UNIVERSITÀ
DEGLI STUDI
DI PADOVA

UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

SVILUPPO ED UTILIZZO DI GNU/LINUX NEI SISTEMI EMBEDDED

Relatore:
Dott. TOMASO ERSEGHE

Tesi di Laurea di:
ALBERTO GAMBARUCCI

— ANNO ACCADEMICO 2010/2011 —

Indice

Capitolo 1

Storia e generalità dei sistemi Embedded

1.1 Introduzione

In data 10/07/2010 ho iniziato l'attività di tirocinio presso l'azienda Nethix S.r.l. di Castelfranco Veneto. L'argomento trattato durante il progetto è la realizzazione di un sistema Embedded Linux basato su microcontrollore ARM. L'azienda Nethix S.r.l. si occupa principalmente di telecontrollo a distanza utilizzando in particolare tecnologia Wireless e GPRS nei propri prodotti. Le sempre maggiori esigenze di gestione di sistemi distribuiti e la volontà di realizzare un sistema centralizzato di controllo flessibile, hanno portato alla scelta di progettare un sistema Embedded basato su Linux. L'argomento principale sarà dunque Linux su sistemi Embedded, tuttavia a causa della vastità dell'argomento per mantenere la necessaria chiarezza di esposizione, esporrò anche alcuni argomenti riguardanti l'implementazione hardware del dispositivo Embedded in esame. La tesi verrà strutturata precisamente attraverso la seguente sequenza di capitoli:

- Nel primo capitolo verrà esposta una breve trattazione storica dei sistemi Embedded, dei sistemi Real Time Linux e non e del loro sviluppo nel corso degli anni.
- Il secondo capitolo introdurrà il sistema Embedded Linux che è stato oggetto del tirocinio e le sue caratteristiche hardware.
- Il capitolo tre tratterà la tecnica di inizializzazione software a basso livello del dispositivo
- Il quarto capitolo sarà dedicato all'architettura interna di un sistema basato su Linux e presenterà alcuni esempi di applicazioni pratiche.
- Il capitolo cinque tratterà alcuni approfondimenti teorici sulla gestione del supporto fisico di memoria utilizzato nel sistema Embedded Linux.

Lo scopo della tesi non sarà quello di creare un manuale pratico per la realizzazione di un sistema Embedded Linux, ma di dare una panoramica sulle

procedure implementative di tali sistemi, approfondendo inoltre alcuni argomenti critici sotto il profilo hardware e software incontrati durante l'esperienza in azienda. Solo in alcuni casi verranno esposi aspetti procedurali specifici nel porting Linux su una piattaforma Atmel custom, privilegiando quanto più possibile un approccio generale. Inoltre verranno introdotti diversi concetti dell' architettura del kernel Linux per analizzarne potenzialità e carenze in un contesto Embedded.

La parola Embedded di matrice inglese, letteralmente sistema incorporato, ovviamente non offre molte informazioni, cerchiamo dunque di spiegare cosa sono i sistemi Embedded, cosa offrono e quale sia il loro impiego.

Sono sistemi basati su microprocessore/microcontrollore progettati in modo specifico per una determinata applicazione si parla dunque di dispositivi *"special puropose"*. *Qu*esta specializzazione come vedremo è modulabile secondo il tipo di applicazione. Quando si fa riferimento a un sistema Embedded è giusto sottolineare l'ampia fascia di dispositivi che appartengono a tale categoria, si va da semplici microcontrollori con istruzioni a 8bit a veri e propri microprocessori dotati di MMU e svariati controllers a 32 bit, già da questa breve introduzione al mondo Embedded è facile capire come l'impiego di tali sistemi sia largamente diffuso.

Al giorno d'oggi i seguenti dispositivi possono essere considerati sistemi Embedded:

- Sportelli Bancomat
- Elettronica aeronautica quali ad esempio sistemi di guida inerziale, hardware/software di controllo per il volo e altri sistemi integrati nei velivoli e nei missili.
- Telefoni cellulari dai più semplici agli smartphone e i PDA veri e propri mini computer.
- Centralini telefonici.
- Apparecchiature per reti informatiche come router, timeserver e firewall.
- Stampanti e Fotocopiatrici.
- Sistemi di automazione casalinghi come termostati, condizionatori e altri sistemi di monitoraggio della sicurezza.
- Distributori di bevande.
- Elettrodomestici come forni a microonde, lavatrici, apparecchi televisivi, lettori o masterizzatori di DVD.
- Apparecchiature biomedicali come ecografi, scanner medici per risonanza magnetica e sistemi per controllo laser.
- Strumenti di misura come oscilloscopi digitali, analizzatore logico, e analizzatore di spettro.
- Console per videogiochi fisse e portatili.

- Centraline di controllo dei motori automobilistici e degli ABS.
- Strumenti musicali digitali quali tastiere workstation, mixer digitali o processori audio.
- Centraline di controllo nelle automobili e autocarri (motore, illuminazione, assistenza alla guida)
- Decoder per TV digitale.

La panoramica di dispositivi appena visti è solo un esempio di applicazioni, in realtà ogni sistema elettronico di elaborazione può essere considerato un sistema Embedded.

Soffermiamoci adesso su alcune caratteristiche distintive dei sistemi Embedded per poi passare in rassegna gli argomenti trattati in questo documento.

Un sistema Embedded ha diversi vantaggi rispetto ad un sistema *"general purpose"* è solitamente un sistema dall'ingombro contenuto, ha un consumo energetico basso essendo un sistema dedicato riesce ad offrire funzionalità Real-Time scalabili secondo le esigenze.

1.2 Evoluzione storica dei dispositivi

Storicamente come primo sistema Embedded si considera l'Apollo Guidance Computer (AGC) costruito da Charles Stark Draper agli inizi degli anni sessanta nel MIT Instrumentation Laboratory. Il suo compito fu quello di controllare il Command Module e il Lunar Module durante le missioni Apollo. L'AGC fu uno dei primi sistemi, o almeno uno dei più noti, che possono essere considerati IC (Integrated Circuit). Nonostante i sui 32Kg di peso e le dimensioni non certo compatte (61x32x17 cm) l'AGC fu uno dei componenti più innovativi e più a rischio nelle missioni Apollo. Nel 1961 viene progettato l'Autonetics D-17 che fu il primo sistema Embedded prodotto in massa progettato per la guida del missile Minuteman, tale sistema utilizzava una tecnologia a transistor e un hard disk come memoria primaria. Intorno alla metà degli anni sessanta lo sviluppo militare portò una spinta tecnologica notevole che fece diminuire il costo dei circuiti integrati (NAND-gate) aprendo le porte per una diffusione su larga scala dei primi microprocessori. Il primo microprocessore commercializzato fu l'Intel 4004 nel 1972, esso venne impiegato su calcolatrici e dispositivi di piccole dimensioni, la frequenza della CPU era di circa 740KHz. L'innovazione cruciale in questo dispositivo fu l'introduzione della tecnologia MOS Silicon Gate ("self-aligned gate") a cui contribuì in modo particolare Federico Faggin fisico vicentino.

Federico Faggin si laurea in fisica summa cum laude nel 1965 all'Università di Padova. Viene quindi assunto dalla SGS (oggi parte di STMicroelectronics) in seguito si trasferisce nesti Stati Uniti dove inizia a dedicarsi alla neonata tecnologia MOS (metallo-ossido-semiconduttore), per la quale inventa innovazioni essenziali (tra queste, lo sviluppo della tecnica della porta al silicio (silicon gate), usando come conduttore il silicio policristallino drogato anziché l'alluminio).

La tecnologia con gate in polisilicio consentì di migliorare le prestazioni dei circuiti integrati garantendo, oltre a una maggiore precisione nel posizionare le regioni di drain e source, anche una diminuzione drastica degli effetti parassiti quali correnti di leakage e capacità parassite del transistor.

Negli anni settanta si ebbe una grande diminuzione dei costi produttivi, inoltre superate le difficoltà sulla stabilità dei gate dei MOSFET (inizialemente conosciuti come IGFET) questi nuovi dispositivi di fatto si imposero sulla tradizionale tecnologia bipolare, la loro caratteristica principale era il basso consumo per porta. L' evoluzione delle tecniche di processo portarono dunque alla realizzazione di circuiti integrati prima interamente in logica PMOS poi in logica NMOS.

Dal 1970 al 1974 sempre Federico Faggin è responsabile della ricerca e lo sviluppo di Intel e lavora al progetto dell'8008, il primo a 8 bit e del successivo 8080 progenitori della famiglia di processori 8086 che domina ancora oggi il mercato.

Successivamente Faggin abbandonerà intel per fondare la ZiLOG, la società che costruisce lo Z80, il microprocessore che fra la versione iniziale e i successivi miglioramenti e cloni è stato prodotto per quasi vent'anni, si calcola in oltre un miliardo di pezzi.

I successivi anni mostrarono la straordinaria validità della legge di Moore, le prestazioni dei processori, e il numero di transistor ad esso relativo, raddoppiarono ogni 18 mesi. Gli anni ottanta segnarono l'avvento dei primi microcontrol-

lori ovvero sistemi sempre più indipendenti da device esterni. Successivamente i sistemi Embedded andarono coprendo una sempre più grande fascia di mercato, tutti i compiti, nei quali non era possibile immaginare l'utilizzo dei tradizionali computers general purpose, furono svolti da sistemi Embedded. I principali vantaggi nell'utilizzo di sistemi Embedded su certe applicazioni sono oltre al costo e dimensioni contenute la possibilità di avere un controllo Real-Time sul proprio dispositivo cosa che solitamente un sistema complesso non permette in modo agevole.

1.3 Embedded Linux

Per sistema Embedded Linux si intende un dispositivo basato su un microcontrollore di fascia alta dotato solitamente di MMU (anche se non strettamente necessaria grazie a progetti come uClinux) capace di eseguire un sistema operativo basato sul kernel Linux. La famiglia di processori a 32 bit , più diffusa attualmente per tale scopo è l'ARM.

E' importante notare la grande quantità di dispositivi, e quindi di partner commerciali, che utilizzano microprocessori ARM:

Tabella 1.1: Tabella di alcuni dei più comuni dispositivi basati su ARM

famiglia	architettura	core	dispositivi
ARM7TDMI	v4T	ARM7TDMI(-S)	Game Boy Advance, Nintendo DS, iPod
ARM7TDMI	v4T	ARM710T	Psion 5 series, eMate 300
ARM7TDMI	v4T	ARM720T	
ARM7TDMI	v4T	ARM740T	
ARM7TDMI	v5TEJ	ARM7EJ-S	
ARM9TDMI	v4T	ARM9TDMI	
ARM9TDMI	v4T	ARM920T	ARMadillo, Neo FreeRunner, GP32, GP2X (first core), Tapwave Zodiac (Motorola i. MX1)
ARM9TDMI	v4T	ARM922T	
ARM9TDMI	v4T	ARM940T	GP2X (second core)
ARM9E	v5TE	ARM946E-S	Nintendo DS, Nintendo DSi, Nokia N-Gage, Conexant 802.11 chips, Samsung YP-K5 MP3 player

ARM9E	v5TE	ARM966E-S	ST Micro STR91xF, incluso Ethernet
ARM9E	v5TE	ARM968E-S	
ARM9E	v5TEJ	ARM926EJ-S	Texas Instruments OMAP 16xx, 17xx; Nokia 6630, 6680, N70; Sony Ericsson (K, W series), Siemens and Benq (x65 series and newer)
ARM9E	v5TE	ARM996HS	
ARM10E	v5TE	ARM1020E	
ARM10E	v5TE	ARM1022E	
ARM10E	v5TEJ	ARM1026EJ-S	
ARM11	v6	ARM1136J(F)-S	Texas Instruments OMAP2, Nokia E51 (369 MHz), Nokia 6700 Classic, Nokia 6120 Classic, Nokia 6220 Classic (369 MHz), Nokia 6720 Classic, Nokia 6290, Nokia 6210 Navigator, Nokia 6710 Navigator, Nokia N93, Nokia N95 (333 MHz), Nokia 5800 Xpressmusic (434 MHz), Nokia N97 (434 MHz) HTC Dream, HTC Magic, HTC Hero
ARM11	v6T2	ARM1156T2(F)-S	
ARM11	v6KZ	ARM1176JZ(F)-S	iPhone
ARM11	v6K	ARM11 MPCore	
Cortex	v7-A	Cortex-A8	Texas Instruments OMAP3, Freescale famiglia i.MX51
Cortex	v7-R	Cortex-R4	Broadcom
Cortex	v6-M	Cortex-M0	NXP famiglia LPC11xx

Cortex	v7-M	Cortex-M3	Luminary Micro microcontroller family, STMicroelectronics famiglia STM32, NXP famiglie LPC17xx e LPC13xx, ATMEL famiglia AT91SAM3x.
Cortex	v7E-M	Arm Cortex-M4	Freescale Kinetis, NXP
XScale	v5TE	80200 IOP310 IOP315	
XScale	v5TE	80219	
XScale	v5TE	IOP321	Iyonix
XScale	v5TE	IOP33x	
XScale	v5TE	PXA210/PXA250	Zaurus SL-5600, IPAQ 54xx
XScale	v5TE	PXA255	Gumstix, IPAQ 55xx
XScale	v5TE	PXA26x	
XScale	v5TE	PXA27x	HTC Universal, Zaurus SL-C1000,3000,3100,3200
XScale	v5TE	PXA800 (E)F	
XScale	v5TE	Monahans	
XScale	v5TE	PXA900	Blackberry 8700
XScale	v5TE	IXC1100	
XScale	v5TE	IXP2400/IXP2800	
XScale	v5TE	IXP2850	
XScale	v5TE	IXP2325/IXP2350	
XScale	v5TE	IXP42x	NSLU2
XScale	v5TE	IXP460/IXP465	

Utilizzare un sistema Embedded Linux comporta la valutazione attenta di vantaggi e svantaggi di una simile soluzione.

I principali vantaggi sono alcuni di carattere tecnico come la possibilità di avere a disposizione un layer di gestione dell'hardware molto evoluto e consolidato, la semplicità della programmazione di applicativi che si svincola dalla programmazione tradizionale dei microcontrollori e sfutta il sistema operativo per assegnare le risorse hardware a disposizione.

Altri sono quelli produttivi, infatti adottando un sistema operativo basato su kernel Linux il codice delle applicazioni diventa riutilizzabile, inoltre i tempi di programmazione sono ridotti grazie alla possibilità di utilizzare un sistema di

librerie standardizzato. Questo si traduce in una riduzione di tempi e costi del software.

Infine utilizzare un sistema operativo ed un kernel open source dà ampia capacità di personalizzazione e soprattutto grande supporto a livello tecnico, potendo contare su una documentazione molto vasta.

Ovviamente esistono anche diversi limiti di utilizzo di questi sistemi, infatti avere un sistema operativo vuol dire delegare a questo molti dei compiti di gestione delle risorse hardware, ciò si traduce nella non completa padronanza delle tempistiche da parte del programmatore software.

Il limite dunque è la difficoltà nell'ottenere un sistema hard Real-Time, quindi per tutti quei dispositivi dove si richiede una precisione nelle tempistiche molto rigida è probabilmente piu semplice affidarsi ad un microcontrollore privo di sistema operativo, gestendo direttamente le interrupt e eventuali delay in modo preciso e controllato.

La conclusione è che i dispositivi Embedded Linux riescono a coprire la maggior parte delle applicazioni comuni, essendo molto adattabili e versatili, ma in caso di applicazioni critiche hard Real-Time è più conveniente l' impiego di un sistema privo di sistema operativo, a meno di non voler impiegare sistemi operativi Real-Time (RTOS).

1.4 Real-Time

Un sistema di elaborazione opera in tempo reale soltanto se fornisce i risultati attesi entro prestabiliti limiti temporali (dipendenti dal contesto applicativo).

Quando viene affrontato un progetto software/hardware in ambito industriale vengono richiesti determinati requisiti, ad esempio:

- correttezza
- efficienza
- affidabilità
- flessibilità
- portabilità
- riusabilità

A questi primi requisiti se ne affianca un altro che può essere più o meno stringente ovvero la

- predicibilità

Definita dunque un' applicazione in termini di processi cooperanti caratterizzati da:

- prefissate interazioni
  - vincoli di precedenza
  - risorse condivise

- prefissate specifiche temporali
  - frequenza (max) di esecuzione
  - tempo max di elaborazione richiesto ad ogni esecuzione
  - tempo limite di completamento di ogni elaborazione

E selezionata l'architettura da voler impiegare per il sistema di elaborazione che potrà essere monoprocessore o multiprocessore, occorrerà individuare una opportuna strategia di esecuzione dei processi in modo tale da rispettare tutti i vincoli imposti dall'applicazione.

Più precisamente è fondamentale chiarire le tipologie di schedulazione da usare. Esistono principalmente sei diversi tipi di schedulazione:

1. *off-line:* Ovvero una schedulazione pianificata in modo integrale a priori, aspetto fondamentale è dunque la conoscenza preventiva del processo che si vuole modellare.
2. *on-line:* Ovvero la schedulazione dei vari processi viene effettuata in funzione di alcuni parametri o pesi che vengono attribuiti ai processi. In questo senso troviamo due possibili soluzioni: static e dynamic.
3. *guaranteed:* La schedulazione in questo caso ha l'obbligo di rispettare le tempistiche tu tutti i processi che fanno parte di un certo set.
4. *best-effort:* Tale metodologia di schedulazione si può contrapporre all' approccio guaranteed in quanto tende a ottimizzare le prestazioni medie dell' insieme dei processi, per cui non pone dei vicoli assoluti sui singoli task.
5. *preemptive:* La tipologia preemptive da l'opportunità di salto dello scheduler ovvero permette la sospensione temporanea di un processo (salvando il contesto).
6. *non-preemptive:* Non-preemptive rappresenta una tipologia contraria a preemptive i task non vengono interrotti e continuano nella loro eseguzione fin tanto chè non rilasciano volontariamente le risorse.

Similmente anche i task o processi possono essere categorizzati e distinti in diverse tipologie.

Abbiamo due macrotipi di processi quelli Real-Time e quelli non Real-Time, esiste però un'importante differenziazione all' interno dei processi detti Real-Time.

Possiamo distinguere i processi Real-Time in:

- **Hard:**
  Se la relativa deadline deve essere sempre rispettata.
  - periodico: Quando il processo presenta frequenza di esecuzione costante
  - sporadico: Quando viceversa il processo ha frequenza variabile.

- **Soft:**
  Se la relativa deadline può essere disattesa in condizioni di temporaneo sovraccarico del sistema.
  - periodico: Come nel caso Hard il processo è periodico se ha frequenza di esecuzione costante.
  - aperiodico: Caso contrario di periodico.

Queste sono le categorie in cui vengono classicamente suddivisi i processi. Introduciamo ora una funzione detta di utilità di un' applicazione composta da un insieme di n task Soft e Hard:

$$K_s = \sum_0^\infty K_i\,(t)$$

I termini K_i rappresentano i costi del task-iesimo, un task ha una funzione di costo tale che:

- soft task: r=f(t)
- hard task: r=∞

$$K_i\,(t) \quad = \quad \begin{cases} 0 & t < d \\ \mathrm{r} & t >= d \end{cases}$$

dove d rappresenta il tempo di deadline del task.

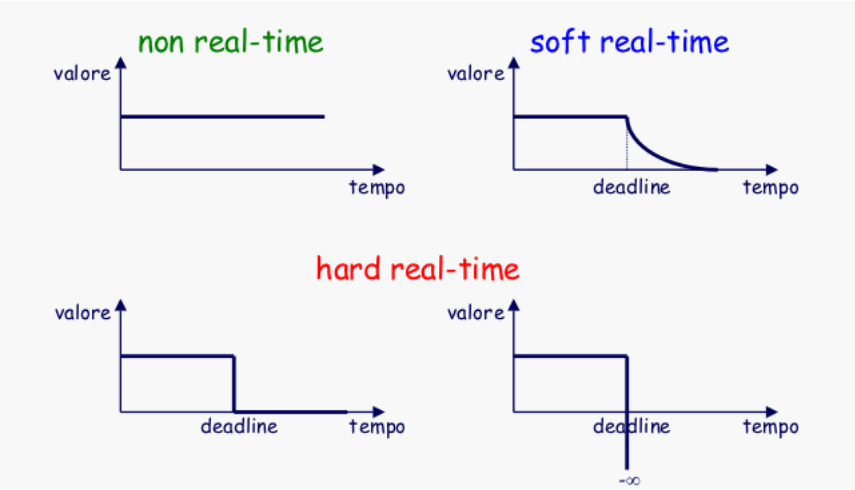


Figura 1.1: Funzione di utilità di un processo

Graficamente in figura 1 si nota come la funzione di utilità nel caso di processi non Real-Time è costante e invariante dunque rispetto al tempo .

Situazione completamente diversa si presenta andando a considerare la tipologia di processo Real-Time in questo caso abbiamo che per quanto riguarda i task soft la funzione è costante all' interno della deadline e descresce esponensialmente per tempi maggiori della deadline. La situazione per un hard task Real-Time è ancora più restrittiva in questo caso la funzione utilità può essere interpretata come un gradino per tempi maggiori o uguali alla deadline il suo valore sarà 0 o più classicamente meno infinito(fallimento completo del processo).

La complessità all' interno delle varie tecniche di schedulazione è molto elevata, possiamo affermare che dato un insieme di task modellati come una terna p,c,f rispettivamente priorità, tempo di esecuzione e frequenza non esiste una tipologia di schedulazione ottimale quindi non esiste un' unico algoritmo di schedulazione ottimo.

In base dunque alla tipologia del dispositivo e soprattuto alla terna p,c,f dipenderà la scelta di un tipo di schedulazione anzicchè un' altra.

In seguito in questo documento parleremo espressamente delle tecniche usate da Linux per realizzare un sistema che possa garantire un certo tipo di Real-Time (soft) a tale proposito è interessante mostrare alcuni esempi di schedulazione preemptive e non preemptive.

Consideriamo una generica applicazione costituita da n task P_n con n=9. Siano fissate le priorità di tali processi in modo che

$$p(P_i) < p(P_j) \qquad \forall \quad i,j \quad con \quad i > j$$

esistano tra i vari processi dell' applicazione dei vincoli di precedenza di questo tipo:

$$P_1 \prec P_9, P_4 \prec P_5, P_4 \prec P_6, P_4 \prec P_7, P_4 \prec P_8$$

In cui $\prec$ rappresenta la precedenza del processo a sinistra dell'operando rispetto al processo sulla destra.

I processi siano caratterizzati dai seguenti tempi di esecuzione:

$$C_1 = 3, \quad C_2 = C_3 = C_4 = 2, \quad C_5 = C_6 = C_7 = C_8 = 4, \quad C_9 = 9$$

Infine fissiato un certo sistema (dispositivo fisico) di elaborazione E con $\prod_k$ processori indipendenti.

Considerando quanto detto e fissato $k = 3$ analizziamo le differenze prestazionali nel caso di schedulazione preemptive e non preemptive:

- **Caso non preemptive:**

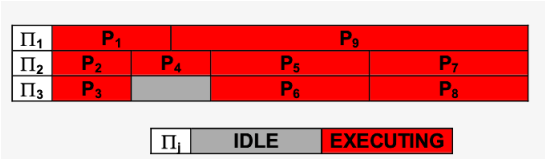


Figura 1.2: Immagine del flusso di esecuzione temporale di un' applicazione secondo una tipologia di schedulazione non preemptive.

In figura 2 viene mostrato attraverso un diagramma temporale semplicifato come vengono schedulati i processi con una molteplicità pari a 3 come processori. Si nota in questo caso specifico un tempo di Idle ridotto indice di un buon utilizzo complessivo delle risorse di calcolo.

- **Caso preemptive:**

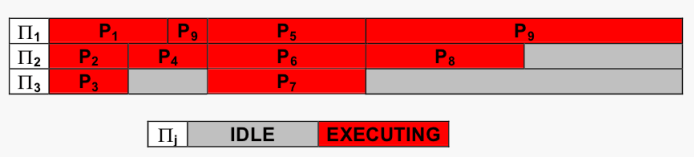


Figura 1.3: Immagine del flusso di esecuzione temporale di un' applicazione secondo una tipologia di schedulazione preemptive.

In figura 3 si nota come in questo caso specifico il tempo di Idle delle CPU è piuttosto lungo indice di una cattiva gestione delle risorse hardware rispetto al caso visto in figura 1.

Mettendo a confronto le due "simulazioni" temporali delle tipologie di schedulazione vediamo che il tempo impiegato per rispondere completamente all' applicazione è maggiore nel caso preemptive.

Una schedulazione non preemptive dunque dà un certo vantaggio in questo caso specifico.

E' importante tenere presente che i precedenti grafici sono del tutto qualitativi e sono molti i parametri che non vengono presi in considerazione, ad esempio il fatto di assumere lo switch di contesto tra processi istantaneo.

Le situazioni descritte fino ad adesso non si sono occupate dell'aspetto forse più importate alla base della schedulazione ovvero la strategia di assegnazione delle priorità.

Dalle considerazioni fatte possiamo comunque fissare già alcuni dei "*building block*" per la costituzione di un modello più specifico per la schedulazione di processi hard Real-time periodici e/o sporadici all' interno di uno scenario ben preciso.

Il modello base si compone dei seguenti elementi:

- N processi P1, P2, ..., PN indipendenti
  - senza vincoli di precedenza
  - senza risorse condivise
- ogni processo Pj (j = 1, 2, ..., N) è caratterizzato da tre parametri temporali

  - Tj: periodo nel caso di processi periodici, minimum interarrival time nel caso di processi sporadici
  - Dj: deadline relativa, con Dj = Tj nel caso di processi periodici, Dj < Tj nel caso di processi sporadici
  - Cj: tempo massimo di esecuzione, con Cj ≤ Dj

- l'esecuzione dei processi è affidata ad un sistema di elaborazione monoprocessore
- ad ogni processo Pj è staticamente o dinamicamente associata una priorità p(Pj) = pj dipendente dalla corrispondente deadline, relativa o assoluta rispettivamente
- ad ogni processo è dinamicamente associata una informazione che ne identifica lo stato ai fini della schedulazione:
  - processo IDLE, READY (SUSPENDED), RUNNING
- l'esecuzione di un processo è sospesa quando un altro processo di priorità superiore è pronto per l'esecuzione (preemption)
- il tempo impiegato dal processore per operare una commutazione di contesto tra processi è trascurabile

Costruito il modello adesso vanno considerate le strategie di schedulazione in relazione ad alcune caratteristiche dei task stessi.

Di seguito alcune delle strategie più adottate.

- *MONOTONIC PRIORITY ORDERING* (RMPO):
  - Tipologia di processi: periodici.

  Ad ogni processo è staticamente associata una priorità tanto maggiore quanto minore è la corrispondente deadline relativa (periodo).
- *DEADLINE MONOTONIC PRIORITY ORDERING* (DMPO):
  - Tipologia di processi: periodici e/o sporadici.

  Ad ogni processo è staticamente associata una priorità tanto maggiore quanto minore è la corrispondente deadline relativa.
- *EARLIEST DEADLINE FIRST* (EDF):
  - Tipologia di processi: periodici e/o sporadici.

  Ad ogni processo è dinamicamente associata una priorità tanto maggiore quanto più prossima è la corrispondente deadline assoluta.

Per ogni algoritmo esistono una serie di teoremi e corollari che danno la possibilità di capire quali sono le condizioni necessarie (a volte anche sufficienti) perchè un certo insieme di task sia schedulabile in un sistema Real-Time.

Il teorema di schedulabilità a tale proposito afferma che:

Condizione necessaria (ma in generale non sufficiente) affinché un insieme di N processi periodici sia schedulabile è che il risultante fattore di utilizzazione del processore sia non superiore a 1:

$$U = \sum_{j=1}^{N} U_j = \sum_{j=1}^{N} \frac{C_j}{T_j} \leq 1$$

Il j-esimo termine della sommatoria Cj / Tj = (Cj (T / Tj)) / T rappresenta la frazione dell'iperperiodo T = m.c.m. (T1, T2, ..., TN) richiesta per l'esecuzione di Pj.

Concludiamo facendo notare come sia critica la scelta dello scheduler all' intero di un sistema operativo, dalla sua realizzazione (e non solo) dipende buona parte delle caratteristiche che potrà offrire in termini di Real-Time. Inoltre i sistemi di elaborazione in tempo reale rappresentano un ambito in costante sviluppo e indubbiamente ancora oggi sono un interessante settore di ricerca informatico.

1.5 Linux Real-Time

Il più grosso problema riguardo al Real-time di Linux è sempre stato che i cambi di contesto (ad esempio il passaggio ad un processo ad elevata priorità) possono (o meglio potevano, fino all'avvento del kernel 2.6) avvenire solo al passaggio dallo spazio del kernel a quello utente. Linux 2.6 introduce miglioramenti che lo rendono molto più valido che in passato in applicazioni dove il tempo di risposta è importante:

- sono stati inseriti dei punti di "preemption" nel kernel, cioè punti nei quali può girare lo scheduler. Fino ad ora ciò non era possibile, principalmente per motivi di semplificazione nel preservare l'integrità delle strutture dati del kernel stesso, ma ciò poteva provocare problemi in quanto un processo a priorità molto alta (e dalle specifiche temporali stringenti) poteva essere ritardato di molto in attesa che una system-call di un altro processo terminasse. Per risolvere questo problema uno dei metodi utilizzati erano le cosiddette "preemption patch" e "low-latency patch", ora introdotte ufficialmente nel kernel.

- è stato ottimizzato lo scheduler, implementando algoritmi più efficienti in modo da ridurre l'overhead introdotto dal S.O., soprattutto in presenza di molti "task".

- il sistema può girare anche su sistemi privi di memoria virtuale (allargando di molto la base di processori supportati, cioè tutti quelli sprovvisti di MMU). Infatti un software che deve rispondere entro certe "deadline" è in un certo senso incompatibile con la memoria virtuale, a causa dell' impredicibile ed eventuale lenta elaborazione dei page faults, che possono rovinare la prontezza di risposta del sistema.

- sono state introdotte nuove primitive di sincronizzazione, che permettono di usare i "mutex" senza ricorrere a chiamate di sistema (che introducono sempre un certo overhead).

- sono stati introdotti nel kernel il supporto per le segnalazioni e per i timer ad alta risoluzione POSIX (oltre ad una ottimizzazione del meccanismo dei thread POSIX).

Un altro miglioramento a favore dei sistemi Embedded è l'introduzione del concetto di "sub-architettura": i componenti del sistema sono chiaramente separati, quindi se ad esempio deve cambiare il codice per gestire una interruzione poiché è cambiato l'hardware, il codice specifico può essere modificato con un minimo impatto sul resto del sistema. È inoltre possibile eliminare completamente il codice per gestire video, tastiera, ecc... ove questi non risultino necessari.

Nonostante tutti questi miglioramenti Linux come già sottolineato non è un vero sistema hard Real-Time. Per ottenere prestazioni Real-time occorre ancora ricorrere a patch del kernel, delle quali una delle più utilizzate è RTAI, sviluppata dal Politecnico di Milano (www.rtai.org).

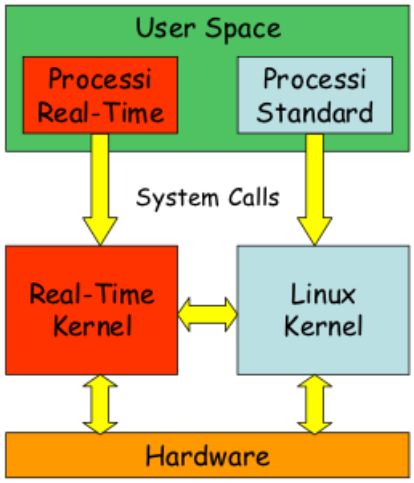


Figura 1.4: Implementazione hard Real-Time su Linux con la tecnica "RTAI"

Si tratta di una delle soluzioni attualmente più efficaci per rendere Linux un sistema veramente Real-time. Essa prevede che un secondo kernel, di tipo hard Real-Time, prenda il controllo effettivo della macchina e faccia girare Linux come una propria applicazione a priorità più bassa. Inoltre il kernel Linux non ha più il controllo diretto sull'abilitazione e la disabilitazione delle interruzioni: queste vengono intercettate da RTAI che sostituisce gli handler Linux con dei propri dispatcher. Lo schema di architetturale descritto precedentemente è presentato in figura 4 tramite uno schema a blocchi .

Molte distribuzioni Embedded commerciali includono direttamente RTAI.

Un sistema Real-Time (RT) non deve essere confuso con "sistema veloce". Un Processo Real-Time deve terminare rispettando i vincoli temporali (le deadline) stabiliti in modo tale che la sua esecuzione abbia senso. Dunque un Sistema Real-Time deve essere in grado di prevedere se i processi che andrà ad eseguire siano in grado di rispettare le rispettive deadline.

La specifica di un sistema operativo Real-Time è di garantire a priori il corretto funzionamento di un preciso insieme di processi in modo deterministico.

I fattori da cui dipende la previsione deterministica delle tempistiche dei processi sono:

1. ***Le caratteristiche architetturali dell'elaboratore***
2. ***Il linguaggio di programmazione***
3. ***Il sistema operativo***

Ad esempio le architetture basate su CPU "*general purpose*" minano fortemente il determinismo, viceversa i microcontrollori *DSP-like* si prestano particolarmente a garantire determinate latenze di risposta *(negli interrupt)*.

1.5.1 Le caratteristiche architetturali dell'elaboratore

Esistono alcuni elementi architetturali che minano il determinismo:

- **Arbitrato del bus causato da dispositivi di I/O "intelligenti"**

  Un esempio è l'uso del DMA che trasferisce dati dalle periferiche alla memoria senza chiamare in causa la CPU per ogni byte trasferito tramite il meccanismo usuale dell'interrupt e la successiva richiesta di operazione desiderata, ma generando un singolo interrupt per blocco trasferito. La CPU rilascia il controllo del bus (dati) e permette al DMAC di gestire la comunicazione con la periferica. Un meccanismo simile chiaramente rende poco deterministico il sistema tanto più se andiamo a considerare certe tecniche di DMA come il *Burst Transfer*. Un modo per ottenere una maggiore prevedibilità a discapito delle prestazioni è ad esempio l'impiego di tecniche DMA dove i cicli di accesso alla memoria sono ripartiti tra CPU e DMAC.

