FilterBox*, *AboutBox*, e tutte le classi che implementano gli ActionListener per gli oggetti dell'interfaccia.

StartCdaf: La classe *StartCdaf* è la classe principale del pacchetto e, come già detto nel capitolo 5, è la classe che contiene il metodo *main()* responsabile di far partire l'interazione con l'utente. Il diagramma UML di questa classe è riportato in Figura 53.
Il metodo *main()* non fa altro che creare un'istanza della classe *StartCdaf*, e di conseguenza viene chiamato il costruttore *StardCdaf()*, il quale provvede a creare un oggetto della classe *Frame1* e a farlo visualizzare a video.

Frame1: La classe Frame1 crea un contenitore di tipo JFrame e all'interno del suo pannello inserisce componenti di tipo JLabel, JButton, JComboBox, e JTextField. Il risultato si può vedere in Figura 49 e Figura 56.
Gli oggetti di tipo JLabel sono tutte le stringhe non modificabili riportate direttamente sul corpo della finestra. La loro funzione è quella di rendere l'interfaccia grafica più intuitiva e semplice da utilizzare.
Gli oggetti di tipo *JTextField* sono quelli usati per permettere l'inserimento degli ip e dei numeri di porta da parte dell'utente.
L'interfaccia grafica prevede l'abilitazione e la configurazione di un filtro per ogni canale. Questa caratteristica è riportata nella parte bassa della finestra, e per ognuno dei 32 canali (dal canale 00 al canale 31) si è usato un oggetto di tipo *JComboBox* e un oggetto di tipo *JButton*. *JComboBox* è un elenco a discesa, che permette di abilitare o disabilitare (se impostato su "none") un filtro. Alla destra d'ogni elenco a discesa è presente un pulsante con etichetta "cfg" realizzato usando la classe JButton. Premendo uno di questi pulsanti, verrà visualizzato un frame, implementato nella classe FilterBox, concernente la configurazione del relativo filtro.
Un ulteriore elenco a discesa è usato per decidere il modo di comunicazione tra e unità, offrendo la scelta tra "User Datagram Protocol" (UDP), e "Reflective Memory" (previste negli sviluppi futuri.
Infine ci sono i pulsanti con etichette "START" e "STOP" usati rispettivamente per iniziare e terminare il processo di acquisizione, elaborazione, ed attuazione dei dati.

FilterBox: Il frame implementato nella classe FilterBox aiuta l'utente a configurare il filtro scelto. Ogni volta che si crea un frame di questo tipo, passiamo dei parametri al costruttore indicando il numero del canale e il tipo di filtro scelto. In questo modo la finestra generata dalla classe FilterBox si adeguerà automaticamente alla scelta dell'utente.
Attualmente l'unico filtro implementato all'interno dell'unità di calcolo è il filtro FIR Moving Average che come parametro di configurazione chiede solo la dimensione del finestra (window) sulla quale eseguire i calcoli. In futuro sono previsti filtri più complessi che richiedono in input vari parametri.

Altre Classi: Le altre classi del pacchetto *InterfacciaGrafica* sono *AboutBox* e tutte le classi che implementano gli *ActionListener* necessari ad associare il codice opportuno ad ogni evento significativo della nostra interfaccia. La classe *AboutBox* non ha uno scopo importante: implementa solamente un frame che da all'utente informazioni sugli autori e sulla versione del programma. I vari *ActionListener* hanno invece lo scopo di interagire con il pacchetto *CdafCtrl* per impartire effettivamente ordini al sistema.