- **Cache multi-livello**

- **Memoria virtuale e unità di gestione della memoria (MMU)**

- **Pipeline di esecuzione delle istruzioni e predizione dei branch**

  Rappresentano elementi soggetti a ritardi anche superiori al tempo di esecuzione delle istruzioni. Introducono un fattore di aleatorietà.

  Ad esempio ogni cache fault provoca l'accesso in memoria. Ogni page fault provoca l'accesso al disco. Ogni predizione sbagliata provoca la perdita della pipeline. Occorre considerare il caso peggiore (es. cache fault per ogni accesso in memoria) che è improbabile ma possibile. Dunque per incrementare il determinismo sarebbe preferibile non avere memoria cache o disabilitarla.

- **Gestione delle interruzioni generate da dispositivi periferici**

Tipicamente nei sistemi non Real-time le routine di servizio delle interruzioni sono prioritarie rispetto alle applicazioni. Tuttavia può non essere vero nei sistemi Real-Time.

1.5.2 Il linguaggio di programmazione

Le caratteristiche del linguaggio di programmazione sono fondamentali per il determinismo della schedulazione dei task infatti è proprio il linguaggio ad alto livello che deve prevedere esplicitamente la gestione dei vincoli temporali. A tale proposito lo standard IEEE POSIX rappresenta un vero e proprio punto di riferimento.

1.5.3 Il sistema operativo

Il Sistema Operativo per quanto detto è fondamentale, il supporto completo per lo standard IEEE POSIX è sicuramente un fattore decisivo nella scelta di un OS Real-Time.

Il sistema operativo di riferimento completamente conforme allo standard POSIX è LynuxWorks LynxOS Embedded la sua certificazione fa si che venga spesso impiegato in ambienti governativi.

1.6 Utilizzo dello Scheduler Linux in user space

Dopo questo "overview" piuttosto ampio sui sistemi operativi e le loro caratteristiche Real-Time passiamo all' analisi effettiva delle capacità di Linux in questo contesto.

Nella versione 2.6 come già accennato il kernel Linux ha subito diversi cambiamenti per migliorare il supporto Real-Time, innanzitutto chiariamo la struttura interna di Linux così da analizzare precisamente i punti chiave del lavoro svolto dalla versione 2.4 alla 2.6.

Un task può operare in due contesti di esecuzione :

- User Mode
  - Modalità di esecuzione dei programmi utente, vengono adottate politiche di sicurezza per evitare l'accesso a zone critiche del sistema
- Kernel Mode
  - Modalità di esecuzione privilegiata che permette di accedere incondizionatamente a tutte le risorse del sistema. Questa rappresenta la modalità di esecuzione delle chiamate di sistema (System Calls)

Le System Calls sono lo strato software con cui i processi utenti possono accedere alle risorse hardware ovvero le funzioni con cui il SO (kernel Linux) fornisce servizi all'utente.

Le System Calls hanno una certa modalità di attivazione che è quella di un interrupt software.

La loro esecuzione avviene in modalità kernel.

Lo schema seguente rappresenta la differenziazione tra le due modalità mettendo in evidenza il ruolo chiave delle System Calls.

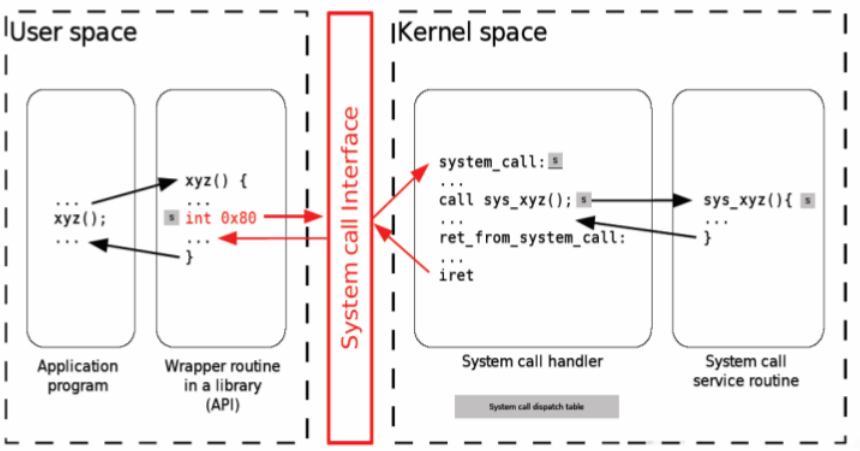


Figura 1.5: Schema struttura Linux

E' possibile effettuare preemption sui processi utente. Ad ogni processo sono associate una politica di scheduling e una priorità.

Le politiche di scheduling (POSIX standard) di Linux sono :

- Per processi tradizionali:
  - OTHER: politica di default, priorità statica pari a 0
- Per processi Real-Time:
  - ROUND-ROBIN (RR)
  - FIFO

    entrambe con priorità statica compresa tra 1 e 99

Banalmente i processi schedulati attraverso le politiche Rounf-Robin e Fifo sono prioritari rispetto ai processi tradizionali.

1.6.1 Scheduling Other

Ad ogni processo è assegnato un quanto di tempo (time slice) massimo di uso della CPU ed una priorità dinamica data dalla somma di un valore di base e di un valore che decresce all'aumentare del tempo di CPU utilizzato. Quando tutti i quanti di tempo dei processi pronti sono esauriti vengono ricalcolate le priorità di tutti i processi, avviene dunque un aggiornamento dinamico della priorità. Lo scopo per cui è progettato è quello di garantire un'equa ripartizione della CPU tra tutti i processi, e di fornire buoni tempi di risposta per i processi interattivi.

1.6.2 Scheduling FIFO

Ogni processo ha solo un valore di priorità statica. I processi al medesimo livello di priorità sono accodati secondo l'istante di attivazione (FCFS). Il processore viene assegnato al processo pronto a priorità maggiore.

Un processo in esecuzione perde l'uso della CPU se e solo se termina, se si sospende o se c'è un processo pronto a priorità maggiore.

Quando un processo perde l'uso della CPU è posto in testa alla lista dei processi pronti al suo livello di priorità (eccetto in caso di terminazione).

1.6.3 Scheduling Round-Robin

Per ogni processo è definito un valore di priorità statica e un periodo di tempo massimo durante il quale è assegnato alla CPU (time quantum). Il processore è assegnato al processo pronto a priorità maggiore.

Il processo che è in esecuzione perde l'uso della CPU se termina, se si sospende, se c'è un processo pronto a priorità maggiore o dopo aver esaurito il proprio time quantum. Quando un processo termina il proprio quanto di tempo viene posto alla fine della lista dei processi pronti al suo livello di priorità. Inoltre quando un processo subisce preemption da parte di un processo più prioritario è posto in testa alla lista dei processi pronti al suo livello di priorità.

Le policy di schedulazione descritte precendentemente vengono effettivamente applicate attraverso l' uso di alcune primitive System Calls che provvedono alla gestione dello scheduler.

Analizzando l'heder file sched.h è possibile isolare le primitive più importanti:

- sched_setscheduler()
  - Setta l'algoritmo di scheduling (SCHED_FIFO, SCHED_RR, SCHED_OTHER) e la priorità statica per un processo. Richiede i privilegi di root.
- sched_getscheduler()
  - Restituisce l'algoritmo di scheduling di un processo
- sched_setparam()
  - Setta la priorità statica di un processo
- sched_getparam()
  - Ricava la priorità statica di un processo
- sched_rr_get_interval()
  - Restituisce il valore del quanto di tempo assegnato ad un processo

- sched_get_priority_min()
  - Restituisce il minimo valore di priorità possibile per un algoritmo di scheduling
- sched_get_priority_max()
  - Restituisce il massimo valore di priorità possibile per un algoritmo di scheduling
- sched_yield()
  - Fa assumere al processo chiamante lo stato READY

Vediamo un esempio pratico in C dell'uso delle primitive appena viste su un sistema Linux. Il codice presentato nell'algoritmo 1 generara all' interno del main 5 processi figli attraverso un fork e per ogni processo perde tempo per effettuare una serie di iterazioni. Ogni processo figlio viene schedulato secondo una priorità diversa crescente attraverso un define (PRIORITA). Il tipo di schedulazione è impostato come FIFO.

Algoritmo 1: Esempio in linguaggio C dell' uso dello scheduler in Linux

```
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#define PRIORITA child_number
#define ALG_SCHEDULING SCHED_FIFO
#define N_ITE 10
#define N_CHILD 5
#define SLEEP_TIME 100000

void new_child(int child_number)
{
        int i=0;
        struct sched_param priority;
        priority.sched_priority = PRIORITA;
        if (sched_setscheduler(0, ALG_SCHEDULING, &priority) < 0)
        {
                printf("Errore nell'impostazione dello scheduler\n");
                exit(-1);
        }
        usleep(SLEEP_TIME);
        printf("Processo %d iniziato priorita %d\n", child_number,PRIORITA);
        for (i = 1; i <= N_ITE; i++)
        {
                printf("Processo%d:iterazione%d\n",child_number,i);
                usleep(SLEEP_TIME);
        }
        printf("Processo %d terminato\n",child_number); exit(0);
}
int main(void)
{
        int count; int fork_return; /* Crea i processi figli */
        for (count = 1; count <= N_CHILD; count++)
        {
                if ((fork_return = fork()) == 0)
                {
                        new_child(count);
                } else
                {

                }
        }
}
```

L'output dell'algoritmo 1 eseguito su un sistema a singolo processore dovrebbe mostrare l'esecuzione completa del processo con priorità più alta comprese

le sue iterazioni prima di proseguire al processo con priorità più bassa, così per tutti i figli in ordine decrescente di priorità.

Le migliorie apportate nella versione 2.6 del kernel Linux riguardano più precisamente:

- L'ottimizzazione dell' algoritmo di scheduling O(1) così che l'overhead dovuto alle operazioni di scheduling è determinabile a priori e non è dipendente dal numero di processi attivi .
- La frequenza interna del clock 1000 Hz ha portato il periodo minimo di esecuzione dello scheduler da 10 ms (versione 2,4) a 1 ms.
- Introduzione di diversi punti di preemption nel kernel in cui è possibile eseguire lo scheduling.

Tutti questi accorgimenti non bastano a garantire il determinismo necessario poichè andando ad analizzare la struttura delle System calls, interrupt ed exception (eseguiti tutti in spazio kernel) notiamo che queste risultano annidabili per cui il tempo di attesa per l'assegnazione del processore ad un processo più prioritario di quello in esecuzione è indeterminabile.

Il Kernel 2.6 per quanto detto risulta essere un ottimo punto di partenza per lo sviluppo di un sistema hard Real-Time, oltre ad offrire un discreto supporto Soft Real-Time.

1.7 Applicazioni e sviluppo sistemi Linux

L'uso di Linux e delle sue varianti Real-time è di grande interesse per applicazioni, anche significative, di automazione industriale, di supervisione, di robotica e nei sistemi di controllo distribuito.

Nel campo dell'automazione si può sfruttare Linux, oltre che come sistema da inserire in un hardware dedicato, anche come piattaforma di sviluppo e prototipazione rapida su piattaforma PC.

Nell'ambito dei sistemi di controllo distribuiti, a tutt'oggi si può riscontrare che non vi sono sistemi e macchine realmente funzionanti in modo distribuito. Soprattutto mancano strumenti metodologici generali e applicativi specifici in grado di aiutare i progettisti e i tecnici nel progetto di ambienti di controllo in tempo reale per sistemi distribuiti.

A questo scopo sembra particolarmente interessante la possibilità di sviluppare nuovi strumenti di controllo in tempo reale basati su ambienti software "open source" (tipicamente Linux e le sue varianti Real-Time, come RTAI e RT-Linux), che integrino notevoli capacità computazionali, possibilità di facili riconfigurazioni in ambienti distribuiti, capacità di simulazione di sistemi dinamici e algoritmi di controllo non banali e funzionanti su piattaforme hardware facilmente reperibili in commercio e di costo contenuto.

Lo scenario attuale dei sistemi operativi in ambiente Embedded mostra un trend positivo per Linux Embedded sia in ambiente industriale che per prodotti commerciali.

Tale situazione attuale mostra uno schieramento delle grandi aziende con Linux tra le più importanti troviamo anche diversi leader dell'elettronica di consumo e dei sistemi operativi, quali: Nokia, Motorola, IBM, ARM, Samsung,

Wind River, Metrowerks e MontaVista. Infine recentemente si assiste alla spinta commerciale impressa da Google al sistema operativo per PDA e Telefonini Android, interamente derivato da Linux .

Tutto ciò conferma che il fenomeno Linux Embedded non è solo una moda o qualcosa di cui si parla, ma è una realtà (una "unstoppable force", sostiene MontaVista) da prendere seriamente in considerazione per nuovi progetti.

Il seguente grafico in figura 6 mostra i fattori che hanno maggiore influenza nella scelta di una soluzione Linux Embedded.

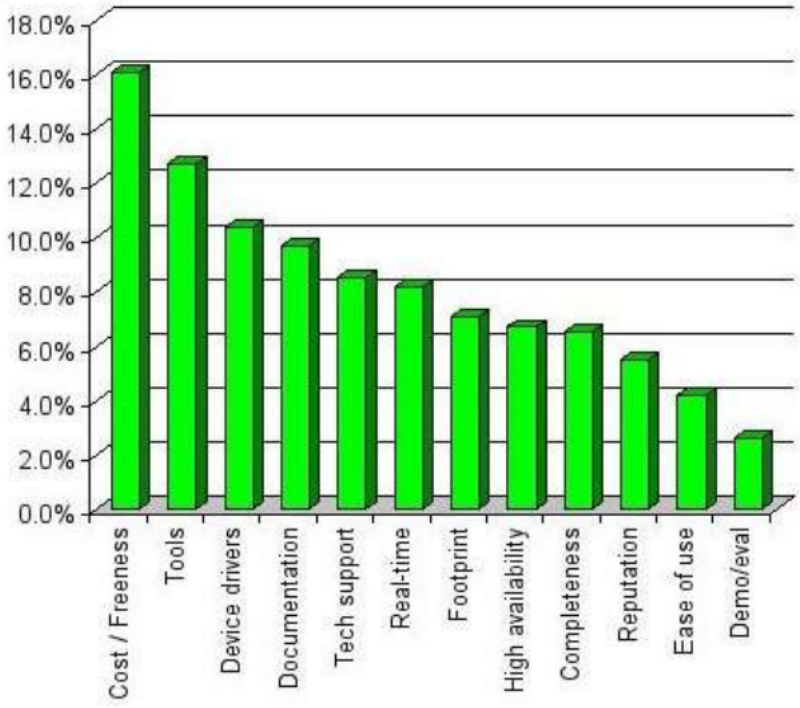


Figura 1.6: Fattori di influenza nella scelta di un sistema basato su Linux Embedded in campo industriale

Il fattore più interessante nella scelta di GNU-Linux anzicchè un sistema operativo Embedded proprietario è senza dubbio l'aspetto economico.

La questione relativa ai costi totali di sviluppo, di 'ownership' e del 'time-to-market' di una soluzione basata su Linux piuttosto che su di un sistema proprietario (tra cui in particolare Windows Embedded) è una questione molto dibattuta. Vanno tenuti presenti alcuni aspetti non immediati.

Da alcune parti si sente dire (anche in base ad indagini di mercato) che alla fine i costi totali (in termini di tempo e di denaro) di uno sviluppo con Linux sono maggiori rispetto ad uno sviluppo fatto con un sistema proprietario (Windows in particolare). A parte che le statistiche non sono facilmente generalizzabili, né facili da realizzare correttamente, poiché il tempo/costo di sviluppo va messo in relazione con la difficoltà del progetto, i fattori da considerare sono numerosi e l'analisi non è facile, questo può essere senz'altro vero se si decide di sviluppare una distribuzione in proprio. Infatti, il fatto che Linux sia disponibile gratuitamente può trarre in inganno: l'assemblaggio e la manutenzione di una distribuzione e la realizzazione di personalizzazioni (peraltro impossibili con i sistemi proprietari) possono richiedere un grosso numero di 'ore-uomo', aumentando anche il time-to-market. Tuttavia rimangono a favore di Linux altri fattori: è royalty-free (cosa che comunque spesso ha impatto solo sui grossi volumi), è meno costoso trovare sviluppatori e formarli, sono disponibili tools

di sviluppo gratuiti, sono disponibili distribuzioni off-the-shelf ('commerciali' o meno, comunque open source), da cui eventualmente partire come base nel caso si voglia sviluppare una distribuzione personalizzata. Comunque, spesso il costo di sviluppo non è il solo fattore che fa la differenza . Molti altri fattori sono importanti (oltre a quelli già visti) tra cui la reale differenziazione del prodotto dalla concorrenza ed il suo costo finale (chiaramente il prodotto sarà più competitivo se il sistema operativo può girare su un hardware più economico).

Capitolo 2

Caratteristiche del sistema Embedded

2.1 Panoramica del sistema

Esponiamo adesso in modo specifico le caratteristiche del sistema Embedded su cui ho lavorato durante il tirocinio. La finalità del progetto di cui ho fatto parte è la realizzazione di un sistema Embedded Linux basato su processore ARM926EJ-S precisamente un microcontrollore Atmel at91SAM9260, un sistema di questo tipo di fatto nonostante molte caratteristiche Embedded ha un'ampia possibilità di impiego. Nel nostro caso il dispositivo sviluppato sarà utilizzato principalmente per il telecontrollo. Fornirà diversi servizi attraverso un web server, a tale proposito verrà sviluppata un' interfaccia PHP che si occuperà di visualizzare tutti i dati relativi al monitoraggio ed il controllo dello stato degli I/O digitali e degli ingressi analogici (ADC converter). Il sistema avrà inoltre un supporto specifico a livello applicativo in linguaggio C/bash per potersi interfacciare ad eventuali dispositivi di rilevamento o azionamento. Infine tramite la tecnologia GPRS e GPS sarà possibile la localizzazione del dispositivo in ogni momento. L'aspetto di cui mi sono occupato principalmente all' interno del progetto è la parte relativa al porting del kernel Linux su piattaforma ARM e la successiva configurazione base del filesystem partendo dal sistema di inizializzazione script di Openwrt. Openwrt è un progetto open source che si propone di fornire un sistema di build semi-automatico per la cross-compilazione di kernel e successiva generazione di filesystem.

Fissato il progetto e l'obiettivo del tirocinio passiamo adesso all'aspetto tecnico, a tale proposito andrò ad esporre le caratteristiche dell'hardware utilizzato dalla board. Lo schema a blocchi sottostante consente di apprezzare una panoramica generale sui componenti esterni del dispositivo e sulla loro interconnessione:

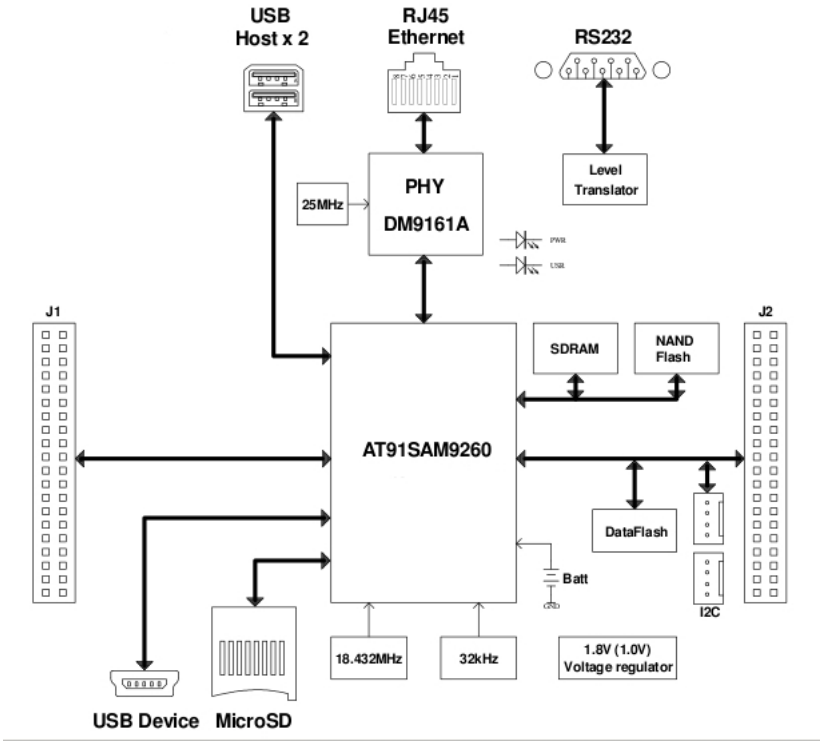


Figura 2.1: Mostra lo schema a blocchi del sistema Embedded e la connessione dei vari bus esterni

Nella figura 7 è mostrato lo schema a blocchi del sistema Embedded, notiamo che le memorie NAND Flash e SDRAM sono collegate attraverso un bus parallelo mentre l'eventuale memoria DataFlash è collegata su un bus seriale SPI.

Inoltre è da sottolineare la presenza delle porte seriali RS232 con il proprio traslatore di livello (MAX232), nel progetto del dispositivo sono previste un totale di 4 porte seriali, di cui specificheremo più avanti le caratteristiche e il loro impiego.

Il device usb apparentemente accessorio al sistema vedremo che avrà un ruolo fondamentale in fase di programmazione del dispositivo (SAM-BA boot).

I due host usb danno la possibilità di collegare al sistema periferiche usb di vario tipo (wireless, bluetooth, storage, dispositivi di acquisizione video ecc...) il lettore MicroSD garantisce un elevato grado di espandibilità del reparto storage, infine l'interfaccia ethernet RJ45 consente la comunicazione lan con pc e altri dispositivi eventualmente collegati in rete.

Il livello layer fisico del sistema OSI per l'interfaccia ethernet sarà un PHY dedicato davicom DM9161A che comunicherà con il microcontrollore ARM tramite MAC interface integrata da Atmel.

Passiamo adesso in rassegna i componenti principali del sistema Embedded iniziando dalle caratteristiche del microcontrollore Atmel at91sam9260 per poi

spingerci all' interno del processore ARM e la sua architettura.

L'at91 è una delle famiglie piu note di Atmel il microcontrollore at91sam9260 ha un tipo di architettura ARMv5TEJ, di fatto rappresenta il successore dello storico at91rm9200, il microcontrollore è costituito oltre che dal microprocessore ARM926EJ-S da 32 KB di rom, 4 KB di SDRAM e da un certo numero di periferiche di cui avremo modo di parlare in dettaglio nel seguito del capitolo.

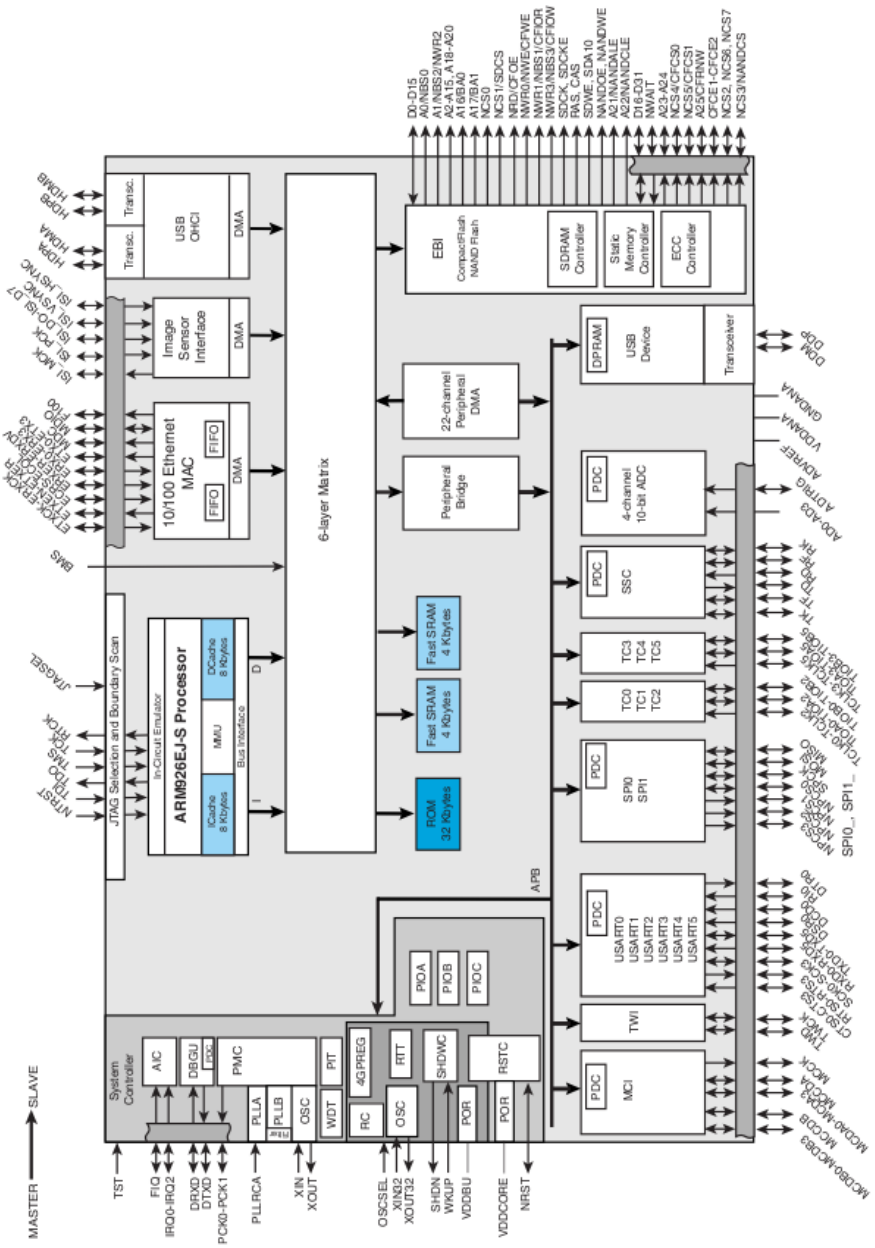


Figura 2.2: schema a blocchi at91sam9260

Il package utilizzato è il 217-LFBGA, la figura 8 ci mostra uno schema a blocchi del microcontrollore at91sam9260, vediamo che il processore ARM926EJ-S è collegato alle periferiche esterne tramite un grande bus a 6 layer lo schema

mette in risalto il collegamento distinto tra I e D cioè instructions e data a cui corrispondono due rispettive cache Icache e Dcache il bus a 6 layer permette al processore (I & D) di comunicare con le altre periferiche, esiste ovviamente una certa gerarchia le freccie indicano infatti il verso master/slave a questo proposito anche se in linea teorica ogni master dovrebbe essere in grado di comunicare con qualsiasi slave in realtà alcuni percorsi sono proibiti o semplicemente not wired (non cablati).

La tabella 2 seguente riassume la comunicazione dei dispositivi master in base alla loro classificazione:

Tabella 2.1: Elenco dei Bus-matrice master

Master 0	ARM926™ Instruction
Master 1	ARM926 Data
Master 2	PDC
Master 3	USB Host DMA
Master 4	ISI Controller
Master 5	Ethernet MAC

Istruzioni e dati sono distinti questo mette in evidenza la scelta architetturale ARM di tipo harvard, tale separazione permette, a differenza dell'architettura classica di von Neumann, di caricare contemporaneamente dati e istruzionui e quindi di eseguire più compiti in parallelo dato che il processore ha la possibilità di parallelizzare le operazioni di lettura e scrittura della memoria. L'aumento di velocità viene comunque compensato dalla presenza di circuiti più complessi all' interno del processore.

La tabella 3 similmente alla tabella 2 riporta l'elenco dei bus slave.

Tabella 2.2: Elenco dei Bus-matrice slave

Slave 0	Internal SRAM0 4 KBytes
Slave 1	Internal SRAM1 4 KBytes
Slave 2	Internal ROM
	USB Host User Interface
Slave 3	External Bus Interface
Slave 4	Internal Peripherals

2.1.1 PDC

Il PDC (Peripheral DMA Controller) a 22 canali permette alle periferiche di accedere alla memoria senza intervento del processore, nello schema a blocchi è riportata la dicitura PDC su ogni periferica che utilizza tale controller, la priorità sul canale viene gestita dal PDC nel seguente ordine dalla più bassa alla più alta:

1. – DBGU Transmit Channel

2. – USART5 Transmit Channel
3. – USART4 Transmit Channel
4. – USART3 Transmit Channel
5. – USART2 Transmit Channel
6. – USART1 Transmit Channel
7. – USART0 Transmit Channel
8. – SPI1 Transmit Channel
9. – SPI0 Transmit Channel
10. – SSC Transmit Channel
11. – DBGU Receive Channel
12. – USART5 Receive Channel
13. – USART4 Receive Channel
14. – USART3 Receive Channel
15. – USART2 Receive Channel
16. – USART1 Receive Channel
17. – USART0 Receive Channel
18. – ADC Receive Channel
19. – SPI1 Receive Channel
20. – SPI0 Receive Channel
21. – SSC Receive Channel
22. – MCI Transmit/Receive Channel

2.1.2 EBI

L'EBI (External Bus Interface) è l'interfaccia per il bus esterno di memoria, si compone di vari controller che permettono di gestire diversi tipi di memoria: NAND Flash, compat Flash e SDRAM.

Per meglio comprendere come viene mappato il bus esterno dal microcontrollore fissiamo alcuni concetti importati. Il microcontrollore può indirizzare fino a 4GB di memoria totale lo spazio di indirizzi è suddiviso in 16 banchi da 256MB. Il banco 0 e il banco 15 sono riservati rispettivamente alla mappatura della memoria interna e alla mappatura delle periferiche interne i banchi da 1 a 8 sono associati al bus esterno (EBI) con i propri chip select da 0 a 7 (EBI_NCS0 - EBINCS7) come mostra la figura 9.

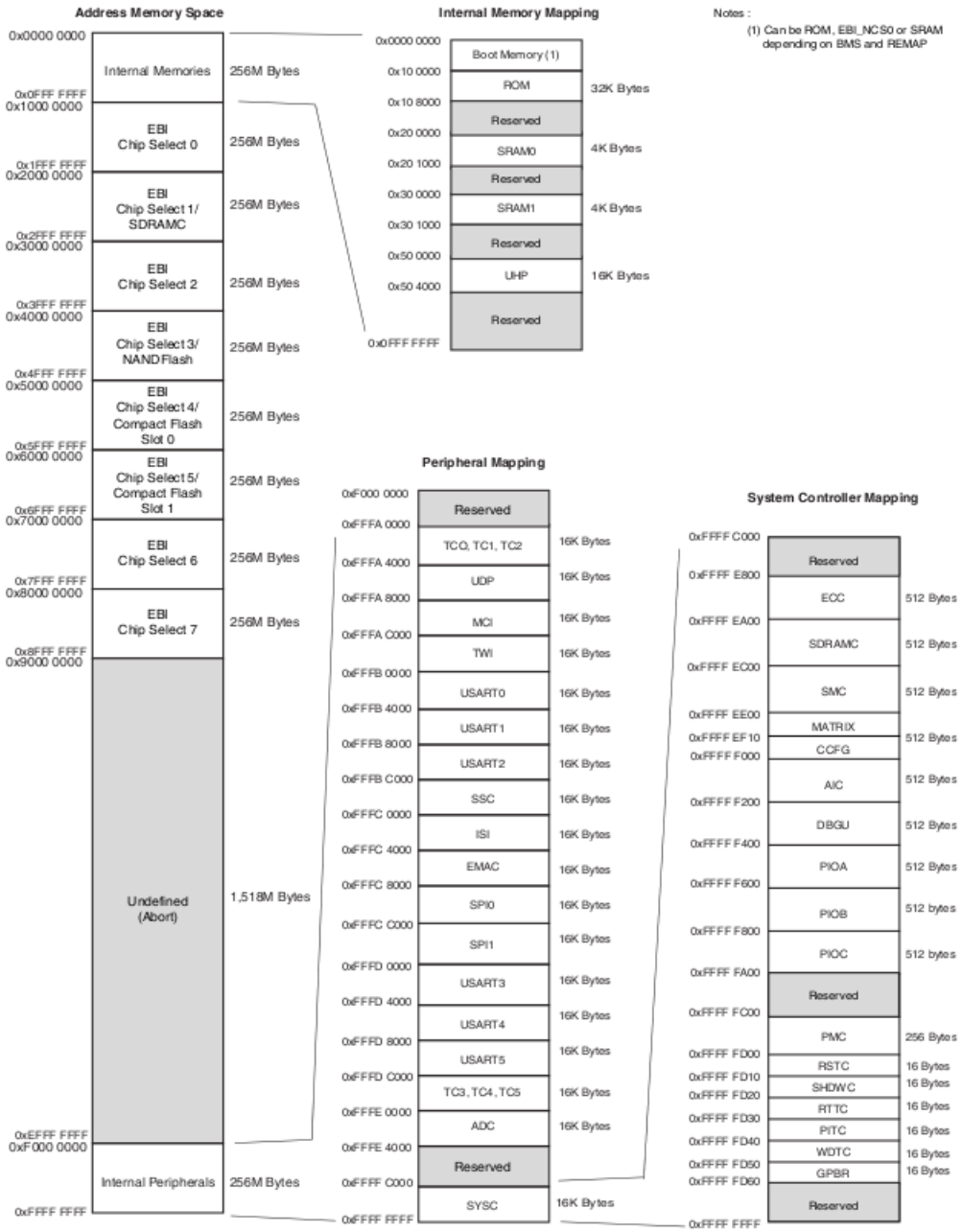


Figura 2.3: Mappatura della memoria at91sam9260

2.1.3 System Controller

Il System controller è un insieme di periferiche che consente la gestione degli elementi chiave del sistema, tra cui: power, resets, clocks time, interrupts, watchdog, etc. l'interfaccia utente del System Controller incorpora anche i registri che consentono di configurare la matrice e un insieme di registri per la configurazione dei chip. I registri di configurazione chip consentono di gestire la configurazione dell'EBI assegnando i chip select e regolando il range di tensione delle memorie esterne. Le periferiche del System Controller sono tutte mappate entro massimo 16 Kbyte dello spazio di indirizzamento, tra gli indirizzi 0xFFFFE800 e 0xFFFF FFFF (vedi figura 9). Tuttavia, tutti i registri del System Controller sono mappati nella parte superiore dello spazio di indirizzamento (0xFFFA0000 - 0xFFFE4000) . Tutti i registri del System Controller possono essere indirizzati da un unico puntatore utilizzando il set di istruzioni standard ARM, come le istruzioni load / store che dispongono di una modalità di indicizzazione di $\pm$ 4 Kbyte (offset 12 bit).

Elenchiamo adesso i vari blocchi che costituiscono il System Controller:

2.1.3.1 Reset Controller

Controlla il reset delle periferiche interne e comanda il pin NRST per il reset delle periferiche esterne.

Viene alimentato da VDDBU può lanciare i seguenti segnali:

-proc_nreset: Viene resettata la linea del processore e anche il timer watchdog.

-backup_nreset: resetta tutte le periferiche alimentate da VDDBU.

-periph_nreset: resetta tutte le periferiche incorporate.

-nrst_out: pilota il pin NRST.

2.1.3.2 Shutdown Controller

Lo shutdown controller controlla le tensioni di alimentazione VDDCORE e VDDIO, è alimentato da VDDBU, l'applicazione più comune è quella di collegare il suo input WKUP0 a un qualsiasi pulsante e il suo output SHDN sul pin input shutdown del convertitore DC/DC che provvede all'alimentazione principale del sistema. Lo shutdown può essere effettuato anche via software con la scrittura di un particolare registo (SHDW_CR).

Il wake-up può essere configurato con l'attivazione su un particolare cambiamento di livello (WKUPMODE0), esiste anche la possibilità di avere un antirimbalzo per il pin WKUP0.

Il wake-up può essere anche programmato attraverso l'RTT allarm (real-time timer).

2.1.3.3 Power Management Controller (PMC)

Il PMC è un controller che si occupa dell'ottimizzazione del consumo controllando ogni clock di sistema e delle periferiche. Esistono diversi tipi di clock nel sistema che vengono controllati dal Power Management Controller: il master clock(MCK) , il processor clock (PCK), i clock di periferica (USART, SSC, SPI,

TWI, TC, MCI, etc.) che vengono chiamati MCK nei datasheet ma sono controlalbili indipendentemente, il clock UHPCK richiesto dall'host usb e infine i pin di clock esterno PCKx.

Un esempio di impiego del controller in esame si ha durante la modalità idle del processore, in tale fase il PMC andrà a disabilitare il PCK riducendo cosi i consumi di potenza.

2.1.3.4 Watchdog Timer

Il watchdog timer si occupa della temporizzazione del watchdog. Esso è composto essenzialmente da un contatore inverso (down-counter) a 12bit . Senza scendere troppo nel dettaglio vediamone brevemente il suo funzionamento, il campo WDV del registro WDT_MR al reset del processore assume per default il valore 0xFFF tale campo si riferisce al valore iniziale caricato nel counter. Il counter ha un clock uguale a Slow Clock/128 che con 12bit a disposizione fornisce un tempo massimo di esattamente 16s (nel nostro caso $2^{12} * \left(\frac{32768}{128}\right)^{-1}$ che ha appunto dimensione di secondi ovvero Hz^{-1}). Nel normale utilizzo, se il watchdog è abilitato, l'utente scrive la flag WDRSTT, del registro WDT_CR, a 1 prima che l'underflow sia avvenuto.

Quando la flag WDRSTT va alta il contatore del watchdog viene ricaricato attraverso il registro WDT_MR e si resetta, se avviene un underflow viene asserito allora il segnale "wdt_fault" che arriva al Reset Controller, questo a condizione che WDRSTEN è settato a 1 nel registro WDT_MR. Ad underflow avvenuto la flag WDUNF del registro dello stato WDT_SR viene settata alta.

Infine è da ricordare una caratteristica interessante del watchdog cioè la possibilità di impostare una finestra temporale limite per il reset dello stesso, attraverso il campo WDD. Consideriamo ad esempio una circostanza software che resetti il watchdog (attraverso WDRSTT) ripetutamente se vogliavo evitare problemi di deadlock è possibile impostare la finestra temporale [0,WDD] all' interno della quale è possibile resettare il watchdog, se questo non avviene e si ha un reset nell'intervallo [WDV, WDD] viene asserito il segnale "wdt_fault" al Reset Controller attraverso la flag WDERR.

Dopo che "wdt_fault" viene asserito e viene letto WDT_SR, "wdt_fault" viene deasserito e di fatto azzerato il watchdog.

2.1.3.5 Periodic Interval Timer (PIT)

Il PIT si occupa della temporizzazione degli interrupt in un sistema operativo. E' progettato per avere la massima accuratezza ed efficienza.

Il periodic interval timer si compone principalmente di 2 contatori uno a 20bit (CPIV) con accuratezza superiore al microsecondo e l'altro counter a 12bit (PICNT).

Descriviamone brevemente il funzionamento: i contatori lavorano su un clock Master Clock /16 (MCK/16), CPIV conta da 0 a un valore fissato nel campo PIV del mode register PIT_MR raggiunto tale valore di overflow CPIV si resetta e incrementa il contatore a 12bit PICNT.

A questo punto il bit PITS (nel registro PIT_SR) va alto e triggera una interrupt, a condizione che la flag binaria PITIEN (nel registro PIT_MR) sia abilitata.

Una eventuale scrittura del PIV in un qualsiasi momento ovviamente non resetta i counter.

Una volta lanciata o no una interrupt la lettura del valore dei counter attraverso il registro PIT_PIVR resetta il counter PICNT e cancella il PITS l'eventuale interrupt allora viene riconosciuta e il valore letto di PICNT indicherà il tempo trascorso dall'ultima lettura di PIT_PIVR.

Esiste anche un registro per la lettura che non genera eventuali reset (PIT_PIIR) utile ad esempio ad un profiler. La figura 10 mostra lo schema a blocchi dell'implementazione dei registri appena descritta.

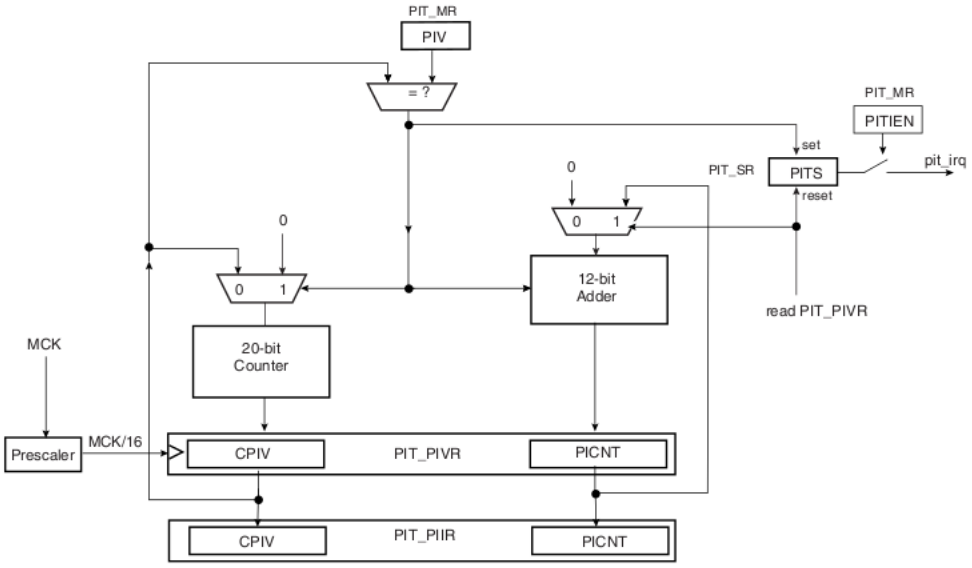


Figura 2.4: Schema a blocchi del Periodic Interval Timer (PIT)

2.1.3.6 Real-time Timer

Il real-time timer è una parte fondamentale del system controller esso è composto da 2 blocchi un divisore di clock e un counter. Il divisore di clock è programmabile attraverso un valore a 16 bit che ha come valori utili un range da 0x00000003 - 0x00008000 e si occupa di dividere lo slow clock (nel nostro caso a 32768 Hz). Ad esempio un valore del registro RTPRES (registro del divisore) settato a 8000 ci darà un'uscita un clock cn periodo 1s ovvero 1 Hz. Il blocco counter è a 32bit ed è quindi in grado di contare fino a 2^32 s (circa 136 anni) l'accuratezza massima si ottiene ovviamente con il valore 3 valori più bassi sono settabili,ma per una questione architetturale e di gestione intrinseca del controller non sono utilizzabili. Inoltre è da ricordare il sistema per generare un RTT allarm che può gestire il wake-up del controller shutdown.

2.1.3.7 Advanced Interrupt Controller

L'advanced interrupt controller o AIC è il controller che si occupa della configurazione e controllo delle linee interrupt del processore ARM. L'approccio utilizzato è quello della vettorizzazione degli interrupt.

Abbiamo 8 livelli di priorità per 32 possibili sorgenti di interrupt vettorizzate ogni vettore da 32 bit, oltre alle sorgenti IRQ troviamo anche la possibilità di gestire le interrupt come FIQ (fast interrupt).

Una caratteristica avanzata si ha nella gestione delle priorità, infatti il microcontrollore è in grado di sospendere un' interrupt con un livello di priorità

più basso per servire un'interrupt con livello di priorità più elevato. L'AIC comunica con il processore ARM attraverso interrupt di tipo IRQ e FIQ.

Distinguiamo le periferiche in due tipi: esterne e incorporate nel microcontrollore.

Le periferiche incorporate (Embedded) sono direttamente connesse all'AIC mentre le sorgenti di interrupt esterne sono connesse all'AIC attraverso il PIO controller questo viene identificato come una periferica Embedded ed ha dunque il suo IRQ connesso all'AIC. La sorgente 0 di interrupt è quella relativa alle fast interrupt che di fatto evitano il controller di priorità. La sorgente 1 si occupa della gestione interrupt del System Controller il resto delle sorgenti interrupt 1-31 sono dedicate o alle uscite interrupt delle periferiche utente o a interrupt esterne.

2.1.3.8 Debug Unit

L'unità di debug è caratterizzata da una uart a 2 poli, ed è un singolo punto di ingresso per ogni funzionalità del microprocessore Atmel. I due poli uart possono essere utilizzati anche come una seriale (rs232 a 2 fili) per scopi generici. La debug unit inoltre comunica con l'ICE (in-circuit emulator) del processore ARM ed attraverso il gestore DCC (debug comunication controller) offre funzionalità di debug con possibilità di interrupt.

La debug unit nel corso del progetto verrà utilizzata principalmente con funzionalità di seriale rs232 di debug con baudrate impostato a 115200.

2.1.4 Parallel Input/Output Controller (PIO)

Il PIO controller si occupa della gestione dei pin di I/O. Ogni PIO controller può gestire fino a 32 I/O ogni I/O è configurabile attraverso opportuni registri come I/O general purpose, I/O periferica A e I/O periferica B.

Si tratta dunque di una gestione multiplexata degli I/O, inoltre per quanto riguarda le interrupt abbiamo visto come il PIO controller sia associato a una sorgente di interrupt. Il PIO controller può disporre anche una serie di interrupt esterni. Nel microcontrollore abbiamo 3 PIO controller rispettivamente PIOA PIOB PIOC ciascuno dispone di 32 I/O. Vedremo come verranno sfruttati all' interno del progetto alcune di queste linea di I/O come general purpose per gestire led, pulsanti, relé ecc...

Sottolineo che le uscite del microcontrollore possono pilotare al massimo 16 mA ad eccezione dei pin da PC4 a PC31.

A titolo informativo riporto in tabella 4, estratta dal datasheet del microcontrollore, l'associazione tra ogni pin e la sua possibile funzionalità intesa come periferica A o B per il PIOC.

Tabella 2.3: Funzionalità multiplexate del Parallel I/O controller C

		PIO Controller C
I/O Line	Peripheral A	Peripheral B
PC0		SCK3
PC1		PCK0
PC2[1]		PCK1
PC3[1]		SPI1_NPCS3
PC4	A23	SPI1_NPCS2
PC5	A24	SPI1_NPCS1
PC6	TIOB2	CFCE1
PC7	TIOB1	CFCE2
PC8	NCS4/CFCS0	RTS3
PC9	NCS5/CFCS1	TIOB0
PC10	A25/CFRNW	CTS3
PC11	NCS2	SPI0_NPCS1
PC12[1]	IRQ0	NCS7
PC13	FIQ	NCS6
PC14	NCS3/NANDCS	IRQ2
PC15	NWAIT	IRQ1
PC16	D16	SPI0_NPCS2
PC17	D17	SPI0_NPCS3
PC18	D18	SPI1_NPCS1
PC19	D19	SPI1_NPCS2
PC20	D20	SPI1_NPCS3
PC21	D21	EF100
PC22	D22	TCLK5
PC23	D23	
PC24	D24	
PC25	D25	
PC26	D26	
PC27	D27	
PC28	D28	
PC29	D29	
PC30	D30	
PC31	D31	

2.2 Architettura ARM

Il microcontrollore Atmel è basato su processore ARM926EJ-S. La serie ARM9 condivide molte delle scelte progettuali specifiche delle architetture ARM.

Vediamo sinteticamente la struttura del processore ARM e le sue caratteristiche salienti.

L' architettura ARM è progettata per ottimizzare la dimensione, tuttavia conservando un' implementazione ad alte prestazioni. La semplicità dell'architettura conduce ad un risparmio oltre che in termini di dimensione anche in termini di consumo.

La filosofia costruttiva di base su cui poggia l'architettura ARM è di tipo RISC Reduced Instruction Set Computer.

Le tipiche caratteristiche RISC adottate nei processori sono:

- Un file register ampio e uniforme .
- Un'architettura load/store, in cui le operazioni di trattamento dei dati operano solo sul contenuto dei registri e non direttamente sul contenuto della memoria.
- Un modello semplice di indirizzamento, con gli indirizzi determinati esclusivamente dal contenuto di registri o da campi di istruzione.
- Un set di istruzioni uniforme di lunghezza fissata per semplificare le operazioni di decodifica.

Inoltre a tali caratteristiche di base l'architettura ARM affianca:

- Un controllo sia su l' ALU (Arithmetic Logic Unit) che sullo shifter in ogni istruzione di trattamento dati per massimizzare l'uso congiunto di ALU e shifter.
- Modalità di auto-incremento e auto-decremento di indirizzamento per ottimizzare i loops nei programmi.
- Load e Store multipli per massimizzare il throughput dei dati.
- conditional execution di tutte le istruzioni per massimizzare il throughput di esecuzione.

Questi arricchimenti all' architettura RISC tradizionale consentono ai processori ARM di ottenere un ottimo bilanciamento tra i seguenti quattro parametri di valutazione:

- high performance
- low code size
- low power consumption
- low silicon area

2.2.1 Registri

I processori ARM hanno 31 registri general-purpose a 32 bit. Tuttavia 16 di questi registri sono effettivamente visibili gli altri sono utilizzati per ottimizzare le operazioni di gestione delle eccezioni.

Il banco principale da 16 registri è usato da tutto il codice unprivileged (modalità di accesso del processore). Questi sono i registri visibili in user mode. La modalità User Mode è diversa da tutte le altre in quanto è non privilegiata, il che significa che in User Mode non si può passare a un' altra modalità di processore senza generare un' eccezione di memoria del sistema.

Inoltre l'accesso alla memoria e alcune delle funzionalità che il coprocessore offre nelle modalità privilegiate, potrebbero essere limitate o non consentite.

Due dei 16 registri visibili hanno un ruolo importante e specifico:

- Link registers(LR)
- Program counter(PC)

Il registro LR è il 14 e conserva l'indirizzo dell' istruzione successiva dopo che viene effettuata un' istruzione di Branch (BL), R14 potrebbe in linea teorica anche essere utilizzato come registro general purpose.

Il registro PC è il numero 15 (R15) esso viene utilizzato come puntatore alla seconda istruzione successiva all' istruzione corrente(questo perchè parliamo di un microprocessore a pipeline 3/5 stage) . Tutte le istruzioni ARM sono costituite da 4 bytes (32 bit) e sono sempre allineate con il confine della word. Questo significa che i due bit inferiori del PC sono sempre a zero, e inoltre il PC conterrà solo 30 bits non costanti. I rimanenti 14 registri non hanno funzioni particolari e il loro utilizzo è puramente deciso dal software. Normalmente si utilizza R13 come Stack Pointer (SP).

2.2.2 Eccezioni

L' architettura ARM supporta cinque diversi tipi di eccezioni e per ognuna di esse una determinata modalità privilegiata.

Le eccezioni ARM sono:

- fast interrupt
- normal interrupt
- memory aborts, che è utilizzata solitamente per implementare la memoria virtuale o la protezione di accesso alla memoria.
- attempted execution of an undefined instruction
- software interrupt (SWI), che può essere usata per costruire le chiamate del sistema operativo.

In generale le modalità possibili per la CPU ARM sono: User (usr), FIQ (fiq), IRQ (irq), Supervisor (svc), Abort (abt), Undefined (und) e System (sys).

Quando avviene un' eccezione alcuni registri standard sono rimpiazzati con altri registri fisici specifici (banked registers) della modalità di eccezione corrispondente. Tutte le modalità delle eccezioni hanno banked register di R13 e

R14. Tuttavia la modalità fast interrupt ha più registri banked (solitamente dal R8 al R14) questo per velocizzare il più possibile il processo di interrupt (context switch).

Di seguito la figura 11 riassume quanto detto fino ad adesso riguardo le eccezioni:

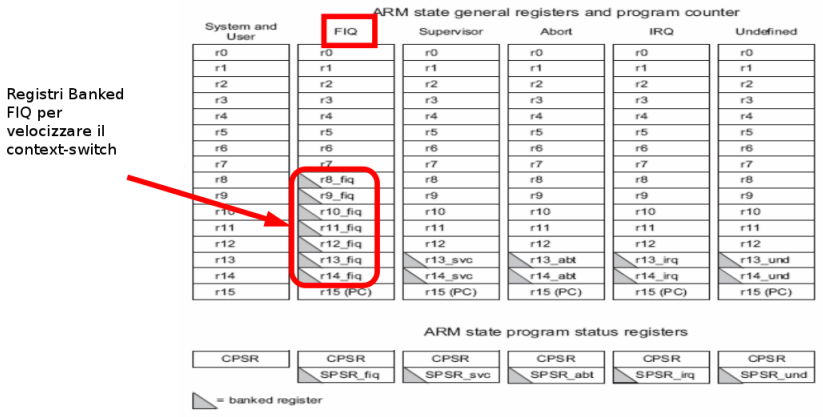


Figura 2.5: Schema registri per le varie modalità di un processore ARM

Ad ogni eccezione a livello software è associato un exception handler. Quando viene richiesto un certo exception handler il registo R14 mantiene l'indirizzo di ritorno e passa ad eseguire il codice dell'eccezione. R13 (banked) viene riservato per lo stack pointer della modalità processore a cui si è arrivati dopo un eventuale context switch. Quando avviene una eccezione il processore ARM interrompe l'esecuzione dell' istruzione corrente e inizia a eseguire un numero prefissato di indirizzi di memoria, ovvero i vettori delle eccezioni (exception vectors). Per ogni tipo di eccezione esiste un vettore delle eccezioni separato. In fase di inizializzazione un sistema operativo installerà dunque i suoi handler exceptions negli indirizzi definiti nei vettori. Dunque riassumendo avremo sette modalità del processore possibili cinque delle quali sono riservate alle eccezioni le altre due sono invece User e System mode dove System è una modalità privilegiata. A questo punto è bene fare una considerazione molto importante, supponiamo che venga lanciata un' eccezione, ad esempio una IRQ, il processore ARM allora bloccherà la sua esecuzione corrente salverà il PC all' interno del LR della modalità IRQ (è un registro banked) e salverà anche CPSR in SPSR infine salterà a processare una serie di indirizzi di memoria (in realtà si utilizza un offset su LR che dipende dal tipo di interrupt) contenuti nel vettore delle interrupt IRQ, qui troverà un certo handler che eseguirà determinate operazioni eventualmente installate dal sistema operativo, ma cosa accade se durante l'esecuzione dell'handler avviene un'altra interrupt? E se viene chiamata una subroutine con relativo salto e sovrascrittura del LR come si può ripristinare l' indirizzo di partenza del primo salto del PC?

La risposta a tali domande è affrontata e risolta in modo sistematico solo nelle versioni ARM v4 e superiori. Nelle eventualità descritte nelle versioni precedenti alla v4 era necessario o evitare eccezioni e subroutine quando si gestiva un handler oppure fare uno switch context per eseguire l'hadler in User Mode.

Mentre la prima soluzione è chiaramente troppo restrittiva la seconda appare percorribile, ma ha un grave problema l'esecuzione in User mode dell'handler impedisce l'accesso ad alcune funzionalità / registri che richiedono una modalità privilegiata. La soluzione dalle versioni v4 in avanti è stata dunque quella di introdurre la modalità System privilegiata con tutti i registri in share con la modalità User. In questo modo gli handler posso essere (infatti è pratica consolidata) eseguiti in System mode così che si possano gestire eventuali salti a subroutine. Prima di passare alla descrizione dell' istruction set chiariamo la struttura dello status register.

Lo status register (CPSR) ovvero il registo di stato è il medesimo per tutte le modalità. Esso è strutturato nel seguente modo:

- 4 flag di condizione o condition code flags (Negative, Zero, Carry and oVerflow)
- 2 bit per disabilitare le interrupt, una per ogni tipo di interrupt (FIQ , IRQ)
- 5 bit che codificano la modalità del processore
- 1 bit che identifica se le istruzioni che si stanno eseguento sono ARM o Thumb.

2.2.3 Istruction Set

L' istruction Set ARM può essenzialmente essere suddivisa in sei grandi classi di istruzioni:

- ***Branch instructions***
- ***Data-processing instructions***
- ***Status register transfer instructions***
- ***Load and store instructions***
- ***Coprocessor instructions***
- ***Exception-generating instructions***

Molte delle istruzioni Data-processing e un tipo di istruzioni del Coprocessore (Coprocessor instructions) possono alterare le quattro flags del registro di stato CPSR in base al loro risultato.

Quasi tutte le istruzioni ARM contengono un campo di 4-bit di condizione. Una delle possibili configurazioni dei 4-bit specifica la condizione di istruzione non condizionata.

Gli altri 14 possibili valori del campo sono utilizzati invece per l'esecuzione condizionata, in altre parole se la flags della condizione corrispondente indica true quando l'istruzione viene processata verra eseguita normalmente, in caso contrario l'istruzione non verrà eseguita. Le condizioni possibili del campo sono:

- tests di uguaglianza o non-uguaglianza
- tests per $<$, $<=$, $>$, e $>=$, considerando i casi signed e unsigned

- Ogni altra condition code flag da testare specificatamente.

Infine la sedicesima combinazione del campo a 4-bit di condizione è usata per alcune istruzioni che non prevedono esecuzione condizionata.

2.2.3.1 Branch instructions

Le istruzioni di salto (Branch) oltre a permettere alle istruzioni data-processing e Load-Store di cambiare il flusso di esecuzione scrivendo il PC , consentono un Branch standard che usa 24 bit di offset per saltare in avanti e indietro fino a 32MB . Esiste un tipo di Branch (BL) Branch e Link che conserva anche l'indirizzo della istruzione in R14(LR) prima di eseguire il salto. Ciò fornisce un modo automatico a una chiamata a funzione di tornale al flusso di esecuzione principale ripristinando il LR nel PC. Infine esistono anche Branch che possono cambiare il set di istruzioni, ovvero dopo il salto interpretano le istruzioni come Thumb o ARM. In questo modo è possibile richiamare codice Thumb all' interno del codice ARM e viceversa.

2.2.3.2 Data-processing instructions

Le istruzioni di data-processing eseguono essenzialmente calcoli sui registri general-purpose.

Sono quattro i tipi di istruzioni data-processing:

- Arithmetic/logic instructions
- Comparison instructions
- Multiply instructions
- Count Leading Zeros instruction

Arithmetic/logic instructions

Esistono 12 istruzioni aritmetico logiche, tutte condividono un formato comune a livello di struttura.

Le istruzioni aritmetico logiche operano sempre su un massimo di due operandi e scrivono il risultato ottenuto in un registro di destinazione. Inoltre in base al risultato dell' istruzione possono anche aggiornare le flags del registro di stato CPSR.

Dei 2 operandi sorgente uno è sempre un registro l'altro invece può avere essenzialmente due forme o è un valore immediato o un valore di un registro (eventualmente shiftato).

Nel caso in cui l' operando risultasse un registro shiftato il valore dell' offset può essere un valore immediato o un valore contenuto in un ulteriore registro. Sono in tutto quattro i tipi di shift che possono essere specificati. E' interessante notare come ogni istruzione aritmetico logica ha la possibilità di eseguire sia l'operazione corrispondente sia un' operazione di spostamento. Come risultato di quanto detto, l'ARM non ha istruzioni di shift dedicate. Infine poichè Program Counter (PC) è un registro general-purpose, le istruzioni aritmetico logiche possono scrivere i loro risultati direttamente all' interno di esso. Questo permette una facile implementazione di una serie di istruzioni di salto.

Comparison instructions

Esistono quattro istruzioni di comparazione che usano il medesimo formato come nel caso delle istruzioni aritmetico logiche. Operano logicamente su due operandi sorgente, ma non scrivono il risultato in un registro. Esse infatti aggiornano sempre le flags del registro CPSR come risultato della loro esecuzione.

Per quanto riguarda i due operandi essi hanno la stessa struttura descritta precedentemente nelle istruzioni aritmetico logiche dunque "ereditano" la possibilità di incorporare operazioni di shift.

Multiply instructions

Le istruzioni di moltiplicazione possono essere di due tipi. Entrambi i tipi moltiplicano valori di registri ovviamente a 32 bit, ma mentre un tipo pone il risultato ottenuto a 32bit in un registro , l'altro tipo pone il risultato a 64 bit in 2 registri distinti. Per i due tipi di istruzione di moltiplicazione è definita l'operazione di accumulate (a = a + (b*c)).

Count Leading Zeros instruction

L'istruzione di conteggio degli zeri in posizione più significativa determinano appunto il numero di bit a 0 nelle posizioni piu significative fino al primo 1. Il valore del conteggio viene scritto appunto nel registro CLZ.

2.2.3.3 Status register transfer instructions

Le istruzioni di trasferimento dello status register si occupano dei trasferimenti del contenuto dei registri CPSR o SPSR verso o dai registri general-purpose.

La scrittura sul registro CPSR può eseguire le seguenti operazioni:

- settare il valore di una condition flags
- settare il valore di un bit di abilitazione interrupt
- settare la modalità del processore(processor mode)

2.2.3.4 Load and store instructions

Le istruzioni di load/store supportate sono:

- Load and Store Register
- Load and Store Multiple register
- Swap register and memory contents

Load and Store Register Le istruzioni di load Register possono caricare dalla memoria ad un registro nei formati 32bit (word), 16bit (halfword) e 8bit (byte). Il load di Byte e halfword può essere automaticamente zero-extended o sign-extended (in base alla rappresentazione numerica). Cosi come le istruzioni di load anche le istruzioni di store permettono di salvare il contenuto di un registro nella memoria, nei formati a 32bit(word), 16bit (halfword) e 8bit (byte).

Le istruzioni Load/Store Register hanno tre modi di indirizzamento primari, che utilizzano tutti un registro base e un offset specificato all' interno dell'istruzione.

Le modalità di indirizzamento sono:

• *offset addressing*

Quando l'indirizzo di memoria viene calcolato a partire dal valore base di un registro sommato o sottratto a un certo offset.

• *pre-indexed addressing*

Quando l'indirizzo di memoria viene calcolato come nell' offset addressing, ma oltre all' operazione di Load/Store viene scritto anche il registro base che assumerà il valore dell' indirizzo di memoria a cui si è fatto riferimento.

• *post-indexed addressing*

Quando l'indirizzo di memoria viene calcolato direttamente dal valore del registro base. E' da notare che anche in questo caso abbiamo la scrittura del registro base che assumerà un valore uguale al valore precedente all'operazione sommato o sottratto ad un offset.

Di fatto dunque abbiamo tre modi di operare che mantengono una certa uniformità di istruzione, esiste sempre un registro base è un offset, tuttavia solo nel primo e nel secondo caso di indirizzamento(*offset addressing, pre-indexed addressing)* l'offset influenza la locazione di memoria scelta, viceversa nel caso di *post-indexed addressing* abbiamo un offset che si "rende disponibile" solo dopo l'operazione di indirizzamento. Nonostante sia poco intuitivo, la possibilità di avere disponibile un indirizzo base (valore registro) con offset è utilissimo poichè ricordando che il PC è un registro general-purpose a 32bit si potranno effettuare operazioni di salto nello spazio di 4GB di memoria, attraverso l'uso composito delle istruzioni appena descritte.

Load and Store Multiple registers Le istruzioni Load and Store Multiple registers ovvero rispettivamente istruzioni LDM e STM operano un trasferimento di un blocco costituito da un qualsiasi numero di registri general-purpose alla memoria o viceversa. Sono quattro le modalità di indirizzamento previste per queste istruzioni:

- pre-increment
- post-increment
- pre-decrement
- post-decrement

L'indirizzo base di memoria è specificato tramite il valore del registro base, che può essere aggiornato prima o dopo l'operazione di trasferimento. Appare chiara l'utilità di tali istruzioni nel caso delle subroutine che sfruttano in modo intensivo lo stack, in quanto le operazioni di trasferimento in ingresso o in uscita dalla

subroutine possono essere implementate in modo molto veloce ed efficiente. Ad esempio una singola istruzione STM all'ingresso di una subroutine può eseguire il push del contenuto dei registri sullo stack e aggiornare lo stack pointer con il nuovo indice raggiunto. Similmente all'uscita di una subroutine con una singola istruzione LDM sarà possibile fare un load dallo stack sui registri, caricare anche il PC con il valore di ritorno e aggiornare lo stack pointer.

Per le funzionalità appena descritte le istruzioni LDM e STM rappresentano un ottimo strumento negli algoritmi ove sia necessario un trasferimento in blocco di dati.

Swap register and memory contents Un' istruzione swap (SWP) esegue la seguente sequenza di operazioni:

- Carica un valore da una posizione di memoria in un registro specificato che chiamiamo A.
- Memorizza il contenuto di un registro B nella stessa posizione di memoria.
- Scrive il valore caricato nel registro A nel registro B.

In coclusione scambia appunto il contenuto di memoria e registro appoggiandosi ad un ulteriore registro temporaneo.

L'operazione di swap è atomica e viene utilizzata per effettuare l'aggiornamento dei semafori .

2.2.3.5 Coprocessor instructions

Esistono tre tipi di istruzioni del coprocessore:

- Data-processing instructions: Che si occupa dell'avvio di operazioni interne specifiche del coprocessore.
- Data transfer instructions: Che trasferiscono dati dalla memoria al coprocessore e viceversa (gli indirizzi di trasferimento sono calcolati dal processore ARM).
- Register transfer instructions: Che si occupano di trasferire i valori contenuti nei registri del Processore al coprocessore e viceversa.

2.2.3.6 Exception-generating instructions

Sono due i tipi di istruzioni capaci di generare un' eccezione:

- Software interrupt instructions SWI

- Software breakpoint instructions BKPT

Le SWI come abbiamo già avuto modo di specificare sono utilizzate per le system call del sistema operativo e di fatto sono l'unico modo (almeno in un sistema operativo classico) che permette ad un task in modalità non privilegiata di avere accesso a una modalità privilegiata.
Le BKPT invece generano un abort exception e solitamente vengono utilizzate per implementare dei breakpoint per funzionatità di debug.

2.3 Scelte di progetto

Dopo aver discusso le caratteristiche hardware del microcontrollore utilizzato, è importante chiarire i fini del progetto per capire alcune scelte che sono state fatte. L'esigenza fondamentale è quella di ottenere un dispositivo versatile in grado comunicare con altre periferiche attraverso porte seriali e eventualmente attraverso internet come web server o attraverso collegamento con protocollo sicuro ssh. Il tutto senza avere esigenze molto spinte in termini di gestione hard real time. Nel momento in cui si decide di progettare un dispositivo che abbia un sistema operativo sono diverse le scelte possibili: si può progettare un sistema operativo totalmente custom, utilizzare un sistema operativo opensource oppure acquistare un sistema operativo proprietario. Nel nostro caso si è deciso di scegliere un sistema operativo opensource opportunamente modificato secondo le esigenze.

Alcuni sistemi basati su sistemi operativi real time sono in grado di mettere in comunucazione il layer applicativo direttamente con l' AIC, tali sistemi sono utili per progetti che mirano ad ottenere una gestione real time molto accurata. Nel nostro caso come gia anticipato non avendo tali esigenze si è deciso di utilizzare un sistema basato sul kernel Linux (2.6.35.7) e per quanto riguarda il sistema operativo e filesystem di appoggiarsi sul progetto Openwrt. Avere un sistema unix-like a disposizione permette di avere un maggior grado di astrazione dall'hardware, per quanto riguarda la programmazione inoltre da l'opportunità di sfruttare le librerie standard C molto evolute messe a disposizione dagli standard BSD e POSIX. Lo svantaggio probalibilmente di una tale implementazione è rappresentato più dal tempo di acquisizione di un certo "Know How", poichè spesso la realizzazione pratica di un tale progetto richiede competenze multi disciplinari su vari ambiti informatici ed elettronici.