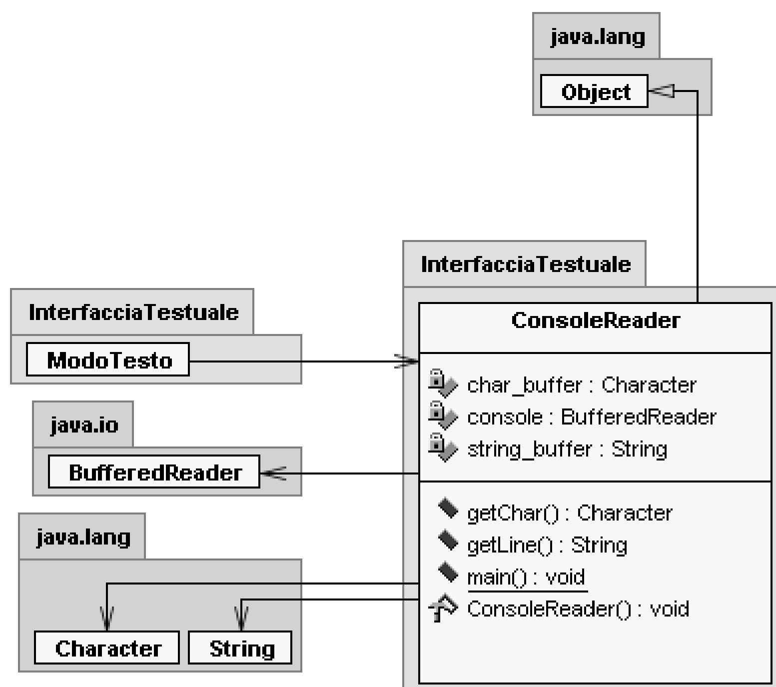

Figura 54: Diagramma della classe ConsoleReader

8.4.4 Pacchetto InterfacciaTestuale

Il pacchetto InterfacciaTestuale implementa un'interfaccia testuale a riga di comando comoda da utilizzare quando non si dispone di un ambiente grafico. Le due classi contenute in questo pacchetto sono ModoTesto e ConsoleReader, la cui analisi architetturale è presente nel capitolo 5. In questa sezione vedremo alcuni dettagli implementativi di queste due classi.

ConsoleReader

Java è un linguaggio pensato per le interfacce grafiche, pertanto non offre agli sviluppatori strumenti comodi per leggere input da linea di comando. Per ottenere ciò con semplicità si è creata la classe ConsoleReader, che mediante i metodi getChar() e getLine() permette di leggere rispettivamente un carattere o una stringa da tastiera. Per funzionare correttamente questa classe fa uso di *InputStream*, *InputStreamReader*, *PrintStream*, e *Reader* contenute in *java.io*. Il diagramma della classe ConsoleReader è illustrato in Figura 54.

```
[]=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=[]

1) Start
2) Stop
3) Opzioni
4) ESCI

[]=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=[]

Inserisci la tua scelta:
```

Figura 55: Menu principale in modalità testuale

ModoTesto

La classe *ModoTesto* è quella che realmente implementa l'interfaccia a linea di comando. L'unico metodo appartenente a questa classe visibile dall'esterno è *main()*, che ha il compito di accettare l'input dall'utente da tastiera usando la classe *ConsoleReader*

precedentemente descritta, e impartire ordini al sistema interagendo con le classi del pacchetto *CdafCtrl*.

Al suo interno la classe ModoTesto implementa dei menu testuali, come quello in Figura 55, che permettono all'utente di controllare il sistema in modo semplice.

8.5 Test dell'interfaccia di comando

L'interfaccia di comando, come tutti gli altri elementi del sistema realizzato, è stata ampiamente testata. I due demoni in esecuzione, uno sull'unità di calcolo e l'altro sull'unità di controllo, riescono a percepire gli ordini impartiti dall'applicazione scritta in Java, e continuano a funzionare anche in caso di sovraccarico di richieste.