I punti di cui mi sono occupato nella mia esperienza di stage sono principalmente la programmazione a basso livello del dispositivo e la realizzazione di un spazio utente adeguato.

Spesso ho affrontato alcuni problemi di tipo implementativo in cui sono state effettuate delle scelte sempre considerando le specifiche richieste, tuttavia in generale in alcune circostanze la libertà implementativa è stata quasi sbalorditiva.

Iniziamo con un' introduzione doverosa attraverso un approccio dall'alto verso il basso (top-down) a esaminare la struttura software del dispositivo. Un

sistema operativo basato su Linux si compone essenzialmente di due componenti fondamentali un filesystem, inteso come gerarchia strutturale delle directory, ed un kernel. Il filesystem si occupa essenzialmente di dare all'utente un ambiente di lavoro, per essere piu chiari è colui che mette a disposizione tutti gli strumenti applicativi, l'utente solitamente tende ad identificare con esso il sistema operativo.

Prima di avviare un certo filesystem (montare), abbiamo bisogno di un kernel ovvero un insieme di istruzioni in grado di comunicare direttamente con l'hardware del sistema che mette a disposizione certe risorse e ne protegge altre. Per avviare un kernel però abbiamo bisogno di un altro layer software che a sua volta inizializzerà solo alcune parti del sistema per avere alcune funzionalità minimali che permettano l'avvio di un kernel, questo software viene chiamato boot loader, nel nostro caso erano due le scelte possibili Redboot e U-Boot la scelta è caduta sul secondo, più per una coerenza con la scelta fatta per il sistema operativo basato su Openwrt.

Normalmente si considera questo il livello software più basso in realtà vedremo che il sistema Embedded in esame ha ancora due layer di inizializzazione ovvero un bootloader di secondo e terzo livello, rispettivamente AT91bootstrap e ROMBOOT.

In questo caso la scelta è vincolata dal produttore del microcontrollore, per quanto riguarda il bootloader di secondo livello è stato possibile un certo margine di customizzazione, mentre per il bootloader di terzo livello (ROMBOOT) non è stato possibile effettuare alcuna modifica poichè closed-source.

Atmel tuttavia pur non rilasciando il codice sorgente del ROMBOOT, offre un'ampia documentazione a riguardo e fornisce gli strumeti necessari per implementare un proprio bootloader di terzo livello eventualmente su una memoria esterna (Data-Flash, NAND-Flash).

Capitolo 3

Inizializzazione del sistema Embedded

Nei primi due capitoli abbiamo affrontato dei temi particolarmente interessanti legati tuttavia più all'aspetto teorico dell'argomento della tesi. Nel seguente capitolo vedremo più precisamente la struttura specifica del dispositivo basato su at91sam9260, su cui ho lavorato durante il tirocinio. Saranno evidenziati gli aspetti pratici per gestire un progetto Linux basato su microcontrollore Atmel AT91. Nei paragrafi successivi verrà dunque mostrato tutto il software di basso livello necessario per inizializzare il microprocessore e le sue periferiche incorporate.

3.1 ROMBOOT

Per ROMBOOT si intende il firmware contenuto all'interno della memoria ROM del microcontrollore. La ROM presente sull'at91sam9260 ha una dimensione di 32KB e la sua mappatura in memoria va dall'indirizzo 0x100000 all'indirizzo 0x108000 come mostrato dalla figura 9.

Il codice sorgente del firmware presente nella ROM non è rilasciato dal produttore del microcontrollore, tuttavia le sue operazioni sono ben documentate all'interno delle Application Notes Atmel. Il microcontrollore quando viene alimentato effettua una serie di operazioni che prendono il nome di sequenza di boot.

La sequenza di boot può seguire essenzialmente due flussi di operazioni. La prima scelta che viene fatta in fase di boot è dipendente dall'impostazione hardware di un pin del microcontrollore detto BMS (Boot Manager Select).

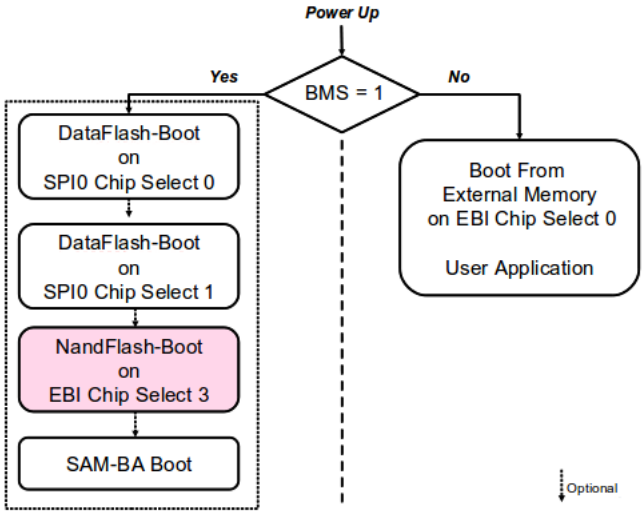


Figura 3.1: Sequenza di boot del ROMBOOT

Come si vede in figura 12 in base al valore logico dell'ingresso BMS si hanno le due diverse modalità di boot.

Se il BMS viene forzato al valore logico alto il boot avviene da ROM (BMS=1). Questo significa che la prima istruzione che il processore andrà a valutare sarà quella presente nell'indirizzo 0x0 della ROM di sistema.

L'avvio attraverso il codice contenuto nella ROM, programmata direttamente da Atmel, effettua una serie di operazioni che consentono essenzialmente di inizializzare un supporto NVM (ad esempio una memory Flash sul bus esterno) per il caricamento di un secondo e infine terzo bootloader che potrà lanciare effettivamente il kernel Linux. La scelta nel sistema Embedded che andiamo ad analizzare è stata proprio quella di impostare il BMS al valore logico 1 e scegliere come dispositivo NVM (Non-Volatile-Memory) la NAND-Flash da 8Gb ovvero 1GB collegata all' EBI Chip Select 3. La NAND-Flash utilizzata per quanto detto dovrà necessariamente contenere al suo interno il bootloader di secondo e terzo livello, rispettivamente AT91bootstrap e U-Boot. Andando ad analizzare nel dettaglio lo start-up del microcontrollore si nota che la prima istruzione ad essere processata è sempre quella presente all'indirizzo 0x0.

La scelta del supporto fisico non volatile da cui fare il boot sembrerebbe per quanto detto vincolata alla mappatura dell'indirizzo 0x0, tuttavia questo vincolo non esiste, poichè il comando di remap può rimappare in posizione 0x0 il primo indirizzo delle memorie: ROM, NOR che può essere collegata all'EBI_NCS0 e SDRAM. L'utilità di poter effettuare un boot da un supporto non volatile diverso dalla ROM interna risiede nella possibilità di poter configurare in modo del tutto personalizzato il vettore delle eccezioni del microcontrollore e in generale anche la sua inizializzazione periferica. Una personalizzazione a così basso livello può essere adottata più per soluzioni non basate su Linux, ad esempio tutte quelle basate su singola applicazione.

La sequenza di boot mostrata in figura 12 fa riferimento alle due alternative possibili per il boot.

1. BMS=1
   In questo caso si considera in posizione 0x0 la ROM interna. Il processore procede alla ricerca di un vettore delle eccezioni valido nelle memorie NVM con la priorità mostrata in figura.

2. BMS=0
   In questo caso il boot considera nella posizione 0x0 la memoria collegata al EBI_NCS0 qui duque dovrà essere scritto un firmware custom per inizializzare il dispositivo secondo le esigenze specifiche.

Il ROMBOOT messo a disposizione da Atmel, come mostrato in figura 12, ha come ultima strategia di boot il SAM-BA boot questo tipo particolare di boot viene utilizzato principalmente per la programmazione del dispositivo.

Il SAM-BA boot è un sistema che permette l'interfacciamento del microcontrollore con un PC tramite USB o seriale DBGU. SAM-BA boot effettua preliminarmente alcuni controlli con i quali stabilisce il tipo di interfacciamento valido disponibile.

Lo schema proposto in figura 13 mostra l'effettiva sequenza di controllo che viene effettuata, notiamo che partendo dall'enumerazione del dispositivo USB si hanno due possibilità:

- Microcontrollore riconosciuto enumerazione eseguita con successo
- Microcontrollore non riconosciuto verifica connessione da seriale RS232

Se non viene riconosciuto nessun collegamento valido il microcontrollore rimane in attesa di una possibile enumerazione via USB.

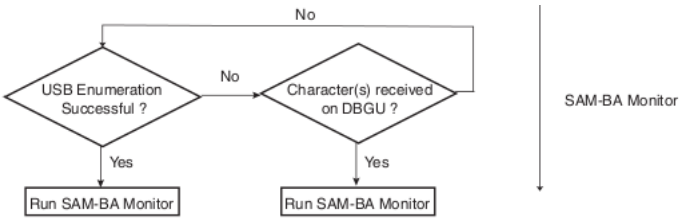


Figura 3.2: Sequenza di controllo SAM-BA monitor

Dunque la priorità nel riconoscimento di un interfacciamento valido viene data al sistema di enumerazione USB device del dispositivo solo successivamente verrà infatti eseguito un controllo sulla seriale DBGU a due fili.

Le operazioni descritte sono quelle effettuate dal microcontrollore nel così detto SAM-BA monitor. Lato host ovvero dal computer che dovrà interfacciarsi con il microcontrollore, verrà utilizzato un tool sviluppato da Atmel capace di collegarsi al sistema SAM-BA, munito di interfaccia grafica e in grado di effetturare una serie di operazioni sul dispositivo.

Le principali operazioni possibili sono:

- Gestione delle memorie presenti sul bus esterno con possibilità di scrittura e lettura.
- Gestione della memoria SDRAM con possibilità di scrittura e lettura.

Le operazioni che in realtà possono essere effettuate sul dispositivo sono molteplici poichè il sistema SAM-BA funziona attraverso l'esecuzione di immagini binarie. Un uso avanzato del tool SAM-BA permette inoltre di creare proprie applicazioni che possono essere eventualmente utilizzate tramite scripts in TCL.

Nello specifico il ruolo del SAM-BA boot, nel sistema Embedded Linux, è quello di effettuare operazioni sulla memoria NAND-Flash. Il SAM-BA boot è essenziale nella programmazione del dispositivo è infatti attraverso questo meccanismo appena descritto che sarà possibile scrivere nella NAND il bootloader di secondo livello (AT91bootstrap) e quello di terzo livello (U-Boot).

Vediamo di seguito come funziona il tool messo a disposizione da Atmel per la programmazione via SAM-BA.

Come detto la programmazione può avvenire attraverso la seriale DBGU o tramite USB device ovviamente risulta notevolmente più pratico e veloce l'impiego di un collegamento con cavo USB. Una volta installato il tool SAM-BA abbiamo a disposizione un'interfaccia grafica che ci permette di collocare in un certo indirizzo in RAM del dispositivo Embedded eventuali file binari da eseguire. Tra questi file binari ne esistono alcuni già forniti da Atmel, che sostanzialmente sono in grado di inizializzare e gestire la memoria NAND (o almeno quelle supportate).

SAM-BA inoltre è in grado di eseguire scripts TCL, questa caratteristica semplifica notevolmente la creazione di sequenze di programmazione custom.

Il TCL è un particolare linguaggio di programmazione scripting creato da John Ousterhout, molto intuitivo ma anche molto potente in grado di dialogare con oggetti C, C++ REXX e Java.

Attraverso la scrittura di un programma in TCL è possibile sfruttare alcune funzioni base presenti nel tool SAM-BA ed è proprio tramite questa strategia che è stata effettuata la programmazione a basso livello del dispositivo Embedded Linux oggetto della tesi.

La figura 14 mostra l'interfaccia utente del tool di programmazione SAM-BA si notano chiaramente le varie sezioni riguardanti i possibili supporti NVM ad esempio DataFlash e NandFlash.

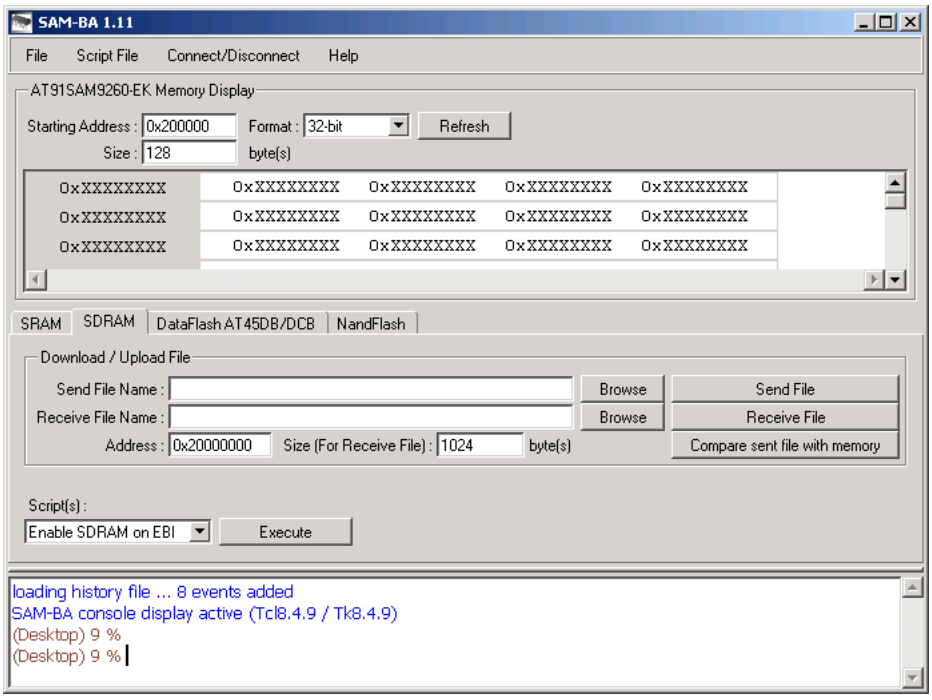


Figura 3.3: Interfaccia grafica di SAM-BA

L'algoritmo 2 mostra invece un esempio di script TCL per la programmazione del dispositivo basato su AT91.

Lo script può essere eseguito attraverso l'interfaccia utente o in modo più immediato utilizzando uno script in bat che avvii l'applicazione SAM-BA passandole come paramentri lo script TCL e il tipo di connessione (USB o Seriale) utilizzata.

Algoritmo 2: Esempio di utilizzo del linguaggio TCL nella programmazione di basso livello della board custom.

```
set bootstrapFile          "AT91Boot.bin"
set ubootFile              "U-Boot_nandflash.bin"
set ubootEnvFile           "uboot-env-prog.bin"

## NandFlash Mapping

set bootStrapAddr          0x00000000
set ubootAddr              0x00040000
set ubootEnvAddr       0x00080000


puts "-I- === Initialize the NAND access ==="

NANDFLASH::Init

puts "-I- === Erase all the NAND flash  ==="

NANDFLASH::EraseAllNandFlash

puts "-I- === Load the bootstrap in the first sector ==="

NANDFLASH::sendBootFile $bootstrapFile

puts "-I- === Load the U-Boot in the next sectors ==="

send_file {NandFlash} "$ubootFile" $ubootAddr 0

puts "-I- === Load the U-Boot environment variables ==="

send_file {NandFlash} "$ubootEnvFile" $ubootEnvAddr 0
```

La sequenza di programmazione nell'algoritmo 2 effettua le seguenti operazioni:

- Imposta le variabili *bootstrapFile, ubootFile e ubootEnvFile* con il nome del binario corrispondente ovvero con le immagini prodotte dalla compilazione rispettivamente di AT91bootstrap e U-Boot. ubootEnvFile fa riferimento ad un'immagine binaria di servizio per U-Boot di cui verrà spiegato succesivamente l'utilizzo nel paragrafo dedicato a U-Boot.
- Imposta le variabili *bootStrapAddr, ubootAddr, ubootEnvAddr* all'indirizzo di memoria in NAND su cui si vorranno scrivere rispettivamente le immagini binarie di AT91bootstrap, U-Boot e Environments di U-Boot.
- Inizializza la NAND-Flash sul bus esterno attraverso la funzione init
  - (NANDFLASH::Init)

- Cancella preventivamente la NAND-Flash attraverso la funzione EraseAllNandFlash
  - (NANDFLASH::EraseAllNandFlash).
- Scrive in NAND le immagini binarie di AT91bootstrap, U-Boot e Environments di U-Boot usando le funzioni:
  - sendBootFile
  - send_file

3.2 AT91bootstrap

Come già introdotto nel paragrafo 3.1, il processo di boot del sistema Embedded oggetto della tesi ha come fase successiva al ROMBOOT un'ulteriore inizializzazione di sistema. Il ROMBOOT di fatto si occupa di preparare il sistema per l'avvio dell'AT91bootstrap che rappresenta dunque il bootloader di secondo livello. A differenza del ROMBOOT l'AT91bootstrap è abbastanza "customizzabile" poichè Atmel ne rilascia il codice sorgente (principalmente in linguaggio C più una parte in Assembly ARM).

I sorgenti dell'AT91bootstrap vanno infatti modificati inserendo il supporto necessario alla propria board. Verranno dunque specificati alcuni parametri di configurazione, il C startup invece è uguale per tutti i dispositivi AT91SAM.

All'interno dello C startup vengono effettuate le seguenti operazioni:

1. Inizializzazione del main oscillator
2. Switch dell'MCK al main oscillator
3. init delle variabili C
4. Branch al codice C

Il file che effettua tali operazioni è il file crt0_gnu.s scritto interamente in Assembly ARM.

Dopo tale inizializzazione standard si ha un'inizializzazione particolare che è specifica per ogni board Atmel o eventualmente come nel nostro caso per una determinata board custom. Di seguito riporto le inizializzazioni necessarie per il sistema Embedded in esame:

1. Disabilitazione watchdog
2. Configurazione PLLA
3. Switch MCK a PLLA/2
4. Configurazione PLLB
5. Abilitazione I-Cache
6. Configurazione PIOs
7. Configurazione Matrix

Questi 7 steps di configurazione verranno implementati attraverso la scrittura di un file in linguaggio C.

Per una questione di chiarezza descrivo adesso brevemente i segnali di clock principali presenti nella board.

Esistono essenzialmente 4 sorgenti di clock: SLCK, MAINCK, PLLA, PLLB.

- SLCK è lo slow clock che nel nostro caso è generato da un quarzo che oscilla a 32768Hz .
- MAINCK è il main clock che viene generato partendo dal quarzo esterno nel nostro caso a 18,432 MHZ.
- PLLA è generato partendo dal MAINCK, è configurabile attraverso un PLL.
- PLLB è generato partendo dal MAINCK, è configurabile attraverso un PLL ed è utilizzato sempre come sorgente di clock per il controller USB.

Queste sono le sorgenti di clock del sistema, poi da queste ovviamente vengono generati altri clock per le specifiche situazioni. Ad esempio il PCK e l'MCK, rispettivamente il processor clock e il master clock, vengono generati partendo da una delle quattro sorgenti di clock descritte e poi opportunamente configurati attraverso una serie prescaler e divisori.

Descritta la struttura fondamentale dei clock di sistema, vediamo come questi vengono gestiti ed impostati nel codice sorgente delle board Atmel.

Di seguito nell'algoritmo 2 è riportata una parte di codice estratta dalla configurazione della evaluation board di Atmel at91sam9260-ek. Tale board è il riferimento per tutte le schede basate sul microcontrollore at91sam9260. L'algoritmo 2 mostra un chiaro esempio di come possono essere inizializzate le sorgenti di clock di sistema.

Algoritmo 3: Configurazione dei clock di sistema

```
/* PCK = MCK = MOSC */
/* PLLA = MOSC*(PLL_MULA + 1)/PLL_DIVA */
pmc_cfg_plla(PLLA_SETTINGS, PLL_LOCK_TIMEOUT);

/* PCK = PLLA = 2 * MCK */
pmc_cfg_mck(MCKR_SETTINGS, PLL_LOCK_TIMEOUT);

/* MCK su PLLA output */
pmc_cfg_mck(MCKR_CSS_SETTINGS,
PLL_LOCK_TIMEOUT);

/* Configurazione PLLB */
pmc_cfg_pllb(PLLB_SETTINGS, PLL_LOCK_TIMEOUT);
```

Le funzioni che vengono usate nell'algoritmo 3 sono messe a disposizione da alcune delle librerie fornite da Atmel. Grazie all'utilizzo delle librerie Atmel è possibile in modo abbastanza semplice effettuare una configurazione dei vari registri e quindi dei controllers presenti nel microcontrollore.

Le funzioni utilizzate sono:

- **pmc_cfg_plla**
  - int pmc_cfg_plla(unsigned int pmc_pllar, unsigned int timeout)
    - Setta il registro PMC_PLLAR ed imposta il timeout per la specifica operazione.
- **pmc_cfg_mck**
  - int pmc_cfg_mck(unsigned int pmc_mckr, unsigned int timeout)
    - Setta il registro PMC_MCKR ed imposta il timeout per la specifica operazione.
- **pmc_cfg_pllb**
  - int pmc_cfg_pllb(unsigned int pmc_pllbr, unsigned int timeout)
    - Setta il registro PMC_PLLBR ed imposta il timeout per la specifica operazione.

L'algoritmo 2 fa uso inoltre di vari define, questi sono customizzati secondo le esigenze specifiche. Nel nostro caso sono tali da generare un PCK di 198MHZ ed un MCK di 99MHZ.

I define usati sono:

- PLLA_SETTINGS
  - Rappresenta il valore che verrà assegnato al registro CKGR_PLLAR che si occupa dell'impostazione del PLLA in particolare è composto da una serie di campi: DIVA, PLLACOUNT, OUTA, MULA.
    * DIVA: Valore per cui sarà diviso il Main Clock per generare l'output del PLLA.
    * PLLACOUNT: Numero di cicli Slow Clock prima che il bit di LOCKA sia posto a 1 nel registro PMC_SR dopo la scrittura del registro.
    * OUTA: Valore di ottimizzazione dipende dalle caratteristiche elettriche.
    * MULA: Moltiplicatore per cui sarà moltiplicato il Main Clock per generare l'output del PLLA.
- MCKR_SETTINGS e MCKR_CSS_SETTINGS
  - Sono i valori che determinano la configurazione del registro PMC_MCKR. Tale registro è composto dai campi CSS, PRES, MDIV.
    * CSS: seleziona la sorgente di clock da usare per generare il Processor Clock e il Main Clock.
    * PRES: E' il valore di prescaler applicato alla sorgente CSS scelta per generare il Processor Clock.
    * MDIV: E' il divisore usato sulla sorgente scelta in CSS per generare il Master Clock.
- PLLB_SETTINGS
  - Rappresenta il valore che verrà assegnato al registro CKGR_PLLBR. Tale registro è sostanzialmente uguale al registro CKGR_PLLAR descritto precedentemente, tuttavia presenta un campo specifico ulteriore denominato USBDIV che è di fatto un ulteriore divisore all'output del PLLB che verrà usato per generare l'USB Clock.
- PLL_LOCK_TIMEOUT
  - E' un valore di timeout poco chiaro se non si conoscono le dinamiche interne e la logica del microcontrollore. Per spiegare cosa rappresenta deve essere fatta una precisazione preliminare. Quando viene reimpostato un valore nei registri CKGR_PLLAR o CKGR_PLLBR automaticamente all'interno del registro PMC_SR viene cancellato il campo rispettivamente di LOCKA o LOCKB e posto a 0. A questo punto i valori dei campi PLLACOUNT e PLLBCOUNT vengono effettivamente caricati nel counter del relativo PLL. Una volta caricato il counter, questo viene decrementato con velocità Slow Clock fino a 0. Raggiunto il valore 0 del counter PLL il campo LOCKA o LOCKB viene posto a 1 e questo verrà segnalato attraverso la generazione di un'interrupt al processore. In questa logica dunque PLLACOUNT e

PLLBCOUNT determinano il numero di Slow Clock necessari perchè venga inviato al processore l'interrupt per segnalare che il PLL è impostato mentre PLL_LOCK_TIMEOUT sarà un valore da assegnare che rappresenterà un tempo ritenuto valido prima che il PLL sia pronto. Le funzioni **pmc_cfg_plla** e **pmc_cfg_pllb** segnalerenno con un valore di ritorno uguale a -1 lo sforamento della restrizione imposta attraverso il timeout PLL_LOCK_TIMEOUT.

Una volta configurati i registri per generare i vari clock di sistema, vengono effettuate alcune operazioni per gestire il coprocessore, la matrice bus, il controller SDRAM e la NAND-Flash.

Algoritmo 4: Abilitazione I-cache

```
/* Configurazione CP15 */
cp15 = get_cp15();
cp15 |= I_CACHE;
set_cp15(cp15);
```

L'algoritmo 4 si occupa di abilitare l'I-cache, **get_cp15** e **set_cp15** sono funzioni che leggono e scrivono dal registro del coprocessore a quello della cpu e viceversa. Entrambe le funzioni sono implementate utilizzando la direttiva "__asm" che consende di inserire un'istruzione o un gruppo di istruzioni Assembly ARM all'interno di un programma C.

- **get_cp15**
  - Legge il contenuto del registro CP15.
- **set_cp15**
  - Imposta il valore passato come argomento nel registro CP15.

Le operazioni su tale registro presentano una particolarità, possono essere effettuate solo attraverso l'uso di una particolare istruzione:

- MCR/MRC{cond} p15, opcode_1, Rd, CRn, CRm, opcode

Uno degli elementi più complessi da gestire è probabilmente l'impostazione corretta del Bus Matrix ovvero la matrice bus da cui dipende la selezione dei controller master e slave e la loro gestione.

Algoritmo 5: Configurazione numero di cicli massimi per il burst mode su EBI (slave slot 3)

```
/* Configurazione EBI Slave Slot a 64 cicli */
writel( (readl((AT91C_BASE_MATRIX + MATRIX_SCFG3)) &
~0xFF) | 0x40, (AT91C_BASE_MATRIX + MATRIX_SCFG3));
```

L'algoritmo 5 ci mostra come venga settato il Bus Matrix, precisamente il registro AT91C_BASE_MATRIX + MATRIX_SCFG3 viene letto e poi riscritto abilitanto un valore di 64 cicli massimi di Master Clock per l'EBI in modalità burst.

E' importante evitare di impostare un tempo per il burst mode sul bus esterno troppo breve o troppo lungo poichè si rischia di compromettere seriamente la velocità del sistema.

Algoritmo 6: Inizializzazione della SDRAM

```
/* Configurazione chip select */
writel(readl(AT91C_BASE_CCFG + CCFG_EBICSA) |
AT91C_EBI_CS1A_SDRAMC, AT91C_BASE_CCFG +
CCFG_EBICSA);

/* Configurazione SDRAM Controller */
sdram_init( AT91C_SDRAMC_NC_9 | AT91C_SDRAMC_NR_13
| AT91C_SDRAMC_CAS_2 | AT91C_SDRAMC_NB_4_BANKS |
AT91C_SDRAMC_DBW_32_BITS | AT91C_SDRAMC_TWR_2
| AT91C_SDRAMC_TRC_7 | AT91C_SDRAMC_TRP_2 |
AT91C_SDRAMC_TRCD_2 | AT91C_SDRAMC_TRAS_5 |
AT91C_SDRAMC_TXSR_8,(MASTER_CLOCK * 7)/1000000,
AT91C_SDRAMC_MD_SDRAM);
```

Nell'algoritmo 6 viene assegnato al Chip select 1 dell'EBI il controller SDRAM (SDRAMC).

Dopo l'assegnazione del controller SDRAM all'EBI CS1 si ha la configurazione del SDRAMC attraverso l'uso della funzione **sdram_init**.

I diversi define presenti descrivono le caratteristiche temporali di assesso al modulo di SDRAM in uso, questi dipendono sostanzialmente dall'hardware che verrà utilizzato, i valori impostati sono reperibili normalmente nel Datasheet della memoria SDRAM.

Vediamo a cosa si riferiscono precisamente i define:

- AT91C_SDRAMC_NC_9
  - si riferisce al numero di colonne nella quale è suddiviso il modulo RAM (9).

- AT91C_SDRAMC_NR_13:
  - si riferisce al numero di righe nella quale è suddiviso il modulo RAM (13).
- AT91C_SDRAMC_CAS_2:
  - si riferisce ai cicli di clock che impiega un dato per attraversare la pipeline interna al modulo RAM.
- AT91C_SDRAMC_NB_4_BANKS :
  - si riferisce al numero di banchi presenti nel modulo RAM (4).
- AT91C_SDRAMC_DBW_32_BITS :
  - si riferisce alla larghezza del bus dati della memoria RAM (32).
- AT91C_SDRAMC_TWR_2:
  - si riferisce al numero di cicli di clock (2 in questo caso) che devono trascorrere dopo il completamento di un'operazione di scrittura, prima che il banco attivo possa essere precaricato.
- AT91C_SDRAMC_TRC_7:
  - determina il numero minimo di cicli di clock (7) tra successivi comandi di BankActivate sul medesimo banco.
- AT91C_SDRAMC_TRP_2
  - determina il numero di cicli di clock (2) che intercorrono tra il comando Precharge e un altro comando.
- AT91C_SDRAMC_TRCD_2
  - determina il numero di cicli di clock (2) tra il comando BankActivate e un'operazione di lettura/scrittura.
- AT91C_SDRAMC_TRAS_5
  - determina il numero di cicli di clock (5) tra il comando BankActivate e il comando Precharge.
- AT91C_SDRAMC_TXSR_8
  - determina il numero di cicli di clock (8) che intercorrono tra l'asserimento di SCKE (alto) e il comando di BankActivate.

I define appena trattati determineranno la configurazione del registro SDRAMC_CR dello SDRAM controllers. Vediamo adesso nel dettaglio la funzione **sdram_init**:

- **sdram_init**
  - int sdram_init(unsigned int sdramc_cr, unsigned int sdramc_tr, unsigned char low_power);
    - sdramc_cr è il valore da assegnare al registro SDRAMC_CR del controller SDRAM.
    - sdramc_tr è il valore da assegnare al registro SDRAMC_TR del controller SDRAM . Tale valore viene ottenuto attraverso la formula: $\frac{MASTER\_CLOCK*7}{1000000}$ dove MASTER_CLOCK è nel nostro caso definito come $\frac{198656000}{2}$ equivalente a circa 100MHZ. Il valore dunque di sdramc_tr rappresenta circa 7us che sono relativi al tempo di refresh per row della memoria. Tale valore è più che sicuro per evitare perdita di dati. Infatti considerando le specifiche sul Datasheet della memeria SDRAM in uso, è possibile notare come il refresh per row possa essere incrementato a 7,8 us senza perdita di dati. I vantaggi nell'avere un ciclo di refresh più rilassato sono di fatto commisurati a un incremento del throughput effettivo tra CPU e memoria.
    - low_power è il valore da assegnare al registro SDRAMC_MDR del controller SDRAM. Il registro SDRAMC_MDR è utilizzato per impostare la modalità con cui verrà gestita la memoria: Low-power SDRAM o SDRAM. AT91C_SDRAMC_MD_SDRAM cioè il define utilizzato per impostare il registro SDRAMC_MDR, mostrato nell'algoritmo 5, seleziona la modalità di gestione memoria non Low-power (valore 0x0).

Configurato il controller SDRAM deve essere assegnata la funzionalità di periferica A al controller PIOC per i pin da PC16 a PC31che diventeranno i dati del bus della memoria SDRAM relativi ai bit 16-31.

Algoritmo 7: Assegnazione periferica A per il PIOC 16-31

```
void SDRAMc_hw_init(void) {
/* Configure PIOs */
/* const struct pio_desc SDRAMc_pio[] = {
{"D16",        AT91C_PIN_PC(16),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D17",        AT91C_PIN_PC(17),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D18",        AT91C_PIN_PC(18),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D19",        AT91C_PIN_PC(19),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D20",        AT91C_PIN_PC(20),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D21",        AT91C_PIN_PC(21),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D22",        AT91C_PIN_PC(22),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D23",        AT91C_PIN_PC(23),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D24",        AT91C_PIN_PC(24),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D25",        AT91C_PIN_PC(25),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D26",        AT91C_PIN_PC(26),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D27",        AT91C_PIN_PC(27),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D28",        AT91C_PIN_PC(28),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D29",        AT91C_PIN_PC(29),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D30",        AT91C_PIN_PC(30),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{"D31",        AT91C_PIN_PC(31),        0,        PIO_DEFAULT,
PIO_PERIPH_A},
{(char *) 0, 0, 0, PIO_DEFAULT, PIO_PERIPH_A},
}; */
/* pio_setup(SDRAMc_pio); */
writel(0xFFFF0000, AT91C_BASE_PIOC + PIO_ASR(0));
writel(0xFFFF0000, AT91C_BASE_PIOC + PIO_PDR(0));
}
```

L'algoritmo 7 presenta una sezione di codice commentato. In realtà tale commento ha una sua ben precisa funzione. Il binario relativo al bootloader di secondo livello ovvero l'AT91bootstrap deve essere inferiore a 4KB, poichè

come abbiamo visto nel paragrafo relativo al ROMBOOT quest'ultimo gestisce solo la memoria SDRAM incorporata nel microcontrollore precisamente può caricare al più 4KB nella memoria SDRAM interna. La limitazione appena descritta impone l'uso delle ultime 2 righe di codice mostrate nell'algoritmo 7. Sostanzialmente il codice commentato e quello non commentato hanno la medesima funzione, tuttavia la scrittura diretta sui registri è notevolmente più snella rispetto all'implementazione classica di un GPIO generico.

- PIO_ASR è il registro Peripheral A Select Register, permette di assegnare la linea di I/O alla funzione di periferica A.
- PIO_PDR è il registro PIO Disable Register, permette di disabilitare il controllo da parte del PIO di una certa linea di I/O consentendo il controllo periferico del pin.

Lo step finale per l'inizializzazione della board utilizzata dovrà essere necessariamente l'impostazione relativa alla NAND-Flash, ovvero il supporto di memoria NVM che conterrà oltre che l'AT91bootstrap anche il bootloader U-Boot il kernel e il filesystem del dispositivo.

Algoritmo 8: Inizializzazione memoria NAND-Flash

```
void nandflash_hw_init(void) {
/* Configure PIOs */
const struct pio_desc nand_pio[] = {
{"RDY_BSY", AT91C_PIN_PC(13), 0, PIO_PULLUP,
PIO_INPUT},
{"NANDCS", AT91C_PIN_PC(14), 0, PIO_PULLUP,
PIO_OUTPUT},
{(char *) 0, 0, 0, PIO_DEFAULT, PIO_PERIPH_A},
};
/* Setup Smart Media, first enable the address range of CS3 in
HMATRIX user interface */ writel(readl(AT91C_BASE_CCFG +
CCFG_EBICSA) | AT91C_EBI_CS3A_SM, AT91C_BASE_CCFG
+ CCFG_EBICSA);
/* Configure SMC CS3 */
writel((AT91C_SM_NWE_SETUP | AT91C_SM_NCS_WR_SETUP
| AT91C_SM_NRD_SETUP | AT91C_SM_NCS_RD_SETUP),
AT91C_BASE_SMC + SMC_SETUP3);
writel((AT91C_SM_NWE_PULSE | AT91C_SM_NCS_WR_PULSE
| AT91C_SM_NRD_PULSE | AT91C_SM_NCS_RD_PULSE),
AT91C_BASE_SMC + SMC_PULSE3);
writel((AT91C_SM_NWE_CYCLE | AT91C_SM_NRD_CYCLE) ,
AT91C_BASE_SMC + SMC_CYCLE3);
writel((AT91C_SMC_READMODE | AT91C_SMC_WRITEMODE
| AT91C_SMC_NWAITM_NWAIT_DISABLE |
AT91C_SMC_DBW_WIDTH_EIGTH_BITS | AT91C_SM_TDF)
, AT91C_BASE_SMC + SMC_CTRL3);
/* Configure the PIO controller */
writel((1 << AT91C_ID_PIOC), PMC_PCER +
AT91C_BASE_PMC);
pio_setup(nand_pio);
nand_recovery();
}
```

Nell'algoritmo 8 viene mostrata la funzione **nandflash_hw_init**, questa svolge tutte le operazioni necessarie per l'inizializzazione della memoria NAND.

- void nandflash_hw_init(void)
  - Viene definita la struttura struct pio_desc nand_pio[] dove vengono configurati i 2 pin generici che verranno utilizzati come RDY_BSY e NANDCS, ovvero quelli collegati fisicamente alla NAND-Flash rispettivamente su R/B e CE.
  - la chiamata: writel(readl(AT91C_BASE_CCFG + CCFG_EBICSA) | AT91C_EBI_CS3A_SM, AT91C_BASE_CCFG + CCFG_EBICSA); si occupa di configurare il campo EBI_CS3A del registro EBI_C-

SA in modo da assegnare EBI Chip Select 3 allo Static Memory Controller e attivare la logica SmartMedia.

- La funzione **nandflash_hw_init** prosegue con la scrittura di 4 registri che configurano lo Static Memory Controller adattando i Timings alle caratteristiche specifiche della NAND-Flash. Tutti i dati di funzionamento della NAND memory sono reperibili all'interno del relativo Datasheet. I registri che effettuano la configurazione dello Static Memory Controller sono:
  * SMC_SETUP3
  * SMC_PULSE3
  * SMC_CYCLE3
  * SMC_CTRL3
- L'impostazione delle linee relative al PIOC è abbastanza semplice. La chiamata di scrittura writel((1 << AT91C_ID_PIOC), PMC_PCER + AT91C_BASE_PMC); ha il compito di abilitare il clock periferico sul PIOC. Questo avviene per sincronizzare le operazioni di comando sulla NAND-Flash con il Master Clock.
- Nella chiamata alla funzione pio_setup(nand_pio); viene passato l'array di strutture pio_desc nand_pio[] per effettuare il setting dei pin GPIO usati dalla memoria NAND.
- L'ultima istruzione dell'algoritmo 8 richiama la funzione nand_recovery (algoritmo 9) che è particolarmente importante a livello pratico, tale funzione può effettuare eventuali procedure di ripristino condizionate ad esempio allo stato di un tasto. In pratica all'avvio dell'AT91bootstrap la pressione di un tasto ovvero il collegamento a massa di un certo pin del microcontrollore, provocherà la cancellazione dell'eraseblock 0x0 della NAND-Flash. Solitamente tale collegamento viene effettuato con l'ausilio di un jumper.

E' da sottolineare che la NAND-Flash è un dispositivo che possiede una logica basata su un bus condiviso, ovvero dati e indirizzi sono fisicamente trasmessi sulle medesime piste(8bit). Tale scelta implica l'utilizzo della tecnica burst per il bus e un aumento di complessità della logica di controllo interna alla memoria.

Per concludere, nell'algoritmo 9 viene mostrato un particolare esempio di funzione per il recovery della NAND-Flash. Se all'avvio del sistema, durante l'esecuzione dell'AT91bootstrap, il pin PA31 viene collegato a massa il blocco 0x0 della NAND-Flash verrà cancellato. A prima vista può sembrare un paradosso cancellare il blocco 0x0 per ripristinare il sistema, tuttavia non lo è. Durante la fase di boot infatti se dovessero presentarsi dei problemi irreversibili sul processo di avvio dell'AT91bootstrap, l'unico modo per accedere al sistema SAM-BA boot e ripristinare il firmware sulla NAND, è quello di avere il settore 0x0 della NAND-Flash privo di un vettore delle eccezioni valido. Come abbiamo visto nel paragrafo 3.1 il ROMBOOT ha una sequenza di boot fissata che controlla la NAND e in caso di vettore delle eccezioni valido avvia l'AT91bootstrap. Se il vettore è valido ma il codice dell'AT91bootstrap ha ad esempio un loop e rimane in stallo la cancellazione del blocco 0x0 della NAND fa si che il sistema non rimanga bloccato sull'avvio. Sarà allora possibile l'enumerazione dell'USB

tramite SAM-BA boot. Attraverso questo sistema si riesce dunque a dare la possibilità al programmatore di riconfigurare adeguatamente il dispositivo senza dover utilizzare metodi più invasivi come l'uso del JTAG o la disconnessione fisica del Chip Select della NAND.

Algoritmo 9: Funzione per il ripristino della memoria NVM Flash

```
static void nand_recovery(void) {
/* Configure PIOs */
const struct pio_desc bp4_pio[] = { {"BP4", AT91C_PIN_PA(31),
0, PIO_PULLUP, PIO_INPUT}, {(char *) 0, 0, 0, PIO_DEFAULT,
PIO_PERIPH_A}, };
/* Configure the PIO controller */
writel((1 << AT91C_ID_PIOA), PMC_PCER +
AT91C_BASE_PMC);
pio_setup(bp4_pio);
/* If BP4 is pressed during Boot sequence */
/* Erase NandFlash block 0*/
if (!pio_get_value(AT91C_PIN_PA(31)) )
AT91F_NandEraseBlock0();
}
```

L'operazione che svolge l'AT91bootstrap è sostanzialmente quella di avviare il bootloader di terzo livello U-Boot, questo viene fatto attraverso un jump sulla memoria RAM esterna dopo aver caricato su di essa l'immagine binaria letta su una certa locazione di memoria NAND-Flash. Il fatto di dover leggere la NAND-Flash e copiare in memoria RAM il suo contenuto giustifica l'importanza di implementare un sistema di recovery pratico con la logica sopra descritta.

L'AT91bootstrap dunque è il primo bootloader scritto sul supporto NVM del sistema, la sua configurazione è come abbiamo visto strettamente legata all'hardware che viene utilizzato nel progettare il proprio sistema Embedded, ed è probabilmente una delle parti software più delicate da sviluppare. L'AT91boot-strap infatti per essere configurato richiede una conoscenza abbastanza approfondita sulla componentistica e le dinamiche temporali che coinvolgono i processi sulle memorie, inoltre va sempre considerata la compatibilità dei componenti scelti con i software impiegati successivamente (U-Boot, Linux).

Uno dei problemi forse meno evidenti in questa discussione sulle memorie e in generale sugli external bus a livello di progettazione hardware è indubbiamente la sincronizzazione e il ritardo sulle piste. Quando vengono valutati segnali con frequenze elevate (centinaia di MHz) i fenomeni di ritardo e riflessione sulle linee sono determinanti e molto spesso impongono stringenti condizioni sulla lunghezza delle piste. Tali restrizioni molto spesso fanno innalzare in modo consistente i costi di produzione per via dell'impiego di processi di fabbricazione multi-layer e l'eventuale utilizzo di resistenze extra di compensazione delle linee. Occorre dunque molta attenzione tanto nella progettazione hardware quanto nella realizzazione di uno stampato di qualità dato che errori sui layer interni

compromettono irrimediabilmente un dispositivo senza possibilità di eventuali riprese manuali delle interconnessioni.

3.3 U-Boot

Das U-Boot è un bootloader rilasciato sotto licenza GPL, è cross-platform ed è condotto dal project leader Wolfgang Denk e mantenuto da un'attiva comunità di sviluppatori e utenti. Il progetto totalmente open source offre un bilanciamento perfetto tra funzionalità e dimensione dell'immagine binaria prodotta. Il compito principale di U-Boot è quello di avviare il kernel Linux.

Per configurare U-Boot in modo adeguato si rende necessario molto spesso la modifica di alcune sue parti del codice sorgente più precisamente l'inserimento del supporto per la propria board ed il makefile specifico.

A differenza dell'AT91bootstrap, U-Boot oltre ad offrire un'interfaccia utente, attraverso la seriale di debug, supporta tutta una serie di dispositivi di comunicazione tra cui USB, TFTP over ethernet e SD Card con cui l'utente potrà interagire. Il concetto da cui si parte nella "customizzazione" del bootloader U-Boot è il seguente.

Ogni produttore di microcontrollori, con architettura supportata, rilascia delle patch per il supporto delle proprie schede di sviluppo ovvero mette a disposizione una serie di librerie specifiche per i suoi prodotti.

Le patch Atmel ad esempio vengono rilascate sia per U-Boot sia per il Kernel Linux per supportare le così dette evaluation board. All'interno del codice sorgente di U-Boot è dunque possibile creare il supporto per la propria board custom basandosi sulle funzioni introdotte nel caso specifico da Atmel.

Si semplifica dunque il compito del programmatore, spesso diventa sufficiente la definizione, all'interno di un opportuno header file, di una serie di configurazioni specifiche. Nell'algoritmo 10 vediamo un esempio relativo alla configurazione della board Atmel at91sam9260ek.

Algoritmo 10: Esempio parte della configurazione della board at91sam9260ek

```
/* bootstrap + U-Boot + env + linux in nandflash */
#define CONFIG_ENV_IS_IN_NAND 1
#define CONFIG_ENV_OFFSET 0x60000
#define CONFIG_ENV_OFFSET_REDUND
0x80000
#define CONFIG_ENV_SIZE 0x20000
/* 1 sector = 128 kB */
#define CONFIG_BOOTCOMMAND "nand read
0x22000000 0xA0000 0x200000; bootm"
#define CONFIG_BOOTARGS
"console=ttyS0,115200 " \
"root=/dev/mtdblock5 " \
"mtdparts=atmel_nand:128k(bootstrap)ro," \
"256k(uboot)ro,128k(env1)ro," \
"128k(env2)ro,2M(linux),-(root) " \
"rw rootfstype=jffs2"
```

Attraverso una serie di define viene scelta la configurazione voluta. Analizziamo dunque la parte di configurazione mostrata nell'algoritmo 10:

CONFIG_ENV_IS_IN_NAND: E' abilitato con il valore 1, stabilisce che le Variabili Ambientali di U-Boot verranno fisicamente salvate sul supporto NAND-Flash.

CONFIG_ENV_OFFSET: Ha un valore esadecimale 0x60000 si riferisce all'offset usato per l'indirizzo delle Variabili Ambientali di U-Boot.

CONFIG_ENV_OFFSET_REDUND: Ha un valore esadecimale 0x80000 si riferisce all'offset per la copia ridondante delle Variabili di U-Boot.

CONFIG_ENV_SIZE: Ha un valore esadecimale 0x20000 è la grandezza in byte delle Variabili Ambientali di U-Boot.

CONFIG_BOOTCOMMAND: è essenzialmente il comando che U-Boot utilizza per effettuare le operazioni di lettura sulla Flash NAND al boot, tale comando non è altro che una variabile di ambiente con nome bootcmd. La direttiva "nand read 0x22000000 0xA0000 0x200000; bootm" è composta da due comandi distinti:

1. nand read:

   Legge dalla NAND dall'indirizzo base più un offset 0xA0000 per una size in byte di 0x200000. I dati verranno copiati in RAM all'indirizzo iniziale 0x22000000.

2. bootm:

   Il comando bootm lancia l'immagine binaria del kernel copiata in RAM all'indirizzo 0x22000000 nel passo precedente.

CONFIG_BOOTARGS: configura gli argomenti del kernel che verrà lanciato. Il define CONFIG_BOOTARGS è composto da una serie di parametri esattamente dalla stringa passata al kernel Linux. Come primo parametro della stringa abbiamo la configurazione della console Linux sulla ttyS0 ovvero sulla porta seriale di debug la velocità di trasferimento è impostata a 115200 bps. Per spiegare gli altri parametri diventa fondamentale aver chiaro come il sistema in esame sia strutturato per quanto riguarda la memoria Flash NAND. Nel caso della board Atmel at91sam9260ek possiamo notare che la NAND utilizzata presenta un block erase di 128KB tale affermazione è confermata dal fatto che nel parametro mtdparts (algoritmo 10) vediamo una prima partizione "128K(bootstrap)ro" che è la partizione su cui è stato copiato l'AT91bootstrap. Come detto in precedenza in realtà l'immagine binaria dell'AT91bootstrap è strettamente inferiore a 4KB (deve esserlo) allora la sceltà dei 128KB riservati alla partizione in sola lettura (ro) dell'AT91bootstrap è pienamente compresibile alla luce di una grandezza di blocco di 128KB che è il minimo erase block della NAND-Flash in uso. Le operazioni di lettura e scrittura in NAND-Flash infatti avvengono sulle pagine mentre l'operazione di cancellazione avviene per erase block, dunque una partizione per essere cancellata correttamente deve presentare almeno una dimensione di 128KB. La board at91sam9260ek presenta complessivamente 6 partizioni mtd, la prima contiene l'AT91bootstrap 128KB, la seconda U-Boot 256KB, terza e quarta sono partizioni dedicate alle variabili d'ambiente di U-Boot ognuna 128KB, la quinta partizione contiene il kernel Linux 2MB infine la sesta contiene il filesystem ovvero il punto di root del sistema tale partizione è formattata in jffs2 ed ha dimensione fino alla fine del dispositivo (indicato con "-" nella stringa di configurazione dell'algoritmo 10).

Jffs2, Journalling Flash File System version 2 è un filesystem specifico per supporto alle memorie NAND Flash, il kernel Linux supporta pienamente questo filesystem nativamente.

La scelta del partizionamento è fondamentale, ma ancor più importante è probabilmente la scelta del filesystem da utilizzare per gestire la partizione di root cioè il rootfs (root filesystem).

La descrizione delle configurazioni presenti nell'algoritmo 10 è solo una parte della configurazione globale di una board in U-Boot, tuttavia è la parte di codice più soggetta a personalizzazione.

Il sistema Embedded che è stato progettato in azienda, utilizza un tipo di configurazione abbastanza diversa da quella analizzata precedentemente per quanto riguarda il partizionamento ed il filesystem.

Nella tabella 5 viene presentato lo schema che si riferisce al partizionamento effettuato sul sistema Embedded Linux custom, notiamo gli indirizzi di partenza e di fine delle partizioni MTD che determinano la loro grandezza. Ogni partizione viene etichettata con un nome particolare che si riferisce al rispettivo contenuto.

Tabella 3.1: Struttura delle partizioni della memoria NAND Flash

Device	Beg. Addr.	End Addr.	Size	Name
mtd0	0x00000000	0x0003ffff	0x00040000	bootstrap
mtd1	0x00040000	0x0007ffff	0x00040000	u-boot
mtd2	0x00080000	0x001fffff	0x00180000	u-boot environment
mtd3	0x00200000	0x007fffff	0x00600000	Kernel
mtd4	0x00800000	0x3fffffff	0x3f800000	filesystems

La partizione mtd4 conterrà inoltre un filesystem formattato in UBI che non è altro che un layer ulteriore di gestione della memoria sopra il layer mtd.

Il filesystem UBI e la sua caratteristica di gestione dei bad blocks verrà approfondita nel capitolo 5.

U-Boot necessita per essere compilato, come l'AT91bootstrap che tuttavia utilizza un linker proprio, di una cross toolchain ovvero di un cross compilatore per ARM.

L'interfaccia di U-Boot è simile a una classica shell, permette operazioni di cancellazione lettura e scrittura della memoria NAND Flash inoltre una delle funzionalità più importanti messe a disposizione è il client per trasferimenti di file via TFTP.

Il trasferimento TFTP permette di caricare in RAM dati presenti su un server TFTP, opportunamente configurato.

Vediamo adesso in dettaglio le variabili d'ambiente di U-Boot che vengono utilizzate per configurarlo e alcuni dei suoi comandi più importanti.

3.3.1 Variabili ambientali e comandi di U-Boot

Le variabili d'ambiente le U-Boot environments sono sostanzialmente un insieme di variabili che definiscono l'effettiva strategia per avviare il sistema.

Esistono due tipi di variabili:

- Custom, ovvero definite dall'utente e utilizzate per eventuali operazioni o per comporre altre variabili.
- Proprie, ovvero alcune variabili riservate riconosciute da U-Boot che rappresentano la configurazione che verrà utilizzata per il dispositivo.

Entrambi i tipi di variabili sono totalmente configurabili dall'utente assegnando un default in sede di compilazione, oppure modificandole dalla console di U-Boot.

Algoritmo 11: U-Boot Environment

```
ethact=macb0
ethaddr=3a:1f:34:08:54:54
serverip=192.168.1.20
ipaddr=192.168.1.55
baudrate=115200
bootdelay=1
```

```
bootargs=mem=64M console=ttyS0,115200 ubi.mtd=4
         root=ubi0:rootfs rootfstype=ubifs
         init=/etc/preinit
bootcmd=nand read 0x22000000 0x00200000 0x00200000;
         bootm 0x22000000
```

L'algoritmo 11 mostra un classico esempio di configurazione delle variabili di U-Boot.

- *ethact*
  - Rappresenta il MAC da utilizzare per l'utilizzo del TFTP.
- *ethaddr*
  - E' l'indirizzo MAC specifico del dispositivo fisico (MAC address).
- *serverip*
  - E' l'IP del server TFTP .
- *ipaddr*
  - E' l'IP assegnato all'interfaccia di rete.
- *baudrate*
  - E' la velocità di comunicazione a cui verrà impostata la seriale di debug.
- *bootdelay*
  - E' il delay prima di ogni boot.
- *bootargs*
  - Sono gli argomenti passati al kernel.
- *bootcmd*
  - E' il comando lanciato al momento del boot.

I comandi supportati da U-Boot sono configurabili in fase di compilazione. Solitamente la strategia di U-Boot è quella di mantenere un set di variabili all'interno dello stesso erase block su cui viene scritto, precisamente nell'area OOB spare. Oltre a queste variabili staticamente compilate normalmente viene creata una partizione mtd come nel nostro caso (mtd2) che contiene altre variabili environments che possono essere modificate a piacimento e che può utilizzare un meccanismo di ridondanza. La tecnica descritta è utilizzata per una questione di sicurezza, infatti se dovessero essere danneggiate le variabili d'ambiente del

bootloader presenti nell'mtd2 queste verrebbero rimpiazzate dalla copia statica nell'OOB area.

Utilizzando il comando help dalla console di U-Boot viene mostrata una lista di tutti i comandi presenti nella particolare compilazione del bootloader.

Algoritmo 12: U-Boot Commands

```
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
bootm   - boot application image from memory
bootp   - boot image via network using BootP/TFTP protocol
branch  - enable or disable branch prediction
cmp     - memory compare
cp      - memory copy
crc32   - checksum calculation
dcache  - enable or disable data cache
echo    - echo args to console
erase   - erase FLASH memory
exit    - exit script
flinfo  - print FLASH memory information
go      - start application at address 'addr'
help    - print online help
icache  - enable or disable instruction cache
imls    - list all images found in flash
itest   - return true/false on integer compare
loadb   - load binary file over serial line (kermit mode)
loads   - load S-Record file over serial line
loady   - load binary file over serial line (ymodem mode)
loop    - infinite loop on address range
md      - memory display
mm      - memory modify (auto-incrementing)
mtest   - simple RAM test
mw      - memory write (fill)
nand    - NAND sub-system
nfs     - boot image via network using NFS protocol
nm      - memory modify (constant address)
ping    - send ICMP ECHO_REQUEST to network host
printenv- print environment variables
protect - enable or disable FLASH write protection
rarpboot- boot image via network using RARP/TFTP protocol
reset   - Perform RESET of the CPU
saveenv - save environment variables to persistent storage
setenv  - set environment variables
sleep   - delay execution for some time
test    - minimal test like /bin/sh
tftpboot- boot image via network using TFTP protocol
usb     - USB sub-system
```

```
version - print monitor version
```

L'algoritmo 12 fa riferimento alla lista dei comandi che possono essere utilizzati dall'iterfaccia shell di U-Boot, tuttavia non tutti di fatto risutano effettivamente utilizzati nel nostro caso.
Quando U-Boot viene compilato è possibile comunque decidere i comandi da inserire per il proprio bootloader, inoltre con un po' di lavoro sul codice sorgente è abbastanza semplice inserire comandi personalizzati.

I comandi più importanti utilizzati nella nostra implementazione sono:

- nand
  - Indispensabile per gestire il supporto NVM NAND-Flash attraverso i sotto comandi read, write, erase. Un esempio del suo impiego è mostrato nell'algoritmo 10 dove la variabile bootcmq lancia i comandi necessari a leggere dalla NAND-Flash il file binario del Linux kernel, opportunamente creato per rendelo eseguibile da U-Boot.
- tftp
  - Utile in fase di sviluppo per caricare file binari o script.
- tftpboot
  - Sostanzialmente simile a tftp viene usato per caricare dalla rete il file binario del kernel all'interno della memoria RAM per essere successivamente lanciato.
- bootm
  - Comando di boot.
- printenv
  - Utile per visualizzare lo stato delle variabili.
- setenv
  - Serve a settare le variabili ad un dato valore fondamentale per lo sviluppo.
- saveenv
  - Salva le variabili modificate in NAND-Flash.

Dopo aver visto le variabili e i comandi disponibili, è importante precisare l'aspetto forse più delicato nella configurazione del bootloader, ovvero i paramentri di boot passati al kernel Linux.

La variabile che viene eseguita come comando all'avvio del bootloader, dopo il tempo stabilito dalla variabile bootdelay, è bootcmd.

Bootcmd sostanzialmente è una variabile che contiene una serie di comandi di U-Boot che vengono eseguiti ad ogni avvio per fare il boot del kernel Linux.

La linea di comando passata al kernel Linux ha il compito di specificare cosa e come dovrà comportarsi il kernel affinchè venga montato un determinato root filesystem.

Il filesystem montato dal kernel Linux risiede in una partizione specifica della NAND-Flash ovvero in una partizione mtd, il compito di U-Boot per avviare il sistema sarà dunque:

1. Leggere la memoria NAND all'indirizzo dell'mtd che contiene il kernel nel nostro caso mtd4.
2. Caricare in memoria RAM il kernel.
3. Avviare il kernel dalla locazione di memoria in RAM.

I parametri passati al kernel Linux ovviamente sono completamente dipendenti dalla struttura del sistema ad esempio dalla disposizione delle partizioni MTD, dal filesystem utilizzato per il mount della ROOT di sistema, dallo script di init, dall'utilizzo di eventuali RAMDISK ecc...

Inoltre i paramentri passati al kernel Linux devono essere in accordo con la sua configurazione in sede di cross compilazione.

Poichè la complessità è molto elevata e una trattazione di tutti gli argomenti e le situazioni possibili è di fatto improponibile, descrivo solo due degli scenari di configurazione della command line passata al kernel.

- Il primo fa riferimento alla configurazione standard del sistema Embedded Linux.
- Il secondo fa riferimento alla configurazione di sviluppo utilizzata dunque dal progettista del sistema.

3.3.2 Configurazione standard command line

La command line passata al kernel da U-Boot nella versione standard è la seguente:

- bootargs=mem=64M console=ttyS0,115200 ubi.mtd=4 root=ubi0:rootfs rootfstype=ubifs init=/etc/preinit

Le parti fondamentali sono costituite dall'argomento init, root, rootfstype, e ubi.mtd.

- mem informa il kernel sulla quantità di memoria RAM disponibile.
- console è settata sulla ttyS0 che rappresenta la seriale di debug del dispositivo il baudrate è a 115200bps.

- ubi.mtd informa il kernel su quale sia la partizione su cui collocare il layer ubi (MTD4).
- root è dove verrà montata effettivamente la root del sistema.
- init indica quale sia lo script o il software da eseguire come padre di tutti i processi.

3.3.3 Configurazione di sviluppo command line

La command line passata nella versione di sviluppo è abbastanza differente, questo perchè quando si lavora su un dispositivo dalle risorse limitate solitamente non si ha a disposizione all'interno di esso tutti i tool di sviluppo necessari. L'esigenza di avere un sistema pratico e veloce di apportare modifiche al filesystem di un dispositivo suggerisce quindi di utilizzare un filesystem di tipo NFS ovvero un network file system.

Il filesystem sarà dunque collocato fisicamente sul disco di un computer host e attraverso il collegamento LAN con il sistema Embedded verrà montato come filesystem del dispositivo.

Questa tecnica abbastanza raffinata è tra le più utilizzate in ambito di sviluppo dei dispositivi Embedded Linux.

La command line passata da U-Boot al kernel è la seguente:

- bootargsnfs=console=ttyS0,115200 noinitrd root=/dev/nfs rw ip=192.168.0.230 nfsroot=192.168.0.3:/home/alberto/NFS init=/init

In questo caso vediamo che la root sarà collocata su un device driver NFS che si occuperà del mount NFS.

Ip rappresenta l'ip dell dispositivo Embedded mentre nfsroot specifica il punto di mount del server NFS host. Questa tecnica è stata utilissima sia in fase di test sia in fase di sperimentazione e rappresenta uno standard consolidato nel settore.

Capitolo 4

Kernel per sistemi Embedded

4.1 Tipi di kernel

Il kernel è una struttura software in grado di garantire un accesso controllato alle risorse hardware di un sistema da parte delle applicazioni. Il compito del kernel è quello di interfacciarsi direttamente con l'hardware ed è dunque un elemento costitutivo per un elaboratore basato su sistema operativo.

L'accesso diretto all'hardware può essere anche molto complesso, quindi i kernel usualmente implementano uno o più tipi di astrazione, il cosiddetto Hardware abstraction layer.

In base al tipo di astrazione è possibile individuare una classificazione per i vari kernel.

Esistono essenzialmente quattro tipi di astrazione:

- kernel monolitici
  - Implementano in modo diretto un completo layer di astrazione dell'hardware.
- microkernel
  - forniscono un insieme ristretto e semplice di astrazione dell'hardware, si appoggiano dunque su software tipo device driver o server per espandere le proprie funzionalità.
- kernel ibridi
  - simile all'approccio a microkernel implementano alcune funzioni aggiuntive al fine di incrementare le prestazioni globali del sistema.
- Esokernel
  - rimuovono tutte le limitazioni legate all'astrazione dell'hardware e si limitano a garantire l'accesso concorrente allo stesso, permettendo alle singole applicazioni di implementare autonomamente le tradizionali astrazioni del sistema operativo per mezzo di speciali librerie.

Ogni tipo di astrazione presenta rispetto alle altre vantaggi e svantaggi.

4.1.1 Kernel Monolitici

L'approccio a kernel monolitico presenta una intefaccia virtuale di alto livello sull'hardware. L'accesso all'hardware avviene attraverso delle chiamate di sistema (system calls) esse implementano servizi base quali:

- gestore dei processi, multitasking
- gestiore della memoria.

I servizi base sono gestiti attraverso moduli che lavorano in modalità supervisore anche detta kernel mode.

Ogni modulo è separato dagli altri, ma tutti i operano nello stesso spazio di indirizzi e un bug in uno di essi può bloccare l'intero sistema. Tuttavia quando l'implementazione è completa ed esente da bug, la stretta integrazione interna dei componenti rende un buon kernel monolitico estremamente efficiente.

Il più grande svantaggio dei kernel monolitici è probabilmente l'impossibilità di aggiungere un nuovo dispositivo hardware senza implementare il relativo modulo al kernel, tale operazione inoltre richiede solitamente la ricompilazione del kernel. In alternativa è possibile compilare un kernel con tutti i moduli di supporto all'hardware, ma il costo è quello di aumentarne la dimensione anche di molti megabyte. Per ovviare parzialmente al problema appena esposto i moderni kernel monolitici ad esempio il Kernel Linux e FreeBSD hanno la possibilità di caricare i moduli in fase di esecuzione a patto che questi siano previsti anche in fase di compilazione, questa soluzione di fatto permette di avere un kernel molto efficiente e soprattutto flessibile. La figura 15 mostra la struttura di un kernel monolitico, si nota la semplicità di interpretazione in questo tipo di astrazione.

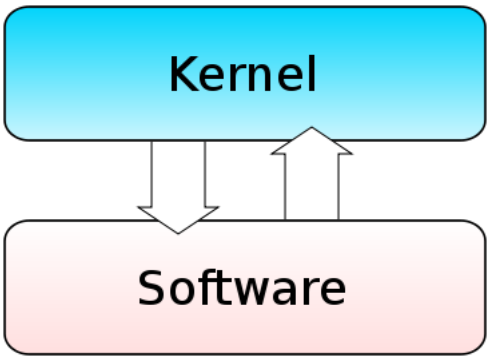


Figura 4.1: Struttura base di un kernel monolitico

4.1.2 Microkernel

La scelta progettuale a microkernel è quella di definire diverse macchine virtuali (servers) semplici al di sopra dell'hardware (vedi figura 16). Un insieme di

chiamate di sistema sono utilizzate per implementare i servizi base del sistema operativo, ovvero thread handler, spazi di indirizzamento e IPC (inter-process comunication). La finalità in questo tipo di implementazione è di dividere il più possibile gli elementi base del sistema operativo dai servizi più di alto livello, i vantaggi sicuramente derivano dal fatto che moduli di alto livello bloccandosi non creano condizioni di blocco dell'intero sistema.

Di contro abbiamo un sistema che in linea di principio ha prestazioni minori rispetto all'approccio monolitico.

Di seguito in figura 16 una rappresentazione a blocchi dell'approccio a microkernel.

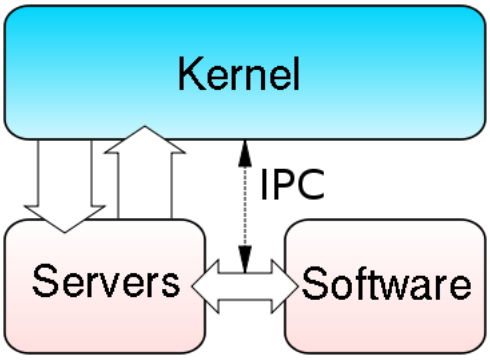


Figura 4.2: Struttura base di un kernel di tipo microkernel

4.1.3 Kernel ibridi

I kernel detti ibridi sono sostanzialmente dei microkernel che contengono all'interno dello spazio kernel codice non essenziale dal punto di vista del layer di collegamento all'hardware. Questo compromesso è finalizzato a massimizzare le prestazioni di alcune parti di codice che normalmente dovrebbero essere implementate nello spazio applicativo del sistema. Tale tecnica di progettazione fu adottata da molti sviluppatori di sistemi operativi prima che fosse dimostrato che i microkernel puri potevano invece avere performance più elevate. Sono diversi i sistemi operativi moderni che impiegano di fatto microkernel modificati, tra questi: Microsoft Windows è l'esempio più noto. Anche XNU, il kernel di Mac OS X, è di fatto un microkernel modificato e anche DragonFly BSD adotta un'architettura a kernel ibrido.

Molto spesso si cade in errore considerando un kernel ibrido come un kernel monolitico in grado di caricare moduli, tale considerazione è però errata poichè il comportamento di transito delle informazioni tra spazio kernel e spazio applicativo nei kernel ibridi condivide concetti architetturali e meccanismi tipici sia dei kernel monolitici sia dei microkernel.

Di seguito in figura 17 un'immagine semplificata puramente concettuale che mostra la struttura architetturale a kernel ibrido.

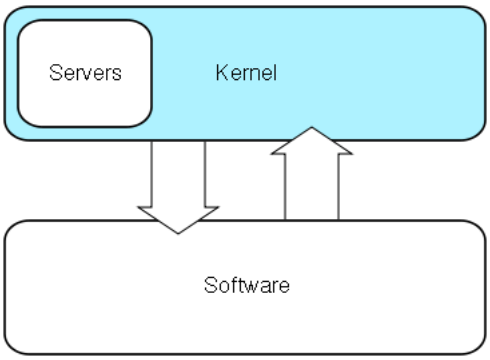


Figura 4.3: Struttura base di un kernel ibrido

4.1.4 Esokernel

Un kernel di tipo esokernel (o exokernel) si basa su un tipo di concetto completamente differente rispetto ai kernel fino qui introdotti. Un esokernel ha un'architettura che divide fortemente il livello sicurezza nell'accesso delle risorse hardware e il livello gestionale. In un esokernel infatti l'obbiettivo è offrire solo sicurezza nell'accesso delle risorse hardware, mentre sarà lo sviluppatore a decidere come gestire tali risorse.

Gli esokernel risultano per quanto detto, di dimensioni contenute inoltre offrono allo sviluppatore un grado di gestione massimo.