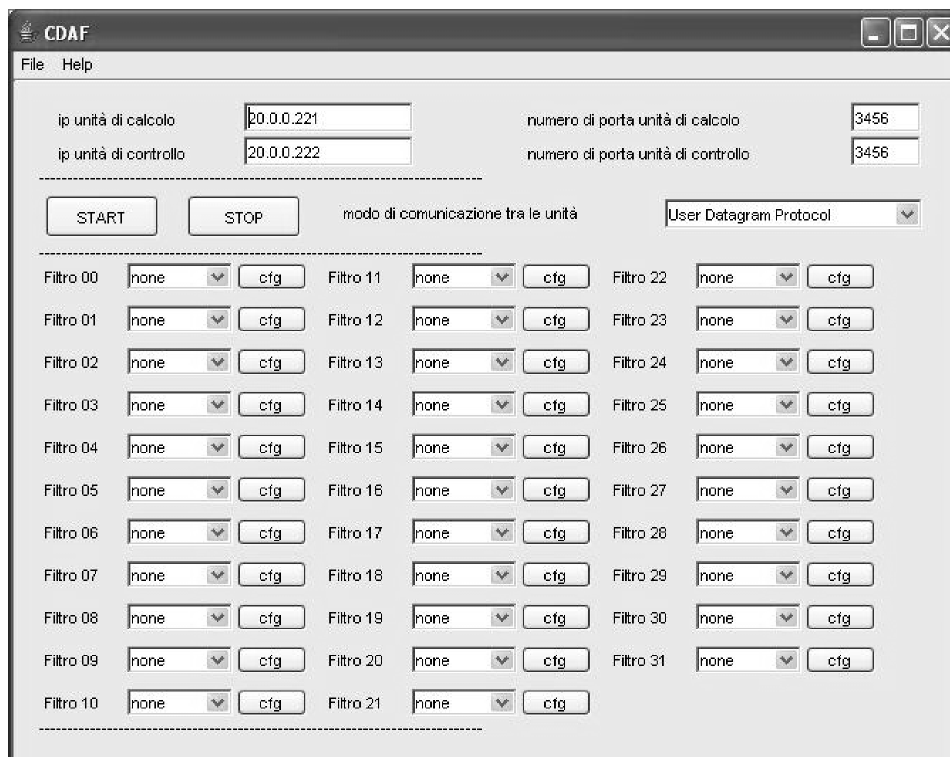

Figura 56: Finestra principale dell'interfaccia grafica sotto Windows

Inoltre, dai test effettuati, la permanenza in memoria dei due demoni e l'operato dell'interfaccia di comando non fa diminuire la velocità di comunicazione tra le due unità del sistema.

Essendo l'interfaccia di comando (grafica e testuale) scritta interamente in Java, può essere eseguita da qualsiasi calcolatore con un sistema operativo compatibile con lo stack di protocolli TCP/IP e dotato di una JVM (Java Virtual Machine).

Sono stati eseguiti test sul sistema operativo Linux (Figura 49) e Windows (Figura 56) e non sono state riscontrati problemi di esecuzione se non piccole differenze grafiche.

Capitolo 9

Conclusioni

Lo compito del sistema realizzato è quello di acquisire segnali elettrici analogici da sensori, inviarli ad una o più CPU per l'elaborazione, e riconsegnarli ad una o più unità di attuazione per la riconversione in segnali elettrici analogici. L'intero sistema è stato pensato per realizzare un sistema elettromeccanico per il controllo automatico degli specchi di un rilevatore interferometrico.

Per il lavoro sono stati impiegati strumenti, come ad esempio il VMEBus, molto usati da aziende produttrici di sistemi a tempo reale stretto; quindi il sistema realizzato può essere impiegato, più genericamente, nel mondo industriale, ovunque ci sia la necessità di applicare un filtro digitale in modo veloce e flessibile ad un segnale elettrico analogico.

L'obiettivo principale è stato, per tutta la durata del lavoro, quello di creare un sistema in grado di acquisire, filtrare ed attuare i dati con frequenze elevate. Per raggiungerlo è stato progettato un device driver per il dispositivo di conversione digitale analogico MPV955 in modo scrupoloso, riducendo al minimo le operazioni fatte durante la scrittura dei dati sul dispositivo. Inoltre è stato implementato un sistema di comunicazione tra l'unità di calcolo e quella di controllo capace di ridurre al minimo i tempi di trasmissione dei dati sul canale di comunicazione, dando poco spazio agli overhead introdotti dal sistema operativo, come il tempo di cambio di contesto.

I risultati dei test effettuati indicano che, nonostante gli sforzi fatti per avere un sistema veloce, la frequenza più alta raggiunta non può comunque considerarsi elevata. Le limitazioni sono dovute essenzialmente a tre motivi:

1. Il bus VME Standard a 32bit ha prestazioni ridotte, e molto inferiori a suoi successori;
2. Il driver per il dispositivo analogico digitale non riesce a gestire frequenze elevate, quindi richiede di una revisione;
3. Il canale di comunicazione adottato per collegare le unità è eccessivamente lento.