Un aspetto interessante degli Esokernel è indubbiamente la possibilità di avere all'interno dello stesso sistema delle libOS diverse (vedi figura 18).

Ad esempio un sistema così progettato può in linea di principio eseguire programmi che facciano uso di librerie di sistema (API) di tipo Unix e Windows.

Le libOS fanno parte del software utente e dunque è possibile avere diverse API sullo stesso sistema operativo.

Attualmente gli Esokernel sono utilizzati più in ambito di ricerca che su applicazioni commerciali.

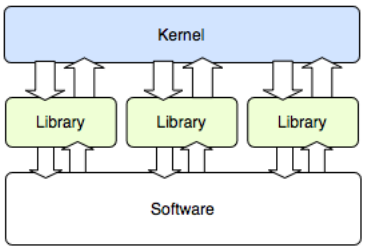


Figura 4.4: Struttura base di un Esokernel

Infine solo per completezza della trattazione è opportuno sottolineare che esistono sistemi che non possiedono un kernel.

Tali sistemi senza kernel delegano completamente allo sviluppatore software il compito di interfacciarsi con l'hardware.

4.2 Il kernel Linux

Il kernel che è stato utilizzato nel sistema Embedded oggetto del tirocinio è essenzialmente il kernel Linux.

Di per sé, il kernel Linux non è molto innovativo. Quando Linus Torvalds ne scrisse il codice, fece riferimento ad alcuni libri classici su Unix, come The Design of the Unix Operating System (Prentice Hall, 1986). Linux ha ancora qualche preferenza verso la linea di base Unix descritta nel libro di Bach (ovvero, SVR4). Tuttavia, Linux cerca di adottare le migliori caratteristiche e scelte di design dei vari kernel Unix.

Tutti i sistemi Unix-Like tendono a concordare su alcune norme comuni IEEE come il Portable Operating Systems based on Unix(POSIX) e X/Open's Common Applications Environment (CAE).

Dalla versione 2.4 il kernel Linux mira ad essere compatibile con lo standard IEEE POSIX. Questo, naturalmente, significa che la maggior parte dei programmi UNIX esistenti possono essere compilati ed eseguiti su un sistema Linux senza la necessità di patch al codice sorgente. Inoltre, Linux include tutte le caratteristiche di un sistema operativo Unix moderno, ad esempio possiede caratteristiche quali: virtual memory, virtual filesystem, lightweight processes, reliable signals, SVR4 interprocess communications, supporto per Symmetric Multiprocessor systems(SMP) ecc.

Di seguito vengono presentate alcune delle caratteristiche del kernel Linux.

4.2.1 Caratteristiche generali del kernel Linux

Come già accennato il kernel Linux fa parte dei kernel detti Monolitici tale caratteristica è tradizionalmente utilizzata nell'implementazione dei kernel Unix o Unix-like, esistono comunque delle eccezioni ad esempio Apple Mac OS X e GNU Hurd.

Linux è compilato e linkato staticamente ed ha la possibilità di caricare o scaricare porzioni di codice attraverso i moduli.

La gestione dei moduli è particolarmente efficiente in Linux essi vengono solitamente impiegati per espandere le funzionalità implementando drivers.

Una caratteristica molto interessante di Linux è la sua organizzazione basata su un insieme di kernel threads.

Un kernel thread rappresenta un contesto di esecuzione che può essere programmato in modo indipendente associandolo a un programma utente o può essere destinato ad eseguire funzioni kernel. Il vantaggio nell'utilizzo di kernel thread risiede nel minor tempo necessario a operare un Context Switch tra thread rispetto ai processi ordinari. I thread infatti solitamente condividono lo spazio di indirizzi rendendo più semplice la comunicazione tra gli stessi.

A livello utente (user space) Linux supporta le applicazioni Multithreads. Il supporto Multithreads è utilissimo per molte delle applicazioni Embedded,

basti pensare a tutte quelle circostanze in cui è indispensabile avere contesti di esecuzione separati capaci di comunicare rapidamente tra loro. L'utilizzo di threads per le applicazioni molto spesso semplifica e ottimizza la comunicazione interprocesso.

Un altro concetto fondamentale è quello di "preemptible kernel", Linux dalla versione 2.6 implementa la possibilità di essere compilato con l'opzione preemptible. Tale caratteristica permette al kernel di sospendere il flusso di esecuzione in kernel mode di un processo per dare priorità ad un altro. In questo modo si rende il sistema più reattivo al prezzo di una diminuzione del throughput. Nello specifico Linux offre varie possibilità di preemptible kernel: No Forced Preemption (Server), Voluntary Kernel Preemption (Desktop) e Preemptible Kernel (Low-Latency Desktop). Un sistema Embedded solitamente cerca di massimizzare il tempo di reazione sacrificando il throughput, questo ragionamento ovviamente ha senso solo nell'eventualità di gestire applicazioni che richiedono tempi di reazione nell'ordine dei millisecondi. Nel caso del progetto di cui ho fatto parte durante l'attività di tirocinio un kernel Preemptible è preferibile, poichè le applicazioni di monitoraggio e gestione dei segnali analogici e digitali vengono spesso implementate attraverso l'uso di programmi in linguaggio C che fanno ampio impiego di funzioni quali poll() e select() su file descriptor.

Gli aspetti implementativi del kernel Linux sono molto complessi e non è dunque semplice conoscere in dettaglio le oltre sei milioni di righe di codice che lo compongono, tuttavia possiamo individuare 5 blocchi funzionali costitutivi:

- process scheduler
- memory manager
- virtual filesystem
- network interface
- Inter process communication.

1. Lo scheduler controlla e permette l'accesso dei processi all'unità centrale di elaborazione.
2. Il Memory manager gestisce l'uso di memoria dei processi fornendo un accesso sicuro e controllato.
3. Il virtual filesystem ha il compito di riassumere alcuni dettagli dei dispositivi hardware attraverso dei file che rappresentano un'interfaccia di una determinata periferica.
4. La network interface è progettata per fornire l'accesso ai protocolli di rete e al relativo hardware.
5. L'Inter-process-communication è un componente funzionale molto complesso, ha il compito di gestire i meccanismi di comunicazione tra i processi del sistema.

4.3 Caratteristiche architetturali

Il codice del kernel Linux è estremamente portabile nel senso che è possibile compilare il kernel per svariate architetture. Nella versione 2.6.x.x le principali architetture supportate sono le seguenti:

- Alpha
- ARM
- x86
- MIPS
- Power
- PowerPC
- SPARC

Per ogni architettura esistono diversi supporti specifici per piattaforma. Ad esempio all'interno dei sorgenti del kernel Linux nell'architettura ARM(arch/arm/) troviamo il supporto per le seguenti piattaforme: mach-aaec2000, mach-at91, mach-bcmring, mach-clps711x, mach-cns3xxx, mach-davinci, mach-dove, mach-ebsa110, mach-ep93xx, mach-footbridge, mach-gemini, mach-h720x, mach-imx, mach-integrator, mach-iop13xx, mach-iop32x, mach-iop33x, mach-ixp2000, mach-ixp23xx, mach-ixp4xx, mach-kirkwood, mach-ks8695, mach-lh7a40x, mach-loki, mach-lpc32xx, mach-mmp, mach-msm, mach-mv78xx0, mach-mx25, mach-mx3, mach-mx5, mach-mxc91231, mach-netx, mach-nomadik, mach-ns9xxx, mach-nuc93x, mach-omap1, mach-omap2, mach-orion5x, mach-pnx4008, mach-pxa, mach-realview, mach-rpc, mach-s3c2400, mach-s3c2410, mach-s3c2412, mach-s3c2416, mach-s3c2440, mach-s3c2443, mach-s3c24a0, mach-s3c64xx, mach-s5p-6440, mach-s5p6442, mach-s5pc100, mach-s5pv210, mach-s5pv310, mach-sa1100, mach-shark, mach-shmobile, mach-spear3xx, mach-spear6xx, mach-stmp378x, mach-stmp37xx, mach-tegra, mach-u300, mach-ux500, mach-versatile, mach-vexpress e mach-w90x900. La quantiatà di piattaforme ARM supportate è dunque notevole. Ogni piattaforma mensionata ha una directory propria all'interno dei sorgenti del kernel, che implementa nello specifico le caratteristiche di un determinato processore. Per comprendere meglio l'affermazione fatta è opportuno sottolineare che esistono almeno due livelli implementativi per un microcontrollore di tipo ARM; il primo è quello del core ovvero del modello ARM di processore, il secondo è relativo alla scelta delle periferiche integrate all'interno del singolo chip. Per quanto detto due piattaforme diverse possono utilizzare di fatto lo stesso core ARM, ma avere implementazioni driver completamente diverse in sintonia con l'hardware periferico del microcontrollore. Le caso del microprocessore trattato nel corso del tirocinio la piattaforma è Atmel precisamente AT91 che indica una grande famiglia di microcontrollori con determinate caratteristiche dei propri controllers.

La forza del kernel Linux nei sistemi Embedded è proprio la varietà di piattaforme disponibili, ogni produttore infatti introduce varie patch di supporto per il proprio hardware, che se adeguatamente riconosciute e validate, vengono inserite nella main line del kernel.

La filosofia implementativa di Linux distingue il codice sorgente in due parti ben distinte: code hardware dipendente e code hardware indipendente.

Questo significa che Linux tende a dividere il layer di supporto per un determinato processore dal layer "più alto" di implementazione dei servizi, in tale maniera Linux riesce ad uniformare e generalizzare il più possibile i layer di supporto per tutto ciò che comprende protocolli o stack vari. In termini pratici questo sforzo a livello di programmazione si traduce in un'omogeneità di Linux sulle varie piattaforme, per cui l'utente (e in misura minore anche lo sviluppatore) si troverà a lavorare su contesti mantenuti il più possibile simili anche se su hardware molto diversi. Il kernel Linux inoltre porta con se tutto un supporto a livello applicativo e di sistema operativo (GNU) che ha la possibilità di essere utilizzato su tutte le piattaforme compatibili. Fino adesso abbiamo trattato dei concetti generali del kernel Linux, ma a mio avviso la trattazione dei moduli drivers richiede un maggiore approfondimento dato l'importanza che questi ricoprono nei sistemi Embedded. Generalmente viene cosiderato un driver un applicativo in grado di gestire opportunamente delle risorse hardware specifiche, siano queste periferiche o incorporate nel sistema.

In ambito dei microcontrollori privi di sistema operativo si può affermare senza troppi timori che l'attività di uno sviluppatore è nella maggior parte dei casi di implementazione di driver specifici per una determinata finalità. Per quanto riguarda un dispositivo dotato di sistema operativo un driver può avere concettualmente vari layer di astrazione, data la grande stratificazione software in cui può essere collocato.

Un driver implementato attraverso user space che si appoggia su drivers esistenti che comunicano con l'hardware (o comunque con il kernel) può elaborare le informazioni e compiere una funzione specifica. In questo caso guardando il dispositivo come uno schema a blocchi, avremo un unico blocco logico composto da driver e applicativi (che utilizzano una o più librerie) che pilotano e gestiscono un certo macroprocesso.

Da user space è possibile dunque operare sull'hardware principalmente attraverso i driver già presenti nel kernel o per mezzo di quelli inseriti attraverso moduli. Contemporaneamente possono essere utilizzate le librerie C per il supporto di protocolli standard.

Dalla perfetta integrazione di driver e librerie C nasce una grande capacità di produrre applicativi, in tempi decisamente più brevi, a parità di complessità, rispetto ad una programmazione su microcontrollori privi di OS.

Inoltre la robustezza del sistema è massima, data la grande stabilità delle librerie POSIX/BSD e del Kernel Linux.

Lo schema di driver descritto in realtà non è concettualmente corretto o meglio non in linea con la terminologia adottata nel kernel Linux, per tale motivo è opportuno introdurre alcuni concetti forndamentali.

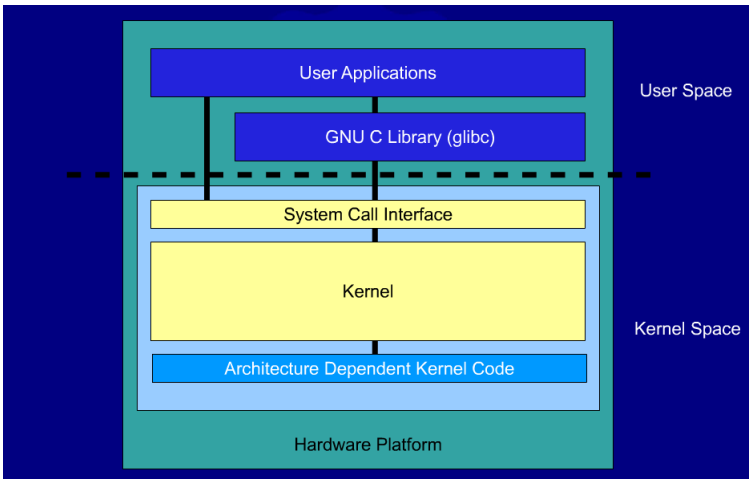


Figura 4.5: Schema a blocchi architetturale di un sistema Linux

Ogni processore ha più modalità di funzionamento Linux sfrutta due modalita che vengono chiamate kernel mode (o modalità supervisore) e user mode (o modalità utente) le due modalità si distiguono per diverse ragioni tra queste la priorità sull'accesso dell'hardware è quella più evidente. In figura 19 notiamo la netta separazione tra i due diversi stati del processore ed i corrispondenti elementi architetturali caratteristici di un sistema Linux. Se consideriamo vari processori con architetture distinte, ovviamente si avranno per ognuno di essi delle procedure "code dependent" dipendenti per effettuare il così detto Context Switch ovvero la commutazione tra user mode - kernel mode e viceversa. Da questa separazione attuata nelle modalità di accesso all'hardware, deriva uno sviluppo software driver ben determinato specializzato per un particolare microcontrollore. La commutazione di contesto non avviene ovviamente in modo istantaneo, tuttavia per le applicazioni comuni tale latenza è sicuramente trascurabile. Esistono delle situazioni particolari dove invece sarebbe preferibile non avare tempi di latenza dettati da cambio di contesto ad esempio è semplice prevedere che un driver che abbia il compito di pilotare un certo hardware possa avere alcune specifiche a livello delle tempistiche più stringenti rispetto ad una comune applicazione.

Per quanto detto risulta essenziale che un driver abbia un contesto di esecuzione in kernel mode e che condivida dunque sostanzialmente lo stesso spazio di indirizzi del kernel e di conseguenza degli altri moduli driver.

Un driver inteso dunque nella sua accezione corretta è nettamente più ottimizzato (oltre ad avere maggiori "libertà") rispetto a un'applicazione che gestisca l'hardware ma attraverso user space.

Un driver di un dispositivo con sistema operativo basato su kernel molitico sarà essenzialmente un modulo che ha un contesto di vita interno al kernel è parte del kernel stesso e ha il compito di "esporre" nello user space un'intefaccia, il più possibile generale e trasparente per l'utilizzatore. In Linux tutto è un file (o almeno questo è quello che si vuole ottenere) la nota frase: *everything is a*

file riassume in modo abbastanza chiaro la filosofia implementativa dei driver Linux. L'idea base è quella che un file possa descrivere in modo intuitivo un certo dispositivo hardware. Il file o in generale una serie di file rappresentano lo strumento con cui Linux espone un'interfaccia utente per un dato dispositivo.

Un file nella sua accezione astratta non è altro che una struttura di memoria su cui è possibile effettuare alcune operazioni.

Un concetto fondamentale per implementare un tale sistema è innanzitutto l'uniformità della trattazione dei diversi tipi di filesystem specifici per supporti NVM di diverso tipo.

Tale uniformità viene raggiunta nel kernel Linux attraverso il così detto VFS Virtual File System che consente di dialogare con diversi tipi di filesystem in modo trasparente all'utente e omogeneo.

4.3.1 Il VFS di Linux

Il VFS è un layer di astrazione per i vari file system supportati da Linux.

Questo strato software, presente nel kernel, permette al sistema di supportare vari filesystem senza "conoscerli" infatti esso mette a disposizione delle applicazioni, che girano sul sistema, un'interfaccia composta da una serie di funzioni generiche che non fanno riferimento ad uno specifico filesystem. Tali funzioni chiamano poi delle opportune routine che devono essere implementate dal driver dello specifico filesystem considerato. In figura 20 viene presentato uno schema a blocchi dell'VFS in cui i filesystem minix, ext2, FAT sono implementati sollo il layer VFS. I driver per la gestione dei vari tipi di filesystem devono comunque rispettare determinate regole per potersi integrare con il sistema VFS.

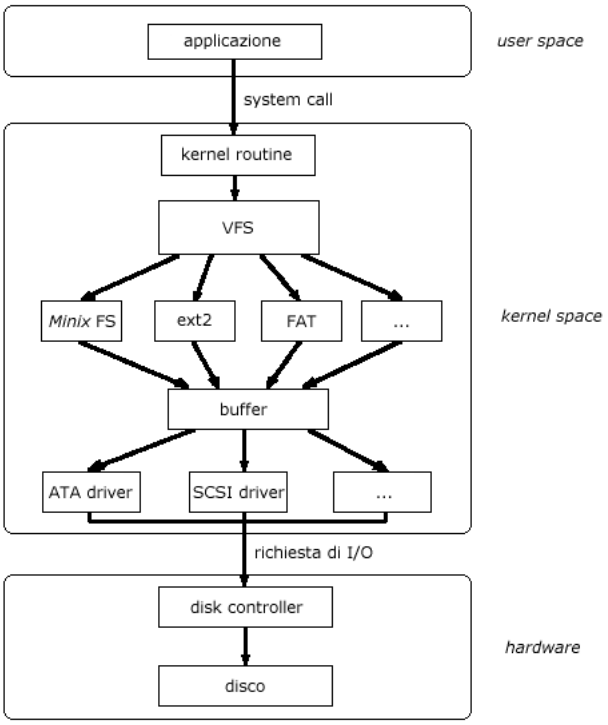


Figura 4.6: VFS unix-like

Proprio grazie al VFS, tutti i filesytem utilizzati da GNU/Linux hanno la stessa struttura logica, ovvero vengono presentati all'utente tutti nella stessa maniera.

Il VFS è un chiaro esempio della filosofia Linux della trasparenza nell'accesso alle risorse.

Per ottenere un'uniformità in termini di gestione files è prima necessario ottenere un'omogeneità di trattamento delle strutture relative ai file anche considerando diversi tipi di supporti NVM.

Dunque essenzialmente Linux garantisce con il suo VFS uniformità tra struttura ed organizzazione della memoria tra vari tipi di hardware. Seguendo lo stesso principio per garantire l'uniformità dell'uso dei vari drivers Linux tende a mascherare le differenze implementative hardware dipendenti, presentando all'utente quanto più possibile le medesime dinamiche gestionali per le interfaccie (file).

4.3.2 Device Driver

Indipendentemente dall'architettura usata, in Linux esistono principalmente due tipi di device driver: character device e block device (in realtà ci sono delle eccezioni).

I block device, abbreviato blkdevs, rappresentano tutti quei dispositivi indirizzabili a blocchi. Tali device generalmente supportano seeking e dunque

random access dei dati. Ad esempio un device block può essere un normale hard disk o un cdrom.

In un sistema Embedded inoltre viene spesso considerata erroneamente una memoria NAND o NOR (mtd device) un dispositivo a blocchi. Nel prossimo capitolo verrà trattato adeguatamente tale argomento e spiegato il perchè una NAND-Flash non è un block device.

Molto spesso i block device sono rappresentati nello user space attraverso un file speciale (node) su cui è possibile eseguire operazioni di mount come filesystem.

I character device , abbreviato cdevs, sono solitamente dispositivi non indirizzabili forniscono accesso ai dati solo come un stream, in genere di caratteri (byte). Esempio di cdevs sono: tastiere, mouse, stampanti, e la maggior parte degli pseudo-dispositivi. I device a carattere sono accessibili nello user space tramite un file speciale chiamato nodo di dispositivo a caratteri. A differenza dei dispositivi a blocchi, le applicazioni possono interagire con i dispositivi a carattere direttamente attraverso i rispettivi node che rappresentano dei veri e propri buffer. Linux fornisce altri tipi di device specializzati per un singolo compito e non appartenenti alle categorie prima descritte, un importante tipo di device, in ambito Embedded e non, sono certamente i miscdevs ovvero i miscellaneous device. Sono in realtà una forma semplificata dei dispositivi a carattere (cdevs) consentono all'autore di un driver per una periferica di rappresentare dispositivi in modo il più possibile semplificato. E' bene ricordare che non tutti i driver di periferica rappresentano dei dispositivi fisici. Alcuni driver di periferica vengono detti virtuali, questi avranno accesso a funzionalità del kernel. Tali driver vengono chiamati pseudo devices, alcuni dei più comuni sono il generatore di numeri casuali del kernel (accessibile in /dev/random e /dev/urandom), il dispositivo null (accessibile in /dev/null), il dispositivo di azzeramento (accessibile in /dev/zero) e il dispositivo di memoria (accessibile in /dev/mem). La maggior parte dei driver di periferica tuttavia tendono a rappresentare hardware fisico.

Appare più chiaro a questo punto la struttura logica dei device driver la loro importanza nell'ambito Embedded è basilare poiché è frequente la loro implementazione per gestire risorse hardware custom.

Fino ad adesso abbiamo fissato l'attenzione su cosa siano i driver e come questi vengano implementati nell'architettura del kernel è fondamentale però avere padronanza anche con alcuni filesystem particolari che molto spesso vengono detti filesystem virtuali, ma non devono essere confusi con l'abstraction layer VFS. Tali filesystem vengono detti virtuali semplicemente perché non occupano spazio effettivo sul supporto NVM (memoria non volatile) ed il loro impiego è strettamente legato ai driver e dunque alla comunicazione tra risorse hardware e applicazioni software.

4.3.3 FHS (Filesystem Hierarchy Standard)

Prima di descrivere le caratteristiche dei filesystem virtuali devfs procfs e sysfs va precisato, che anche se in linea di principio il kernel Linux tende ad offrire un filesystem uniforme per ogni tipo di architettura supportata, nulla vieta al progettista di un sistema Linux di utilizzare alberi strutturali del filesystem completamente personalizzati fuori standard.

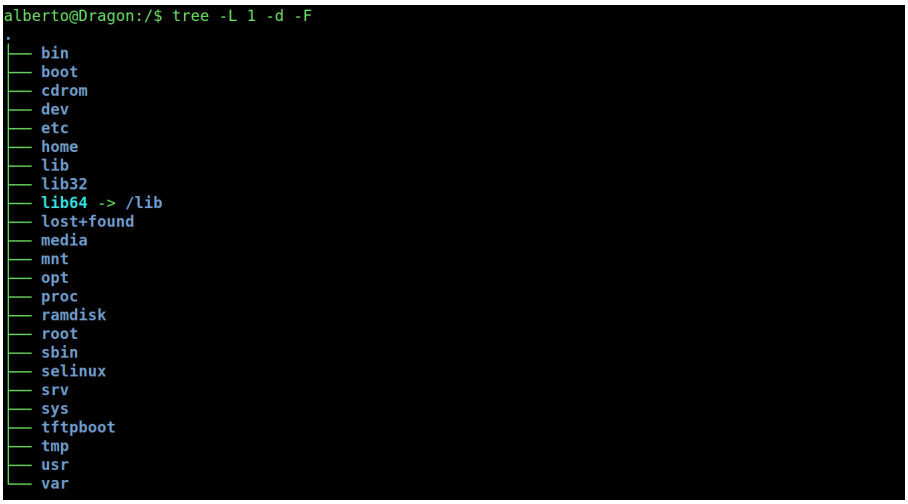


Figura 4.7: Gerarchia standard filesystem Linux in Ubuntu 11.04

La figura 21 mostra la gerarchia utilizzata in un classico sistema Linux Ubuntu, il programma tree visualizza la gerarchia troncata al primo livello delle sole directory.

4.3.3.1 Panoramica FHS

L'FHS (Filesystem Hierarchy Standard) definisce le directory principali ed il loro contenuto nei sistemi operativi Linux e in generale nei sistemi Unix-like.

Il processo di sviluppo di una gerarchia standard per i filesystem iniziò nell'agosto 1993 con l'idea di ristrutturare la struttura dei file e delle directory di Linux. L'FSSTND (Filesystem Standard), uno standard di gerarchia dei filesystem, fu rilasciato il 14 febbraio 1994. Successivamente seguirono altre modifiche allo standard originario. Da questa esigenza nata in ambito puramente open source, agli inizi del 1996 viene presa in considerazione l'idea di sviluppare una versione dell'FSSTDN da utilizzare non solo su Linux, ma anche in tutti gli altri sistemi Unix-like con l'aiuto di alcuni membri della comunità di sviluppo BSD. Come risultato di questa collaborazione si misero a fuoco le caratteristiche che accomunavano tutti i sistemi Unix-like. Considerando questa nuova apertura al mondo Unix, il nome dello standard venne cambiato in Filesystem Hierarchy Standard (abbreviato in FHS). Tuttavia lo standard non viene molto spesso seguito in modo sistematico e alcune modifiche ad esempio l'introduzione di /svr/ vengono spesso ignorate. Lo standard molto spesso viene anche arricchito con altre directory secondo il sistema operativo o la distribuzione utilizzata.

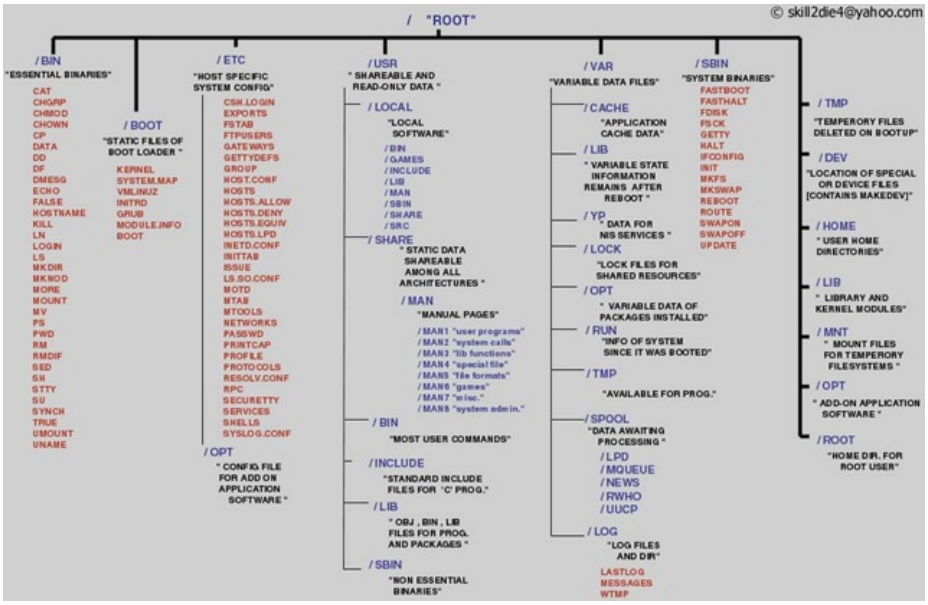


Figura 4.8: Struttura FHS per i sistemi unix-like

La figura 22 mostra la struttura gerarchica dello standard filesystem per i sistemi Unix-Like. Ogni directory viene definita precisando quale dovrebbe essere il suo contenuto. Di seguito la descrizione di alcune delle directory più importanti dell'FHS.

- /dev/
  - Contiene voci del filesystem che rappresentano dispositivi collegati al sistema. Questi file sono essenziali perchè il sistema possa funzionare correttamente.
- /bin/
  - Contiene i file binari per tutti gli utenti.
- /home/
  - Contiene i file personali di uno specifico utente del sistema. Viene solitamente assegnata dal super utente root al momento della creazione dell'utente specifico.
- /etc/
  - E' riservata ai file di configurazione locali presenti sulla macchina. Nessun file binario deve essere inserito in /etc/.
- /lib/

  - La directory /lib/ dovrebbe contenere solo le librerie necessarie all'esecuzione dei file binari presenti in /bin/ e /sbin/. Queste immagini di librerie condivise sono particolarmente importanti per l'avvio del sistema e l'esecuzione di comandi all'interno del filesystem di root.

- /mnt/
  - La directory /mnt/ è riservata ai filesystem montati temporaneamente, come i mount del filesystem NFS.

- /opt/
  - La directory /opt/ fornisce un'area per la memorizzazione di pacchetti applicativi statici di grandi dimensioni. E' utile per raccogliere molti dei file che rimarrebbero sparsi nel filesystem.

- /proc/
  - La directory /proc/ contiene i file speciali che estraggono o inviano informazioni al kernel.

- /sbin/
  - La directory /sbin/ contiene normalmente i binari essenziali per l'avvio, il ripristino, il recupero e/o la riparazione del sistema oltre a quelli presenti in /bin. Qualunque altro programma eseguito dopo il montaggio della directory /usr/ (quando non si verifica alcun problema) deve essere collocato in /usr/sbin.

- /sys/
  - La directory /sys/ utilizza il nuovo file system virtuale specifico al kernel 2.6. Grazie al maggior supporto per dispositivi hardware hot plug con il kernel 2.6, la directory /sys/ contiene le stesse informazioni contenute in /proc/, ma in grado di visualizzare una panoramica gerarchica di informazioni specifiche riguardanti i dispositivi hot plug.

- /usr/
  - La directory /usr/ contiene tutti i file che possono essere condivisi attraverso macchine multiple. La directory /usr/ è solitamente nella propria partizione, ed è montata come sola lettura.

- /var/
  - Tale directory contiene le directory e i file di spool, dati di loggin e di amministrazione, file temporanei e transitori.

Ovviamente a meno di particolari necessità la tendenza nei sistemi Linux è sempre quella di seguire lo standard, per cui da adesso in avanti anche per quanto riguarda i filesystem virtuali che verranno trattati successivamente considereremo sempre il loro punto di mount standard, che per procfs sysfs e devfs è rispettivamente: /proc, /sys, /dev.

4.3.4 Pseudo filesystem

Linux possiede all'interno della propria gerarchia di directory alcune strutture di file fisicamente non presenti nel supporto NVM e che vengono mostrate all'utente in modo del tutto trasparente come se fossero effettivamente presenti nello storage.

Tali directory vengono popolate con alcuni file creati al momento dell'inizializzazione del sistema, più esattamente all'interno degli script di avvio rcS. Il ruolo di tali file è quello di esporre delle interfacce all'utente su alcuni servizi gestiti dal kernel. Possiamo individuare essenzialmente tre tipi di pseudo filesystem.

4.3.4.1 Procfs

Il procfs è uno pseudo filesystem che secondo lo standard FHS trattato precedentemente viene fisicamente montato nella directory /proc.

Innanzitutto è utile chiarire cosa si intende per montare il filesystem proc e in quale momento questo viene fatto. Proc storicamente fu utilizzato inizialmente per ottenere alcune informazioni sui servizi gestiti dal kernel il tutto attraverso la lettura dei suoi files. Quando il kernel Linux viene lanciato attraverso un bootloader cerca, secondo le specifiche impartite dalle opzioni presenti nei parametri di boot, di montare il root ovvero la radice di tutta la gerarchia delle directory. Quando avviene questa operazione viene avviato immediatamente un certo programma di init che può essere indipendentemente uno shell script o un programma C. L'init è il processo padre per ogni altro processo che il sistema potrà avviare. L'init normalmente esegue alcune operazioni base prima di andare a leggere un file chiamato inittab (almeno in certe implementazioni) che di fatto specifica le varie inizializzazione dei run level e avvia il relativo script rcS. Lo script rcS si occupa dell'inizializzazione del sistema ed è qui che normalmente viene montato il procfs attraverso il comando:

- mount -t proc proc /proc

al momento del mount la directory /proc viene popolata da una serie di cartelle e file speciali che espongono alcuni servizi di comunicazione direttamente con il kernel.

Ecco una serie di esempi delle informazioni ricavabili dai file presenti in /proc:

- /proc/cpuinfo: Informazioni sul processore.
- /proc/meminfo: Stato della memoria.
- /proc/version: Versione kernel e informazioni di compilazione.
- /proc/cmdline: parametri passati al kernel (command line).
- /proc/<pid>/cmdline: command line del relativo processo.
- /proc/modules: moduli inseriti.
- /proc/interrupts: handler installati per interrupt.

4.3.4.2 Sysfs

Il sysfs può essere considerato l'evoluzione moderna del procfs. Ci sono vari motivi per cui è stato introdotto il sysfs, sostanzialmente il vero motivo è che di fatto il procfs fu nel corso degli anni usato in modo abbastanza caotico, per cui attualmente è estremamente complicato gestirlo, dunque si è deciso di introdurre il sysfs che avendo una struttura organizzata in modo funzionale in sotto directory si presenta come un sistema notevolmente più gestibile e comprensibile.

Come per il procfs il momento in cui viene solitamente montato è negli script di rcS attraverso il comando:

- mount -t sysfs sysfs /sys

Contrariamente a quanto si può immaginare attualmente un sistema moderno basato su un kernel Linux recente continua a mantenere il procfs anche se implementa il sysfs. Questo perchè la migrazione da procfs a sysfs purtroppo non è cosi agevole, in molti casi allora si è preferito mantenere entrambe le soluzioni.

Per dare un aspetto concreto al sysfs presento un esempio classico di utilizzo per un sistema Embedded.

L'esigenza più comune in ambito industriale è quella di utilizzare il GPIO (general purpose I/O) di un certo microcontrollore per pilotare (solitamente attraverso un circuito driver a transistor o mosfet) o acquisire lo stato logico di un pin collegato ad una periferica esterna. Tale esigenza ha spinto alla realizzazione di un sistema generale di controllo dei GPIO attraverso user space precisamente dialogando con certi file presenti appunto nel sysfs.

```
alberto@Dragon:~/NFS/etc/init.d$ tree /sys/ -L 1 -F
/sys/
├── block/
├── bus/
├── class/
├── dev/
├── devices/
├── firmware/
├── fs/
├── hypervisor/
├── kernel/
├── module/
└── power/
```

Figura 4.9: Gerarchia interna del sysfs su Linux Ubuntu 11.04

La figura 23 rappresenta la gerarchia all'interno del sysfs ogni cartella fa riferimento ad un particolare insieme di informazioni ad esempio module conterrà i moduli del kernel, nel caso dei GPIO l'oggetto che viene creato è di fatto un device. La directory class (vedi figura 23) contiene una serie di link simbolici a device raccolti secondo una certa classe per cui accedendo a tale directory troveremo la directory gpio che è appunto una classe registrata.

La directory gpio contiene essenzialmente due file rispettivamente export e unexport oltre a una serie di directory che fanno riferimento ai vari bank gpio possibili.

Per utilizzare un gpio del sistema usado sysfs è sufficiente scrivere all'interno del file export il numero del gpio di cui vogliamo ottenere il controllo. Se l'export va a buon fine troveremo all'interno della cartella /sys/class/gpio/ la directory corrispondente al gpio richiesto con nome gpioN dove N è il numero del gpio.

La cartella specifica gpioN conterrà a sua volta una serie di file che interagistono direttamente con l'hardware ad esempio troveremo il file value che se letto restituirà il valore corrente del pin gpio, mentre se scritto (0 o 1) imporrà il valore logico a condizione che sia stato impostato come output.

L'impostazione della direzione di un gpio si fa sempre attraverso un file presente nella cartella gpioN.

4.3.4.3 Devfs / udev / devtmpfs

Per devfs solitamente si intende il Virtual filesystem che viene storicamente montato nella directory /dev di un sistema Linux. Cerchiamo di spiegare a cosa serve e quale è il suo successore realmente utilizzato.

Devfs viene introdotto nella versione 2.3 del kernel Linux e si occupa essenzialmente della gestione dei device creando dei file speciali all'interno di una determinata directory (/dev), tali file sono caratterizzati essenzialmente da due numeri Major e Minor oltre a una lettera che indica il tipo di device: c per character b per block. I file sono legati sostanzialmente ai device registrati nel kernel. In sede di programmazione di un driver registrato come device sarà dunque possibile decidere quali funzioni avrà il file speciale che lo rappresenterà in user space. Il meccanismo di creazione di una struttura di file speciali veniva fatto appunto attraverso l'uso di un Virtual filesystem denominato devfs. Tuttavia nel corso dello sviluppo di Linux sono sorte diverse problematiche relative al funzionamento del devfs, essenzialmente esso era staticamente integrato nel kernel e non dava la possibilità di nominare in modo persistente un certo dispositivo. Tali limitazioni hanno portato alla realizzazione di un sistema notevolmente più flessibile interamente gestito in user space denominato udev presente a partire dalla versione kernel 2.5 . E' giusto precisare che udev si avvale ovviamente di un filesystem di tipo tmpfs per implementare i vari file speciali. Recentemente dalla versione 2.6.32 è stato introdotto un altro sistema di mount dei device driver attraverso un filesystem dedicato denominato devtmpfs che ha la possibilità di integrarsi perfettamente con udev dal punto di vista della gestione dinamica dei device, ma introduce un notevole vantaggio soprattutto nei sistemi Linux provi di initrd / initramfs. La panoramica sulle tecniche di mount della directoy /dev di fatto non è conclusa poichè le tecniche appena illustrate solitamente non vengono utilizzate in ambito Embedded. Normalmente in un sistema dalle risorse hardware particolarmente limitate si tende ad utilizzare un software particolare chiamato BusyBox. BusyBox non è altro che un insieme di tool incorporati in un semplice file binario che a seconda del parametro con cui viene invocato da accesso ad una certa funzionalità. BusyBox per gestire i device driver e creare i file speciali in /dev si affida ad un approccio simile a quello visto con udev. Ovvero viene montato un filesystem virtuale (tmpfs) in /dev e successivamente viene utilizzato un software user space denominato mdev che gestisce come deve essere popolata la directory /dev. Concludendo possiamo affermare che indipendentemente dal sistema di mount utilizzato la directory /dev contiene una serie di file speciali che rappresentano i device driver

registrati nel kernel Linux e che non rappresentano uno spazio di memoria fisica del dispositivo NVM utilizzato per lo storage.

4.4 Esempio di implementazione di un character device driver

Nella sezione precedente abbiamo trattato alcuni aspetti relativi ai device driver. Il mezzo di comunicazione tra hardware e utente scelto in Linux è il file che rappresenta una vera e propria interfaccia per un dispositivo sia questo fisico che virtuale. Esistono varie tecniche per progettare un driver, tuttavia tutte hanno in comune il meccanismo del caricamento come modulo. Come già accennato il kernel Linux presenta uno spazio di memoria riservato, in questo spazio sono implementate tutte le sue funzionalità. Il concetto di modulo è legato indissolubilmente al tipo di kernel monolitico. L'esigenza comune per un normale dispositivo, sia questo destinato ad uso industriale che in ambito home-entertainment, è quella di interfacciarsi con una periferica esterna, che utilizzerà un certo mezzo fisico e un certo protocollo elettrico e software per comunicare. Un kernel monolitico per sua natura dunque dovrebbe mantenere sempre nel suo spazio di memoria tutto il codice di gestione driver di tutti i possibili dispositivi con cui può andare ad interfacciarsi. Questa soluzione risulta particolarmente inefficiente sopratutto quando il sistema deve avere una flessibilità molto grande in termini di quantità di dispositivi gestibili. La problematica appena descritta probabilmente potrebbe, in un primo momento apparire come secondaria per un dispositivo Embedded Linux poichè in sede di configurazione del kernel è possibile compilare tutto ciò che si ritiene necessario. In realtà la situazione nei sistemi Embedded Linux a mio parere richiede una considerazione più ampia.

Innanzitutto un'analisi della crescita dell'integrazione ad esempio delle memorie NVM NAND-Flash mostra una tendenza netta alla dimunuzione del rapporto $\frac{prezzo}{capacità}$ di queste ultime.

Quanto detto è possibile, a livello di processo produttivo, grazie all'introduzione di chip di memoria MLC e TLC oltre che alla riduzione delle dimensioni di canale dei MOS (attualmente 20nm).

Dunque questo dato ci induce a considerare che a livello economico i costi per avere grandi capacità su NAND-Flash non dovrebbero presentare criticità. Viceversa le memorie SDRAM risultano particolarmente costose se confrontate con memorie NAND Flash in termini di rapporto $\frac{prezzo}{capacità}$.

Queste considerazioni portano a sostenere che un kernel monolitico che non utilizzi moduli per supportare i propri device driver è sostanzialmente inefficiente non solo per quanto riguarda le prestazioni ma anche per quando riguarda i costi di mercato. Il sistema Embedded Linux su cui ho lavorato in azienda presenta caratteristiche hardware tali da consentire l'introduzione nel filesystem di un certo quantitativo di moduli anche se non tutti utilizzati, questo potrà evitare ricompilazioni frequenti del kernel Linux per adattarlo magari a un'esigenza che potrebbe sorgere in futuro.

I moduli kernel sono delle parti del kernel che possono essere caricate e scaricate dallo spazio di memoria attraverso i comandi:

- insmod </path_module_name/module_name.ko> <args>

  - inserisce il modulo passato come primo parametro e fornisce al modulo eventuali argomenti

- rmmod <module_name>

  - rimuove il modulo dallo spazio di memoria del kernel

Solitamente per caricare un modulo su un sistema Linux attualmnete viene utilizzato il comando modprobe che di fatto esegue un insmod gestendo però la lista delle eventuali dipendenze.

4.4.1 Driver gpio_irq

Per la compilazione o come nel nostro caso la cross compilazione di un modulo si possono eseguire varie strade. Le due più comuni tecniche di compilazione sono: creare un makefile proprio, modificare i makefile ed eventualmente i kconfig nei sorgenti del kernel Linux.

Per comodità nell'esempio di driver che presento utilizzerò un Makefile molto semplice che cross compilerà il modulo chiamato gpio_irq.

Algoritmo 13: Makefile per il modulo gpioirq

```
1 obj-m = gpio_irq.o
2
3                   all: make ARCH=arm CROSS_COMPILE=
                         arm-linux- -C /home/alberto/NFS/
                         lib/modules/2.6.39.1/build M=$(
                         PWD) modules
```

L'algoritmo 13 mostra il Makefile utilizzato per cross compilare il modulo gpio_irq, le variabili del make richiamato alla riga 3 specificano il tipo di architettura e il cross compilatore da utilizzatre per generare il modulo. Il modulo trattato in questo esempio è stato sviluppato per un microcontrollore differente rispetto a quello utilizzato nel tirocino. Il microprocessore su cui è stato sviluppato il modulo è precisamente un s3c6410 Samsung basato su ARM1176JZ(F)-S. L's3c6410 è dotato di una ricca dotazione multimediale per quanto riguarda i controllers offerti, dunque trova applicazione principalmente su smartphone, dispositivi tablet e navigatori.

Come visto precedentemente sono diversi i modi di implementare un driver come modulo:

- Può essere creato all'interno del procfs un file che permetta di gestire le funzionalità del driver

- Si può registrare il device driver e renderlo disponibile attraverso il filesystem in /dev

- Si può creare una struttura di file o un file singolo sul sysfs

Nell'esempio che andrò a mostrare ho scelto di registrare un device driver per utilizzarlo come file speciale all'interno della directory /dev.

Prima di passare al codice sorgente del driver descrivo cosa fa il driver e come funziona.

L'idea di base è mostrare la semplicità nell'integrazione di un driver dalle funzionalità custom in un sistema Linux based.

Il driver gpio_irq è sostanzialmente un esempio molto semplice di installazione di un Handler su una sorgente di interrupt esterna di un pin GPIO. L'handler installato sul pin GPIO scelto come sorgente di interrupt, imposta a sua volta alcuni GPIO come output e opera una commutazione del loro stato logico. Inoltre per mostrare un esempio di lettura di un file speciale e soprattutto della netta distinzione esistente tra kernel e user data segment ho implementato un counter che mostrerà il numero delle interrupt rilevate dalla sorgente esterna specifica. Il driver è stato realizzato sul kernel Linux 2.6.39.1 ovvero il più recente disponibile in versione stabile rilasciato in questo momento. Il kernel è stato da me cross compilato e configurato per gestire l'hardware della board su cui sono stati effettuati i vari test.

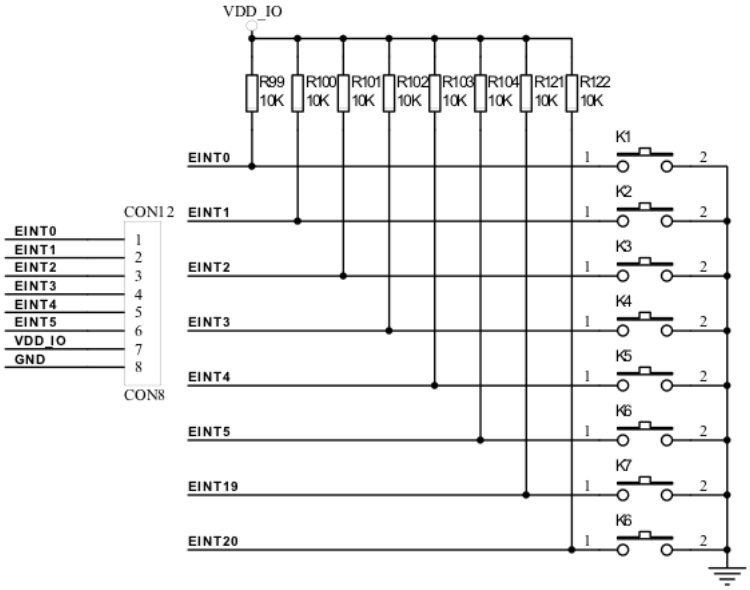


Figura 4.10: Schema elettrico Buttons

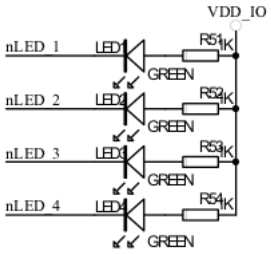


Figura 4.11: Schema elettrico Led

Le figure 24 e 25 mostrano lo schema elettrico di prova per il driver, attraverso l'interrupt generata verranno effettivamente pilotati i leds (figura 25). I leds risulteranno accessi quando il valore logico del GPIO corrispondente sarà portato basso sul catodo di ciascuno di essi.

Negli Algorotmi 14 e 15 riporto il codice sorgente del del Device Driver opportunamente commentato:

Algoritmo 14: File di intestazione

```
/* linux/arch/arm/mach-s3c64xx/include/mach/gpio-bank-k.h
 *
 * Author: Alberto Gambarucci
 *
 * GPIO Bank K register and configuration definitions
 *
 * This program is free software; you can redistribute it
     and/or modify
 * it under the terms of the GNU General Public License
     version 2 as
 * published by the Free Software Foundation.
*/

#define S3C64XX_GPKCON0                    (S3C64XX_GPK_BASE
     + 0x00)
#define S3C64XX_GPKCON1                    (S3C64XX_GPK_BASE
     + 0x04)
#define S3C64XX_GPKDAT                       (
    S3C64XX_GPK_BASE + 0x08)
#define GPKPUD
       (S3C64XX_GPK_BASE + 0x0C)


#define S3C64XX_GPC_CONMASK(__gpio)        (0xf << ((__gpio)
     * 4))
#define S3C64XX_GPC_INPUT(__gpio)          (0x0 << ((__gpio)
     * 4))
#define S3C64XX_GPC_OUTPUT(__gpio)         (0x1 << ((__gpio)
     * 4))
```

Algoritmo 15: File gpio_irq.c

```
/*
 * @platform: s3c6410
 *
 * @file: gpio_irq.c
 *
```

```
 * @author: Alberto Gambarucci <alberto.gambarucci@gmail.
    com>
 *
 * @brief Change the output state of GPK4, GPK5, GPK6,
    GPK7 on falling and rising edges
 *        on GPN0.
 *
*/

#include <linux/module.h>

#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/irq.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>
#include <mach/hardware.h>
#include <mach/map.h>
#include <mach/regs-gpio.h>
#include <plat/gpio-cfg.h>
#include <gpio-bank-k.h>

/*
 * Define debug symbol for developers
*/

//#define DEBUG


/*
 * Fuction prototypes
 */


int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t,
    loff_t *);
static ssize_t device_write(struct file *, const char *,
   size_t, loff_t *);
static irqreturn_t button_interrupt(int irq, void *dev_id
   );
```

```
/*
 * Generic define
*/


#define DEVICE_NAME "gpioirq"
#define SUCCESS 0 #define BUF_LEN 80


/*
 * Global variables
*/


static int Major; static int Device_Open = 0;
static char msg[BUF_LEN];
static char *msg_Ptr;

/*
 * file_operation enable possible file operations on this
     driver
*/
static struct file_operations fops =
{
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};

static int counter = 0; static unsigned irq_run=0;


/*
 * This function is called when the module is loaded
*/


int init_module(void)
        {
                Major = register_chrdev(0, DEVICE_NAME, &
                    fops);

                if (Major < 0)
                {
                printk(KERN_ALERT "Registering char
                    device failed with %d\n", Major);
                    return Major;
                }
```

```
        printk(KERN_INFO "Assigned major number %d. \n
            Create special file device for use the driver!
             \n >> mknod /dev/gpioirq c %d 0 << \n", Major
            ,Major);

        return SUCCESS;
}


/*
 * This function is called when the module is unloaded
*/


void cleanup_module(void)
{

        /*
         * Free irq if installed prevent kernel errors
        */

        if(irq_run)
        {
                free_irq(IRQ_EINT(0), NULL);
        }

        /*
         * Unregister the device
        */
        unregister_chrdev(Major, DEVICE_NAME);
}


/*
 * Interrupt handler for button 0
 *
*/
static irqreturn_t button_interrupt(int irq, void *dev_id
    )
{

/*
 * Set output for all led
*/

unsigned regcon0 = readl(S3C64XX_GPKCON0);
regcon0 = (regcon0 & 0x0000FFFF) | 0x11110000;
writel(regcon0, S3C64XX_GPKCON0);
```

```
/*
 * Change led status values
*/


unsigned tmp = readl(S3C64XX_GPKDAT);
tmp = (tmp & 0xFF0F) | (~tmp & 0x00F0) ;
writel(tmp, S3C64XX_GPKDAT);

#ifdef DEBUG
unsigned Gpk4_val = (tmp >> 4) & 0x1;
unsigned Gpk5_val = (tmp >> 5) & 0x1;
unsigned Gpk6_val = (tmp >> 6) & 0x1;
unsigned Gpk7_val = (tmp >> 7) & 0x1;
printk(KERN_ALERT "S3C64XX_GPKDAT value= %X.\n",tmp);
printk(KERN_ALERT "Gpk4_val value= %X.\n",Gpk4_val);
printk(KERN_ALERT "Gpk5_val value= %X.\n",Gpk5_val);
printk(KERN_ALERT "Gpk6_val value= %X.\n",Gpk6_val);
printk(KERN_ALERT "Gpk7_val value= %X.\n",Gpk7_val);
#endif

counter++;

return IRQ_RETVAL(IRQ_HANDLED);

}

/*
 * This function is called when the special file is
    opened
*/

static int device_open(struct inode *inode, struct file *
   file)
{
        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        sprintf(msg, "You have generated %d interrupts!\n
            ", counter);
        msg_Ptr = msg;
        try_module_get(THIS_MODULE);
        return SUCCESS;

}
```

```
/*
 * This function is called when the special file is
    closed
*/
static int device_release(struct inode *inode, struct
   file *file)
{

        Device_Open--;

        /*
         * Decrement the usage count, or else once you
            opened the file, you'll
         * never get get rid of the module.
        */

        module_put(THIS_MODULE);
        return 0;
}

/*
 * This function is called when the special file is read
*/

static ssize_t device_read(struct file *filp, char *
   buffer, size_t length, loff_t * offset)
{

        int bytes_read = 0;

        /*
         * If we're at the end of the message,
         * return 0 signifying end of file
        */

        if (*msg_Ptr == 0)
                return 0;
        /*
         * Actually put the data into the buffer
        */

        while (length && *msg_Ptr)
        {

                /*
                 * The buffer is in the user data segment
                    , not the kernel
                 * segment so "*" assignment won't work.
```

```
                    We have to use
                 * put_user which copies data from the
                    kernel data segment to
                 * the user data segment.
                */

                put_user(*(msg_Ptr++), buffer++);
                length--;
                bytes_read++;

        }

        /*
         * Most read functions return the number of bytes
             put into the buffer
        */

        return bytes_read;
}

/*
 * This function is called when the special file is
    written
*/


static ssize_t device_write(struct file *filp, const char
    *buff, size_t len, loff_t * off)
{

        /*
         * Valid string for install/uninstall the module
        */

        const char in[] = "install\0";
        const char di[] = "uninstall\0";
        char st[10]; memset(st,'\0',sizeof(st));

        if(sizeof(st) < len)
        {

                printk(KERN_ALERT "Sorry, string too long
                    .\n");
                return -EINVAL;

        }

        memcpy(st,buff, len);
```

```
#ifdef DEBUG
printk(KERN_ALERT "st:%s\n",st);
printk(KERN_ALERT "buff:%s\n",buff);
printk(KERN_ALERT "len:%d\n",len);
#endif


/*
 * Use this command "echo -n -e "install\0" > /
    dev/gpioirq" in shell to test module and
    install irq
*/

if( strncmp( st, in, sizeof(in) ) == 0)
{

        printk(KERN_ALERT "Inizializing IRQ.\n");

        request_irq(IRQ_EINT(0), button_interrupt
            , IRQ_TYPE_EDGE_BOTH, "button_irq",
           NULL); irq_run=1;

        return EINVAL;

}


/*
 * Use this command "echo -n -e "uninstall\0" > /
    dev/gpioirq" in shell to test module and
    uninstall irq
*/


if( strncmp ( st, di, sizeof(di) ) == 0)
{

        printk(KERN_ALERT "Uninstall IRQ.\n"); if
            (irq_run) free_irq(IRQ_EINT(0), NULL);
             irq_run=0;

        return EINVAL;

}

printk(KERN_ALERT "No valid string found!\n");

return -EINVAL;
```

```
}

MODULE_LICENSE("Proprietary");

MODULE_AUTHOR("Alberto Gambarucci");

MODULE_DESCRIPTION("GPIO example driver");
```

Ci sono diversi punti del codice che meritano un commento più approfondito. Innanzitutto l'impiego di un file di intestazione personalizzato, inserito direttamente negli include del kernel necessario per avere a disposizione i define del memory mapped I/O usato per gestire il bank k GPIO.

La struttura del driver è abbastanza semplice sono implementate alcune funzioni comuni a quasi tutti i device driver e sono relativamente pochi i punti del codice specifici per l'architettura utilizzata.

- Funzioni comuni di un device driver
  - int init_module(void);
    - E' la funzione chiamata quando il modulo viene inserito
  - void cleanup_module(void);
    - E' la funzione chiamata al momento della rimozione del modulo
  - static int device_open(struct inode *, struct file *);
    - Viene richiamata quando si tenta di aprire il file speciale specificato da Major e Minor number assegnati per il device
  - static int device_release(struct inode *, struct file *);
    - Viene richiamata quando si chiude il file speciale specificato da Major e Minor number assegnati per il device
  - static ssize_t device_read(struct file *, char *, size_t, loff_t *);
    - Funzione richiamata al momento della lettura del file speciale specificato da Major e Minor number assegnati per il device
  - static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
    - Funzione richiamata al momento della scrittura del file speciale specificato da Major e Minor number assegnati per il device

Chiaramente la parte di codice specifica per l'architettura è quella che comprende tutte le operazioni sui registri.

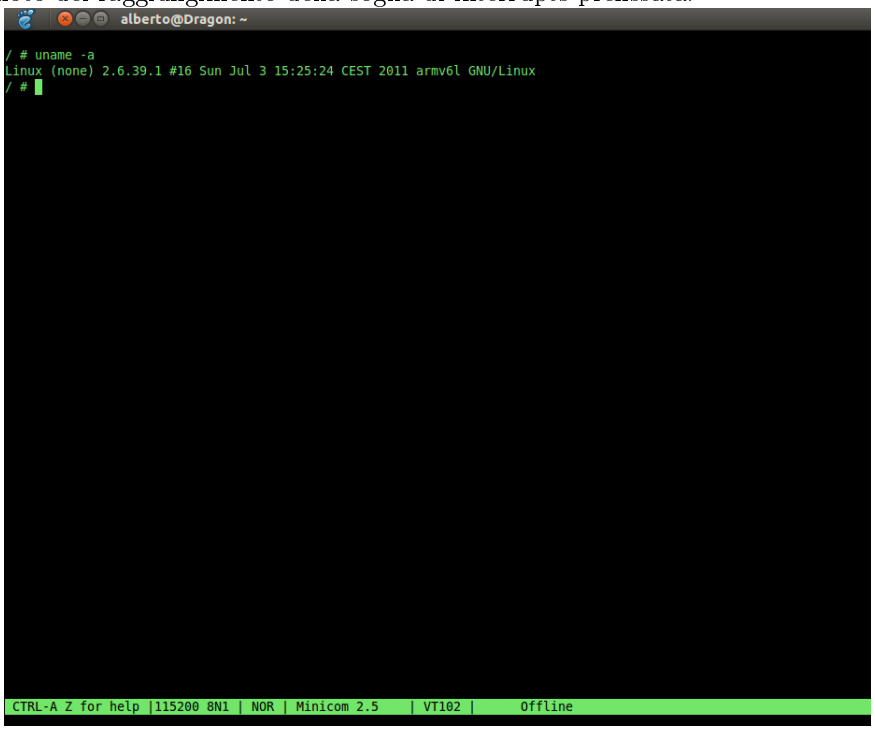
Figura 4.12: Shell sistema Embedded Linux

4.4.2 Uso del gpio_irq driver

In questo sottoparagrafo mostrerò un esempio pratico di utilizzo del driver integrandolo nello user space con un semplice shell script per dimostrarne la duttilità del sistema.

L'interrupt nel nostro caso comanda dei led sulla board di sviluppo, consideriamo di voler lanciare un certo comando dopo un numero di interrupt prefissato ovvero dopo un certo quantitativo di blink dei leds.

Inoltre supponiamo di voler trasmettere un trigger che avvisi un dispositivo remoto del raggiungimento della soglia di interrupts prefissata.



Nella figura 26 mostra come si presenta la shell del sistema Embedded di sviluppo su cui è stato implementato il driver. La fase di sviluppo sul driver è stata effettuata utilizzado un mount NFS come filesystem, mentre per la programmazione ho preferito utilizzare un'inizializzazione minimale su filesystem initramfs che possa permetere un collegamento LAN.

Nel filesystem ho cross compilato e installato BusyBox così da avere a disposizione un certo quantitativo di tool e applicativi.

La shell (sh) permette di poter creare scripts estremamente semplici e indubiamente efficaci, di seguito l'algoritmo 16 mostra lo script che può essere utilizzato per gestire il driver gpioirq e inviare ad un dispositivo in rete un certo comando via LAN attraverso il tool software nc.

Algoritmo 16: Script sh che gestisce il Driver gpioirq

```
#!/bin/sh

# remote args

REMOTE_IP=192.168.0.3

REMOTE_PORT=6666

# variables declaration

MODULE_PATH="/gpio_irq.ko"

N_TRIGGER=10

# load gpio_irq module only if really need it

TEMP=`cat /proc/modules | grep "gpio_irq"`

if [ "$TEMP" == "" ] then

        `insmod "$MODULE_PATH"`
fi

MAJOR_N=`cat /proc/devices | grep "gpioirq" | awk {'print
    $1'}`

# create special file to manage gpioirq driver

`mknod /dev/gpioirq c "$MAJOR_N" 0`

# install irq

`echo -n -e "install\0" > /dev/gpioirq`

# main loop

while [ "$C" == "" ]
do
        IRQ_N=`cat /dev/gpioirq | awk {'print $4'}`
        if [ "$IRQ_N" -ge "$N_TRIGGER" ]
        then

                # send remote message

                DATE=`date +%d/%m/%y-%H:%M:%S`

                `echo "$DATE trigger!" | nc "$REMOTE_IP"
                    "$REMOTE_PORT"`
```

```
        `rmmod gpio_irq`

        `rm /dev/gpioirq`

                exit
    fi

        sleep 1
done
```

Il codice sh mostrato nell'algoritmo 16 permette di eseguire di fatto un controllo remoto sull'interrupt GPIO anche a distanza attraverso la rete locale lan o attraverso il web. Se consideriamo l'integrazione di un tale sistema con un web server o in generale con un'interfaccia utente si riesce subito ad intuirne la flessibilità.

Gli screenshoot in figura 27 mostrano l'output del mini progetto sviluppato, composto da driver e scripts.

Il dispositivo Embedded è connesso attraverso un cavo cross al computer host. Il computer host è in ascolto sulla porta 6666.

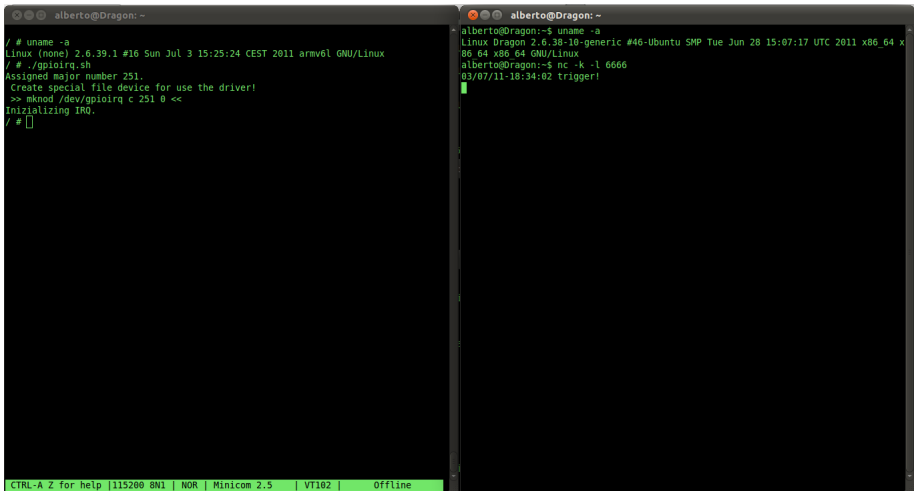


Figura 4.13: A sinistra la shell del sistema Embedded Linux, A destra la shell del computer host

In conclusione l'esempio mostrato non ha sostanzialmente un'utilità specifica il suo scopo è puramente valutativo. Tuttavia è un ottimo esempio per spiegare le potenzialità nell'integrare linguaggi a basso livello (C assembly) e linguaggi scripting. La piattaforma Embedded Linux ha dunque dimostrato come sia possibile un livello di programmazione del sistema altamente flessibile. L'open source ed in particolare i sistemi Linux riescono a fornire dunque soluzioni ad

alto livello senza perdere tuttavia la possibilità di personalizzazione e sviluppo ad ogni livello dello strato software.

Capitolo 5

Supporto NVM Flash

Ho più volte fatto riferimento all'impiego all'interno dei sistema Embedded Linux delle memorie NAND-FLASH, senza tuttavia entrare nei particolari hardware e sopratutto di gestione software di tale dispositivo.

In questo capitolo tratterò alcuni aspetti peculiari delle memorie NAND Flash a livello hardware e successivamente, dopo aver introdotto alcuni concetti base, spiegherò come Linux gestisce tali dispositivi ponendo particolare attenzione alla scelta del filesystem utilizzato nel dispositivo Embedded sviluppato in azienda.

Innanzitutto è bene comprendere la terminologia che verrà adottata, precisando in particolare che esiste una certa ambiguità nel termine filesystem. Sostanzialmente quando si parla di filesystem si possono intendere almeno due concetti, di fatto nettamente distinti:

- Per Filesystem si può intendere la struttura gerarchica composta da directory e sottodirectory presenti in un sistema. A tale proposito per evitare ambiguità spesso in ambiente Linux si parla di Rootfs per indicare appunto l'albero composto da tutte le directory e file di un certo sistema.

- Filesystem in ambito informatico è da intendere come una strategia di organizzazione dei contenuti su un certo supporto di memoria. In questo caso si tratta quindi di un livello di astrazione che comprende le regole e i meccanismi per gestire i dati in memoria.

Nonostante l'ambiguità appena riscontrata, solitamente sarà molto semplice in base al contesto capire in che accezione venga utilizzato il termine Filesystem.

5.1 Funzionamento fisico delle memorie Flash

La tecnologia utilizzata nella costruzione delle memorie Flash è sostanzialmente basata sull'impiego di un array di Floating Gate MOSFET, una tipologia di transistor ad effetto di campo in grado di mantenere la carica elettrica per un tempo lungo. Ogni transistor costituisce una "cella di memoria" che conserva il valore di un bit nelle implementazioni SLC, mentre nelle nuove Flash spesso si utilizzano delle celle multilivello, che permettono di registrare il valore di più bit attraverso un solo transistor (MLC-TLC).

Le memorie Flash uniscono alcune delle caratteristiche delle memorie EPROM ed EEPROM, espongo di seguito brevemente il loro principio di funzionamento.

Nel corso degli anni si è cercato di trovare un particolare componente che potesse in qualche modo conservare il suo stato anche in assenza di alimentazione. Il componente che si è imposto in questo settore è il così detto FAMOS ovvero il *floating-gate avalanche-injection MOS*. Il principio di funzionamento del floating gate MOS è abbastanza semplice, fisicamente è molto simile a un normale MOS a differenza di quest'ultimo però presenta l'inserimento di un sottile strato (circa 100nm) di polisilicio tra il gate e il canale. Il polisilicio permette un isolamento tra il canale e il gate, tale soluzione ha diversi risvolti fisici nelle caratteristiche del MOS:

- Aumenta il t_{ox} ovvero lo spessore dell'ossido di gate.
- Diminuisce la capacità $C_{ox} = \frac{e_{ox}}{t_{ox}}$ dove e_{ox} è la permettività dell'ossido e vale e_0 x 3.97 = 3.5x10^{-11}F/m.
- Diminuisce la transconduttanza di processo k'$_n$ e di conseguenza la corrente I_D di drain.
- Aumenta la tensione di soglia V_T

Questi effetti solitamente vengono considerati negativi e parassiti, tuttavia nel caso dei FAMOS vengono accettati, poichè l'obbiettivo è di avere un dispositivo che mantenga uno certo stato anche se non alimentato. In figura 28 vediamo un transistor FAMOS, si nota l'ossido SiO_2 tra control-gate e floating-gate.

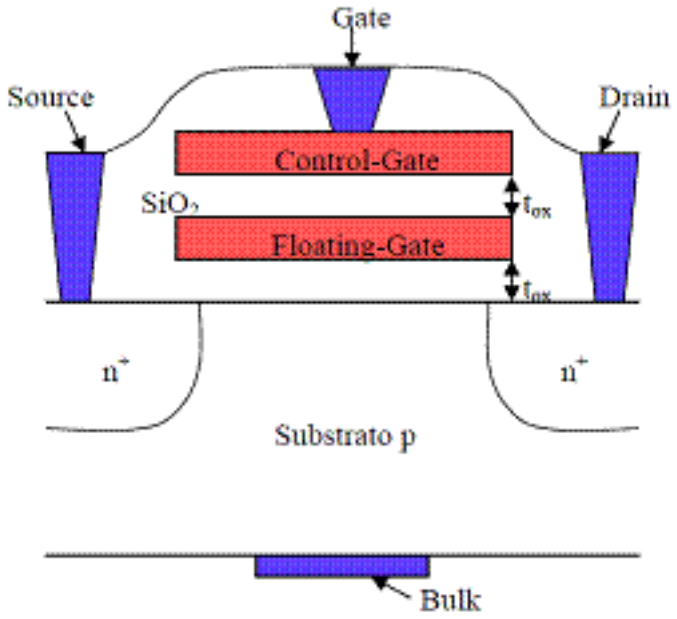


Figura 5.1: Transistor floating-gate

Il primo gate, detto control gate, è un gate tradizionale, il secondo, chiamato floating gate, è immerso in uno strato di polisilicio, isolato tramite l'ossido di silicio e accoppiato capacitivamente con l'elettrodo relativo al gate di controllo. Il dato binario è rappresentato dalla presenza o meno di una carica di elettroni nel floating gate. La soglia del MOS cambia a seconda della carica immagazzinata. In una memoria Flash durante la programmazione si controlla l'iniezione a valanga di carica nel Floating-Gate in maniera da variare la tensione di soglia V_T della cella e provocare uno spostamento della caratteristica corrente-tensione I_{DS}/V_{GS} del transistor.