Un rilevante incremento delle prestazioni, quindi, richiede di una strumentazione più veloce e moderna.
L'altro obiettivo perseguito è l'affidabilità del contenuto informativo dei dati acquisiti. Vista la struttura semplice del sistema il numero d'errori commessi durante la trasmissione è praticamente nullo. Per quanto riguarda l'errore introdotto dalle conversioni analogico-digitale e digitale-analogico dai dispositivi ADC e DAC, nel nostro contesto è trascurabile. Tuttavia se in altri contesti è necessaria una maggiore accuratezza dei dati bisognerà sostituire i suddetti dispositivi con altri modelli a risoluzione più elevata: ciò comporterebbe la riprogettazione dei driver e una modifica alle strutture dati usate per il trasporto dei dati. A tal proposito si ricordi che i device driver sono la parte del sistema operativo più vicina al dispositivo e, quindi, i driver progettati sono compatibili con il sistema operativo LynxOS e con i dispositivi VGD5 Analog-Digital Converter ed MPV955 Digital-Analogic Converter della Pentland System.
Per quanto concerne la parte di filtraggio, momentaneamente, è stato implementato solo un filtro molto semplice, con una scarsa capacità nell'attenuare i rumori indotti dal processo di conversione e dall'ambiente esterno. In futuro è previsto l'aggiunta di nuovi filtri digitali poiché la struttura del programma permette in maniera semplice, l'aggiunta del codice.
Per rendere più semplice l'utilizzo del sistema, è stata implementata un'interfaccia di comando capace di impartire ordini ad entrambe le unità da remoto, dando la possibilità all'utente di eseguire tutte le operazioni di gestione concernenti l'acquisizione al filtraggio ed all'attuazione.
Risultati incoraggianti sono stati raggiunti sui tempi d'andata e ritorno dei dati dall'unità di controllo a quella di calcolo che, nonostante la nostra strumentazione con prestazioni ridotte, sono stati quasi dimezzati rispetto a quelli previsti dalla specifica dei requisiti del

sistema. Purtroppo la comunicazione tra le unità avviene mediante un comune cavo di rete di categoria 5e adottando il protocollo UDP. La buona progettazione del software permette l'aggiunta d'eventuali altri modi di comunicazione, semplicemente aggiungendo delle funzioni al modulo relativo. In futuro a tal proposito è previsto l'utilizzo di schede reflective memory che, per comunicare tra loro, sfruttano la velocità delle fibre ottiche: in questo modo le prestazioni dell'intero sistema saranno migliorate significativamente.

Bibliografia

[1] A.S. Tanebaum. *Modern Operating Systems*. Prentice-Hall, 2001.

[2] W. Stallings. *Operating Systems*. Prentice-Hall, fourth edition, 2000.

[3] LynuxWorks Inc. *LynxOS Installation Guide*, 2002.

[4] LynuxWorks Inc. *LynxOS User's Guide*, 2002.

[5] LynuxWorks Inc. *Writing Device Drivers for LynxOS*, 2002.

[6] John Rynearson. *VMEBus FAQ's. VITA Journal*, 1997.

[7] Pentland System Ltd. *VGD5*, 1998.

[8] Pentland System Ltd. *VGX Mother Board*, 1998.

[9] Pentland System Ltd. *MPV955 Analog Output Board*, 1998.

[10] R. Pressman. *Principi di ingegneria del software*. McGraw-Hill, 2000.

[11] J. Ganssle. *The Art of Designing Embedded Systems*. Newnes, 1998.

[12] A. Rubini. *Linux Device Driver*. Second edition, 2001.

[13] W.R. Stevens. *UNIX Networking Programming*. Prentice-Hall, second edition, 1998.

[14] J. P. Farrel B. Nicholos, D. Buttlar. *Pthreads Programming*. O'Reilly, second edition, 1998.

[15] Meilir Page-Jones. *Progettazione a oggetti con UML*. Apogeo, 2001.

[16] Bruce Eckel. *Thinking in Java*. Apogeo, third edition, 2002.