- L'assenza di carica nel Floating-Gate viene associata al livello logico "1" (transistor spento = cella cancellata)
- La presenza di carica viceversa viene associata al livello logico "0" (transistor acceso = cella programmata).

Il principio fisico su cui si basa la fase di programmazione del FAMOS è l'iniezione a valanga di elettroni caldi "hot electron" che consiste in un processo energetico che fornisce agli elettroni energia sufficiente per superare la barriera di potenziale che nasce tra Floating-Gate, ossido sottile e Bulk. Per effetto della programmazione della cella di memoria la soglia V_T del transistore viene portata a valori tali da non permettergli la conduzione.

Per la cancellazione invece viene sfruttato l'effetto tunnel Fowler-Nordheim.

5.1.1 Processo di programmazione

La programmazione avviene come già accennato attraverso l'iniezione di cariche nel gate flottante. Le tensioni applicate ai morsetti di gate e drain regolano i campi elettrici nella zona in prossimità del drain in cui è maggiore la presenza di portatori caldi (ossia più energetici) ed è più probabile l'iniezione di questi verso il gate. Tale potenziale può favorire o meno un'iniezione di portatori di carica (elettroni o lacune) attraverso l'ossido. Per permettere l'iniezione delle cariche libere del canale verso il gate sono necessarie più condizioni. Per prima cosa è indispensabile una significativa popolazione di portatori con valori di energia elevati e superiori, nel caso degli elettroni, ai 3.1 eV della barriera nell'ossido. Affinchè questo accada il campo elettrico laterale deve raggiungere intensità elevata. Inoltre la tensione di gate non deve essere così elevata da portare il transistore in zona lineare, ma neppure così bassa da rendere la popolazione di cariche libere nel canale troppo esigua per produrre correnti misurabili. Con l'applicazione di una tensione di polarizzazione sul Control-Gate, viene creato attraverso l'ossido un campo elettrico elevato, che permette agli elettroni di guadagnare energia, divenire "caldi", e superare la barriera di ossido, causando l'iniezione a valanga dei portatori, che vengono intrappolati nel polisilicio del gate flottante, che essendo isolato può conservare tale carica per un tempo infinitamente lungo. La programmazione di una cella viene quindi realizzata applicando una tensione elevata ai terminali di gate e di drain, mentre il terminale di source viene mantenuto a massa. Gli elettroni, divenuti caldi ed intrappolati nel polisilicio del gate flottante, vanno a ridurre il potenziale dell'elettrodo. Questo meccanismo è quindi un processo autolimitato in quanto la carica negativa accumulata nel gate flottante riduce il campo elettrico e di conseguenza è più difficile rendere "caldi" e iniettare altri elettroni. Essendo il biossido di silicio, SiO_2, un ottimo isolante, la carica intrappolata nella gate flottante può essere mantenuta per molti anni e senza che il circuito venga alimentato. La scrittura della cella per Channel-Hot-Electrons può portare all'intrappolamento di elettroni ad alta energia nell'ossido sovrastante il drain e ciò impedisce l'ulteriore accumulazione di elettroni nel Floating Gate, provocando un aumento della tensione di soglia ed un aumento del tempo di scrittura.

5.1.2 Processo di cancellazione

Secondo il fenomeno quantistico del tunnelling Fowler-Nordheim, gli elettroni con contenuto energetico inferiore alla barriera di potenziale possono attraversare l'ossido di silicio purché questo sia sufficientemente sottile e siano verificate opportune condizioni di campo elettrico. La cancellazione viene realizzata mantenendo il gate a massa e applicando al source una tensione elevata, che permette agli elettroni di uscire dal gate flottante ed iniettarsi nel source, grazie al meccanismo di tunnel Fowler-Nordheim. I tempi tipici per l'esecuzione dell'operazione di cancellazione variano da 100 ms a 1 s. Durante la cancellazione, la giunzione di source, sotto l'ossido, si piega per azione del campo elettrico verticale e si formano coppie elettrone-lacuna per tunnel di elettroni; gli elettroni fluiscono nel source mentre le lacune nel subtrato, ma non tutte, alcune guadagnando energia per azione del forte campo elettrico verticale, passano la barriera di potenziale dell'ossido e si intrappolano nell'ossido stesso. Queste

cariche positive intrappolate nell'ossido rendono più lenta la cancellazione e degradano la cella. L'intrappolamento di cariche nell'ossido per Band-to-Band Tunnel (BBT), durante la cancellazione, o l'intrappolamento di cariche nell'ossido per Channel-Hot-Electron(CHE) producono problemi macroscopici come la perdita di dati (Data Retention) per degradazione della qualità dell'ossido e problemi di Endurance, cioè resistenza della cella ponendo un limite al numero di cicli di scrittura/cancellazione.

5.2 Considerazioni ed applicazioni sui sistemi Embedded delle memorie NAND Flash

La tecnologia Flash appena introdotta viene implementata in due distinte tipologie di memorie dette rispettivamente NOR Flash e NAND Flash. Le due tipologie di memorie si differenziano non solo per il tipo di disposizione logica dell'array di transistors ma anche per la gestione interna del dispositivo.

La Flash memory a differenza di una EEPROM non permette di indirizzare singoli bit, ma viene scritta a blocchi (pages) più o meno come un disco fisso. Questa scelta tuttavia accelera il processo naturale di deterioramento dell'isolante che assicura ai transistor la loro proprietà di mantenere lo stato per lungo tempo, e accorcia di molto la longevità delle Flash rispetto alle EEPROM in termini di numero di cicli di cancellazione sopportabili (dell'ordine del milione di cicli per le Flash memory). In compenso, le Flash memory risultano molto più economiche da fabbricare delle EEPROM e hanno riscosso uno straordinario successo, come memorie di massa per tutti i tipi di dispositivi digitali.

Esistono due tipi di Flash memory, a seconda del tipo di porta logica usata per implementare una cella da un bit: i chip NOR e i chip NAND. I chip NAND, i più usati, hanno migliori prestazioni, maggior densità, minor costo e maggior durata ma, a differenza dei NOR, non si prestano bene ad accessi non sequenziali. Pertanto, nelle applicazioni in cui il chip Flash deve ospitare un programma da eseguire direttamente senza trasferirlo prima in RAM, è necessario usare Flash memory basate su chip NOR. Nel sistema Embedded sviluppato durante il tirocinio una Flash memory NAND è utile per conservare i dati relativi al Filesystem, tuttavia diventa necessario utilizzare una Flash NOR ad esempio per la creazione di un ROMBOOT esterno. Dalla panoramica fin qui esposta non emerge tuttavia il problema fondamentale per un sistema Embedded, ovvero la capacità di gestire eventuali perdite di dati.

Una memoria solitamente ha un certo quantitativo di celle che possono per varie ragioni (alcune descritte nel paragrafo 5.1) deteriorarsi e divenire di fatto inutilizzabili. In un dispositivo NAND Flash le operazioni sulla memoria vengono fisicamente gestite attraverso una serie di segnali inviati al chip di memoria di 8 o 16 bit. Tali segnali seguono una logica tale da poter essere riconosciuti come sequenze di dati, indirizzi o appunto comandi. Solitamente per dispositivi Embedded il modulo di memoria NAND Flash più utilizzato è con package tipo TSOP1-48, nella figura 29 è raffigurato il package TSOP1-48 della memoria K9K8G08U0A Samsung, sono raffigurati i vari segnali presenti sul chip.

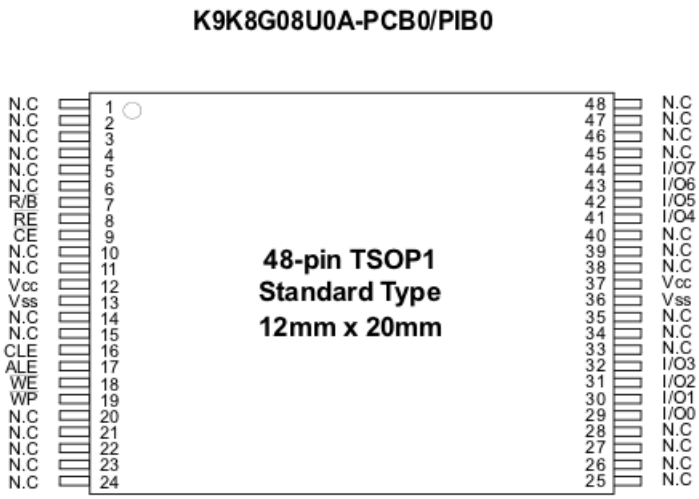


Figura 5.2: Package TSOP1-48 di una memoria NAND Flash Samsung

I principali segnali mostrati in figura 29 sono:

- DATA INPUTS/OUTPUTS:
  - I pin I/O sono utilizzati come input dei comandi, degli indirizzi e dei dati e come output per i dati durante la lettura. Tali pin rimangono flottanti ad alta impedenza quando il chip non è selezionato CE oppure quando l'output viene disabilitato.
- COMMAND LATCH ENABLE:
  - Il segnale CLE di input controlla la possibilità di inviare comandi al command register del chip di memoria. Quando è attivo alto i comandi vengono valutati e salvati nel command register attraverso le porte di I/O sul rising edge del segnale WE.
- ADDRESS LATCH ENABLE:
  - Il segnale ALE di input controlla la possibilità di inviare comandi all'address register del chip di memoria. Quando è attivo alto gli indirizzi vengono valutati e salvati nell'address register attraverso le porte di I/O sul rising edge del segnale WE.
- CHIP ENABLE:
  - Il segnale CE è un input che controlla la selezione del device. Esiste la possibilità di avere anche un secondo CE questo poichè in linea di principio una Flash NAND potrebbe avere al suo interno due chip fisicamente separati.

- READ ENABLE:
  - Il segnale RE di input controlla i dati seriali di output. I dati vengono considerati validi dopo un certo tempo T_{REA} successivo al falling edge del segnale RE, viene inoltre incrementato automaticamente di uno il counter che conta l'indirizzo di colonna interna del chip.
- WRITE ENABLE:
  - Il segnale WE di input controlla la scrittura dalle porte I/O. Comandi indirizzi e dati sono campionati sul rising edge del segnale WE (nelle sue pulsazioni).
- WRITE PROTECT:
  - Il pin WP di input quando attivo basso protegge la memoria da operazioni di programmazione o cancellazione durante le transizioni elettriche. Viene di fatto disattivata la possibilità di generare le tensioni di programmazione e cancellazione.
- READY/BUSY:
  - Il segnale R/B di output indica lo stato del chip durante le sue operazioni. Quando tale segnale è basso viene segnalata la situazione di busy, mentre se alto segnala che le operazioni interne della memoria sono completate si ha quindi lo stato di ready.

Infine abbiamo i pin dedicati ad alimentazione e ground rispettivamente V_{cc} e V_{ss}.

Per comprendere come vengono gestiti eventuali problemi hardware dal layer software è importante fissare in modo chiaro alcuni concetti riguardanti l'architettura interna di una memoria NAND Flash.

Ogni memoria è divisa in delle sotto unità che possiamo considerare il minimo numero di byte che possono essere letti o scritti. Tali unità di memoria vengono chiamate page ogni page ha la medesima dimensione solitamente 512 o 2048 byte. Un insieme di page forma un block o eraseblock. Come la prima suddivisione relativa alle page anche la divisione in eraseblock viene fatta poichè tale quantità di byte rappresenta la minima quantità di memoria in questo caso cancellabile, da qui appunto il nome eraseblock.

La suddivisione appena introdotta porta a diverse riflessioni riguardo la presenza di celle di memoria danneggiate, infatti per quanto detto non è più opportuno parlare di celle di memoria non funzionanti, ma più correttamente di bad eraseblock ovvero blocchi danneggiati. Ogni produttore di NAND Flash solitamente si preoccupa di segnalare i bad eraseblock marcandoli in un modo specifico, così che un eventuale software sia in grado di riconoscerli ed eviti dunque di effettuare operazioni sulle page di tali eraseblocks.

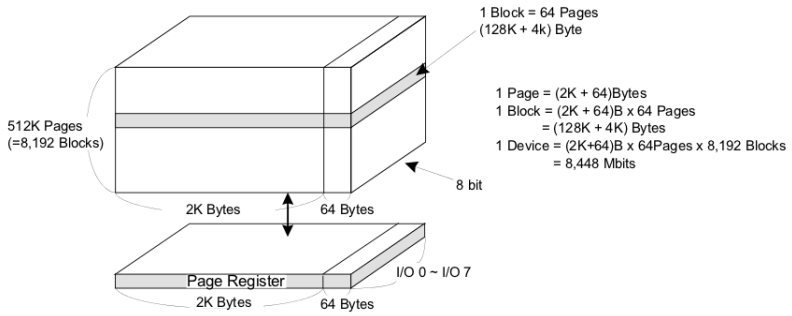


Figura 5.3: Suddivisione in page ed eraseblock della memoria K9K8G08U0A Samsung

La figura 30 mostra l'organizzazione della memoria interna di un particolare chip di memoria NAND Flash, si nota in questo una page size di 512 byte, un erase size di 2048 byte e infine una device size di 8448 Mbits corrispondente ad 8192 blocchi.

Tornando all'ambiente Linux le memorie Flash vengono gestite come dei device MTD che non rappresentano ne un tipo a blocchi ne un tipo a caratteri.

Memory technology devices (MTD) è un tipo di device Linux che si intefaccia con le memorie Flash è sostanzialmente un abstraction layer tra hardware e software.

MTD è in grado di gestire correttamente tutti i tipi di Flash su Linux: NAND, OneNAND, NOR, AG-AND,NOR, etc.

E' importante far notare che il subsystem MTD non gestisce i device MMC, eMMC, SD, CompactFlash. Tali device non sono delle vere e proprie NAND Flash, infatti questi possiedono un Flash Traslation Layer interno che li rendono gestibili come device a blocchi.

Ci si potrebbe chiedere il perchè Linux gestisca le Flash attraverso un tipo di device MTD e non uno tradizionale a blocchi o a caratteri. Il problema principale è che la descrizione di block device e character device non è esaustiva in questo caso. Ad esempio se consideriamo le operazioni di lettura e scrittura queste vengono eseguite in modo non omogeneo, dunque non seguono la logica a block device, tuttavia anche la logica a character device è inconsistente. Per i motivi appena citati si è reso necessario introdurre un tipo di device completamente nuovo. Fino adesso abbiamo considerato il layer più basso di traduzione di un dispositivo Flash in un sistema Linux, tuttavia il device sia questo MTD che a blocchi nel caso di normali Hard Disk si occupa quasi esclusivamente di un interfacciamento tale da consentire la gestione delle operazioni di base sull'hardware. Un moderno sistema operativo possiede come caratteristica fondamentale un certo Filesystem per la propria partizione su cui è montata la radice principale delle directory, ed eventualmente per le partizioni che gestiscono dati. Il filesystem realizza due livelli di astrazione, che rendono le risorse di memorizzazione di massa facilmente utilizzabili dagli utenti:

1. Il primo livello di astrazione è quello che organizza i settori in un insieme di archivi (file) di dimensioni arbitrarie, che possono andare da zero all'in-

tera dimensione disponibile del dispositivo: ciascun file viene distribuito in un insieme di settori. Normalmente l'utente vede solo un file e non deve preoccuparsi di quali settori sono stati utilizzati per memorizzarlo. Le operazioni disponibili sono la lettura o la scrittura di un blocco di dati di dimensione arbitraria in un punto arbitrario del file. Il software di gestione del filesystem è responsabile dell'organizzazione di questi settori in file e di tenere traccia di quali settori appartengono a quali file e quali settori invece non sono utilizzati. L'utente ha normalmente la totale libertà di creare nuovi file, cancellare file esistenti (liberando così i blocchi che questi occupavano), modificare file esistenti (cambiando così anche la loro dimensione e quindi il numero di blocchi occupati).

2. Il secondo livello di astrazione è quello che permette di organizzare i file assegnando loro dei nomi gerarchici. I filesystem tipicamente hanno tabelle di associazione dei nomi dei file con i rispettivi file tramite collegamenti fisici, usualmente collegando il nome del file a un indice in una tabella di allocazione dei file (file allocation table) di qualche genere, come un inode in un filesystem di tipo Unix. Le strutture di cartelle possono essere a singolo livello, oppure possono permettere una struttura annidata di sottocartelle multilivello.

Tutta l'infrastruttura descritta fa parte dei compiti svolti dal Filesystem. Nel caso di un sistema Embedded Linux basato su un supporto NVM Flash i filesystem più comunemente utilizzati sono: JFFS2, YAFFS2 e UBIFS. Il più recente è sicuramente l'UBIFS ed il suo supporto è nativo per Linux, ovvero è possibile utilizzarlo semplicemente configurando opportunamente il kernel senza utilizzo di patch esterne.

Attualmente i filesystem più usati supportati nativamente per la gestione delle memorie Flash su Linux sono JFFS2 e UBIFS.

JFFS2 Journalling Flash File System version 2, è un filesystem log-structured detto LFS. I filesystem LFS presentano una serie di innovazioni rispetto ai tradizionali filesystem. La differenza più importante è che mentre i filesystem classici scrivono i file su disco cercando i blocchi o nel nostro caso le pages tra quelle al momento disponibili, LFS scrive sempre tutti i blocchi in un dato momento in posizioni adiacenti, indipendentemente dal file di cui fanno parte. In questo modo lo stesso blocco di un file, scritto in momenti diversi, esisterà sul disco in posizioni diverse. Ciò permette la creazione sicura ed asincrona del file inoltre il vecchio indice rimane tra i dati della directory che lo contiene anche in caso di crash.

Dunque si hanno due vantaggi con un design LFS:

1. una più veloce scrittura sul supporto (tutti i blocchi sono scritti insieme, senza necessità di trovare un posto libero).

2. un recupero istantaneo in caso di arresto del sistema (il filesystem ricomincia dall'ultimo punto di controllo e prosegue, invece di dover essere controllato nella sua totalità per verificarne la consistenza).

La tecnica di gestione del supporto Flash attuata da JFFS2 si basa essezialmente sul concetto che ogni nodo deve essere esattamente contenuto in uno specifico eraseblock. Se un nodo si trova al confine di due eraseblock questo viene di fatto

ignorato causando la perdita dei dati relativi la nodo. Durante la scrittura di un nodo complesso se avviene uno sforamento dell'eraseblock, JFFS2 provvede saltando il block e passando a quello successivo. Infine ogni blocco viene scritto dall'inizio e non devono esistere spazi vuoti all'interno, tuttavia una situazione di questo tipo con un eraseblock scritto solo alle estremità non provoca particolari problemi se non per lo spreco di un certo quantitativo di memoria.

La situazione di spazio vuoto all'interno del blocco viene segnalata con un messaggio in fase di scrittura di questo tipo:

- *jffs2_scan_empty(): Empty block at 0x0012fffc ends at 0x00130000 (with 0xe0021985)! Marking dirty!*

JFFS2 nonostante le ottime caratteritiche appena illustrate di fatto attualmente è quasi completamente superato. Nel caso del sistema Embedded Linux sviluppato in azienda l'esigenza di ottenere un filesystem robusto e veloce ha fatto orientare la scelta su UBIFS di cui parleremo nel paragrafo seguente.

5.3 UBI e UBIFS

Il filesystem UBIFS è come nel caso di JFFS2 di tipo log-structured ed eredita duque tutti i vantaggi di questa scelta di design. A differenza di JFFS2 tuttavia possiamo notare immediatamente la composizione fatta da 2 diversi layer. Il primo chiamato UBI "Unsorted Block Images" che è un sistema di gestione di volumi per Flash memory. Il secondo UBIFS è il layer che si occupa della gestione dei file e della loro integrità, si tratta quindi del vero e proprio filesystem.

5.3.1 UBI layer

UBI è in grado di gestire volumi multipli sul singolo Flash device, inoltre ha un importante caratteristica di dividere e distribuire il livello di usura fisica sull'intero chip Flash in modo uniforme. UBI può essere paragonato a un LVM Logical Volume Manager con la sostanziale differenza che in questo caso sono mappati logical eraseblocks (LEB) in physical eraseblocks (PEB). Un volume UBI non è altro dunque che una serie di eraseblocks logici (LEBs) consecutivi. Ogni eraseblock logico può essere associato a qualsiasi eraseblock fisico(PEB). Questa mappatura viene gestita da UBI, in modo del tutto trasparente all'utente ed è il meccanismo di base per fornire un wear-leveling globale. Di fatto UBI è in grado di valutare lo stato di usura contando le cancellazioni di un blocco, di conseguenza gestisce la mappatura dei LEB. I volumi UBI possono essere creati attraverso alcuni tool utente nel caso di un sistema Embedded generalmente vengono creati fuori dal dispositivo su un computer Host.

Esistono due tipi di volumi:

- Volumi statici
- Volumi dinamici

I volumi statici sono di tipo read-only e sono protetti da CRC-32 (cyclic redundancy check).

I volumi dinamici invece sono di tipo read-write e il loro layer superiore sarà dunque un filesystem che gestirà l'integrità dei dati (UBIFS).

UBI identificando i bad eraseblocks libera il layer superiore ovvero il filesystem dalla necessità di controllo dei blocchi danneggiati. UBI confina i bad eraseblocks fisici, per cui quando un eraseblock fisico si corrompe, in modo del tutto trasparente UBI lo sostituisce con un good eraseblock fisico. UBI si occuperà dello spostamenteo dei dati dal bad block fisico al good block fisico. Il risultato di questo tipo di gestione è la completa assenza di messaggi di errori di I/O inviati all'utente in caso di corruzione di eraseblocks. Un altro problema che può incorrere su un modulo di memoria Flash è il cosi detto bit-flip, cioè un problema di transizione di stato logico non controllato su uno o più bit, una situazione di questo tipo viene gestita solitamente attaverso tecniche di error-correcting code ECC, tuttavia un eccessivo bit-flip può comunque comportare perdita di dati. UBI gestisce i casi di bit-flip spostando i dati dagli eraseblocks fisici che hanno bit-flip ad altri eraseblocks fisici privi di tale difetto. Questo processo è chiamato Scrubbing. Lo Scrubbing viene effettuato in modo trasparente in background ed è un processo nascosto per i layers superiori.

Algoritmo 17: Comandi per creare un'immagine binaria UBI

```
mkfs.ubifs -r ./storage/ -m 2048 -e 258048 -c 4096 -o img
    -storage.ubifs
mkfs.ubifs -r ./rootfs/ -m 2048 -e 258048 -c 4096 -o img-
    rootfs.ubifs
ubinize -o img.ubi -m 2048 -p 256KiB ubinize.cfg
```

Nell'algoritmo 17 vengono presentati i comandi per la creazione dell' immagine binaria da utilizzare per programmare una NAND Flash. I tool utilizzati sono due:

- mkfs.ubifs
  - Crea i volumi storage e rootfs
- ubinize
  - Unisce i volumi creati in un'unica immagine binaria di programmazione

I parametri che si notano nell'algoritmo 17 sono direttamente dipendenti dall'architettura della NAND Flash utilizzata.

Il tool mkfs.ubifs utilizza i seguenti parametri:

- -m, –min-io-size= dimensione page
- -e, –leb-size= dimensione eraseblock
- -c, –max-leb-cnt= numero massimo di eraseblocks

- -o, –output= nome del file di output

Il tool ubinize prende i seguenti parametri da riga di comando più un file di configurazione:

- -o, –output= nome del file di output
- -p, –peb-size= dimensione eraseblock
- -m, –min-io-size= dimensione page

5.3.2 UBIFS filesystem

UBIFS è un filesystem sviluppato da Nokia in collaborazione con l'Università di Szeged. In un certo senso lo si può considerare come l'evoluzione del JFFS2. E' da precisare tuttavia che il layer UBIFS lavora sopra 2 sottosistemi, in ordine MTD e UBI, mentre JFFS2 lavora al di sopra del MTD subsystem.

Dunque UBIFS introduce un concetto abbastanza diverso rispetto al JFFS2, di fatto il compito di gestione dei bad blocks e in generale delle problematiche di usura di un chip Flash è demandato completamente al layer UBI, mentre il compito di filesystem viene affidato all'UBIFS. Nella figura 31 viene mostrato uno schema nel quale è possibile individuare i vari layer di gestione presenti in una memoria Flash di un sistema Embedded. Si nota dal basso verso l'alto, lo strato MTD che fa da supporto di traduzione hardware, il layer UBI che gestisce le problematiche fisiche delle Flash memory ed infine il filesystem di tipo UBIFS.

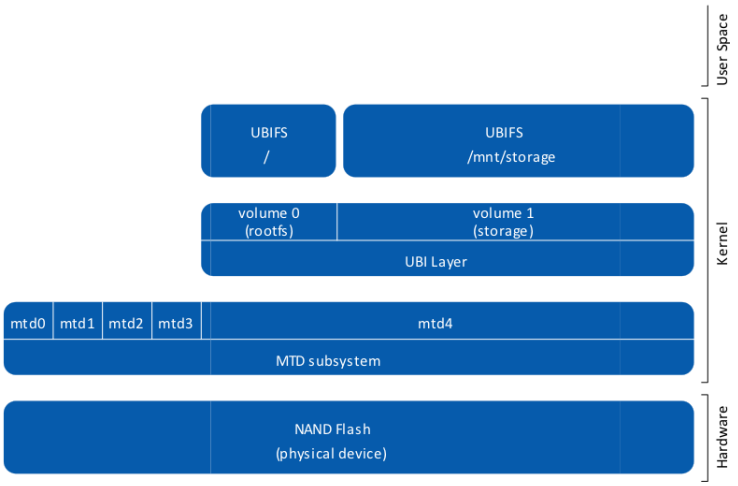


Figura 5.4: Organizzazione di una NAND Flash in un sistema Embedded Linux

Le principali caratteristiche del UBIFS sono:

- Scalabilità - UBIFS scala molto bene rispetto alle dimensioni della Flash; nel senso che, il tempo di montaggio, il consumo della memoria e la velocità di I/O non dipendono dal dimensione della Flash.
- Velocità nel mount - A differenza di JFFS2, UBIFS non deve eseguire la scansione di tutto il supporto per eseguire il montaggio. Per questo motivo il filesystem UBIFS ha bisogno solo di pochi millisecondi per essere montato, inoltre l'operazione non dipende dalle dimensioni della Flash; tuttavia, se consideriamo anche il tempo di inizializzazione UBI ci si accorge che questo dipende dalle dimensioni Flash duque tutto ciò deve essere considerato ai fini di una valutazione corretta.
- Supporto write-back - UBIFS a differenza del JFFS2 supporta il write-back, questo significa che i cambiamenti dei file non vengono immediatamente scritti sul supporto Flash, ma viene fatto uso di una cache. I cambiamenti dunque sul filesystem verranno effettivamente scritti solo quando sarà necessario farlo e solo attraverso la cache. Con questa tecnica si aumenta il throughput effettivo del filesystem in molte situazioni di carico del sistema.
- Tolleranza ai riavvii improvvisi - UBIFS è un filesystem journaling dunque preserva l'integrità dei dati da eventuali cadute di tensione. Il metodo adottato dai filesystem journaling è quello di creare un vero e proprio log delle operazioni, in tale modo riescono a mantenere coerente il filesystem in caso di spegnimenti improvvisi. Le eventuali operazioni sul log implicano un certo aumento del tempo di mount tuttavia UBIFS riesce ad eseguirle abbastanza rapidamente, poichè non ha bisogno di controllare l'intero supporto.

- Velocita dell'I/O - Come abbiamo detto utilizzando il supporto write-back UBIFS risulta estremamente veloce, tuttavia anche considerando di disabilitare il write-back (ovvero utilizzare un modo sincrono di scrittura) UBIFS rimane estremamente veloce. In caso di operazioni sincrone JFFS2 si dimostra più rapido rispetto a UBIFS ma questo è pienamente giustificato dal fatto che JFFS2 non ha un overhead derivante dalla gestione di un indice delle strutture dati all'interno del supporto Flash. UBIFS mantiene in ogni caso ottime prestazioni grazie al modo con cui gestisce il suo log tale da garantire operazioni di spostamento dati senza avere effettivamente cambiamento di eraseblock a livello fisico.

- Compressione al volo dei dati - UBIFS è in grado di memorizzare i dati in forma compressa, questo garantisce un aumento dei dati effettivamente memorizzabili sul supporto Flash. La tecnica di compressione adottata da UBIFS in realtà è molto simile a quella di JFFS2 ma risulta molto più flessibile. Infatti la compressione può essere attivata o disattivata in base all'inode, questo permette ad esempio di attivarla solo nel caso di dati che possono effettivamente avere buone percentuali di compressione.

- recuperabilità - UBIFS può in linea teorica essere pienamente recuperato se dovessero essere corromperse le informazioni di indicizzazione, questo poichè ogni blocco di informazione in UBIFS ha un'intestazione che contiene le informazioni di indicizzazione del filesystem. La recuperabilità tuttavia è ancora non implementata completamente, o meglio non esiste ancora uno strumento utente che possa recuperare le informazioni di indicizzazione del filesystem per il ripristino.

- Integrità - UBIFS (così come UBI) controlla attravero checksum tutto quello che scrive sul supporto Flash per garantire l'integrità dei dati. JFFS2 attua di fatto la medesima strategia in questo caso.

Per quanto detto circa le caratteristiche dei layer UBI e UBIFS, appare ora abbastanza chiara la scelta fatta per il sistema Embedded sviluppato in azienda di utilizzare un tipo di organizzazione della NAND Flash basata su tali sottosistemi.

Capitolo 6

Conclusioni

6.1 Considerazioni finali

Nella tesi presentata ho esaminato molti degli argomenti riguardanti l'ambito Linux e i sistemi Embedded. Nei vari capitoli ho cercato di utilizzare sempre un grado di approfondimento il più omogeneo possibile. L'argomento trattato tuttavia non può essere esaurito completamente. In generale ho preferito un'esposizione degli argomenti a basso livello trascurando la parte applicativa di alto livello. Questa scelta è dovuta più alla volontà di rendere chiari alcuni concetti, molto spesso utilizzati in pratica ma sostanzialmente poco approfonditi, che ad una mia preferenza personale. In conclusione mi sento di esporre il mio pensiero sul lavoro svolto e sulle tecnologie trattate. Le considerazioni tecniche fatte permettono di affermare che un sistema Linux Embedded ha notevolissime potenzialità di impiego, in quasi tutti i settori industriali in particolare quelli che richiedono connettività. I limiti che i sistemi di questo tipo hanno, sono ampiamente compensati dalla flessibilità da questi offerta. I pregi di un sistema Embedded Linux a livello tecnico sono stati già ampiamente trattati, tuttavia è importante fare una riflessione che ha un triplice ambito di osservazione. Un sistema Linux Embedded utilizza software libero ovvero opensource, in un periodo come il nostro in cui si parla molto spesso di condivisione e distribuzione delle risorse, secondo me diviene essenziale come primo obbiettivo la condivisione della cultura come risorsa principale dell'uomo. Sotto questo aspetto un progetto opensource è praticamente, economicamente ed eticamente valido. Appare dunque molto sensato l'investimento su progetti opensource, a patto di sviluppare una conoscenza approfondita di ciò che si sta utilizzando.

6.2 Ringraziamenti

Ringrazio in modo sentito mamma, papà, Sergio e Alessandro i quali hanno sempre contanto sulle mie capacità, aiutandomi nei momenti più difficili.

Ringrazio il mio relatore Professor Tomaso Erseghe che ha avuto molta pazienza nel seguirmi durante il tirocinio e mi ha sempre incoraggiato, lo stimo oltre che sotto l'aspetto professionale soprattutto dal punto di vista umano.

Ringrazio particolarmente il mio tutor aziendale Ing. Andrea Girardi che ha saputo impartirmi delle lezioni di vita e professionali molto importanti.

Ringrazio Matteo e Roberto che mi hanno accompagnato nel mio persorso di studio. Mi hanno arricchito come persona.

Ringrazio infine i miei amici di lunga data Gabriele, Mattia, Rosario e Dino con i quali ho sempre condiviso tutto malgrado la distanza che ci separa.

Bibliografia

[1] Craig Hollabaugh Ph.D. *Embedded Linux: Hardware, Software, and Interfacing*, 2002

[2] Daniel P. Bovet, Marco Cesati *Understanding the Linux Kernel, Third Edition*, 2005

[3] Tanenbaum Andrew S. *I moderni sistemi operativi*, 2002

[4] Christopher Hallinan *Embedded Linux Primer*, 2006

[5] Pearson Education *ARM system-on-chip architecture*,2000

[6] Robert Love *Linux System Programming* , 2007

[7] Peter Jay Salzman Michael Burian Ori Pomerantz *The Linux Kernel Module Programming Guide* , 2007

Manuali e datasheet consultati

[8] *ARM Architecture Reference Manual* , 2000

[9] *SAM9260 Datasheet* , 2008

[10] *Getting Started with the AT91SAM9260 Microcontroller Application Notes*, 2007

[11] *AT91 Assembler Code Startup Sequence for C Code Applications Software Application Notes*, 2002

[12] *AT91Bootstrap Application Notes*, 2006

[13] *NAND Flash Support on AT91SAM9 Microcontrollers Application Notes*, 2009

Siti consultati

[14] *http://www.at91.com/*

[15] *https://openwrt.org/*

[16] *http://www.linux-mtd.infradead.org/*