
UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Laurea Triennale in Informatica

# Sviluppo di un driver Linux per gestione energetica in un gateway edge

Tutor interno:

**Prof. Paolo Milazzo**

Candidato:

**Francesca Pinna**

Tutor esterno:

**Dott. Davide Salaris**

ANNO ACCADEMICO 2023/2024

# Indice

# Capitolo 1

# Introduzione

Questa relazione dettaglia il processo di sviluppo di un driver Linux, il cui scopo è di interfacciare il Dynagate, un gateway Edge per l'Internet of Things, al suo circuito integrato LTC3350. Questo è stato l'obbiettivo centrale del progetto di tirocinio svoltosi presso l'azienda Abinsula srl. nel periodo dall'8 luglio al 20 settembre 2024.

L'Internet of Things (IoT) si riferisce ad una rete di dispositivi connessi che comunicano e scambiano dati tra di loro attraverso internet. Questi vanno dai sensori ambientali e termostati intelligenti, a dispositivi indossabili come gli smart watches, fino ai sistemi integrati nelle automobili moderne, che permettono di monitorare le condizioni del veicolo, migliorare l'efficienza del carburante, e fornire servizi di navigazione in tempo reale. Un gateway IoT edge è un dispositivo che funge da intermediario tra i dispositivi IoT e le reti esterne, come il cloud ed altre infrastrutture di rete.

Il Dynagate è un gateway edge progettato per fornire connettività a dispositivi nel settore automotive e in applicazioni industriali. Offre un'elevata capacità di elaborazione locale, supporta diversi protocolli di comunicazione e garantisce una robusta resistenza agli attacchi informatici. La sua capacità di analisi dei dati riduce il carico sulla comunicazione e migliora l'efficienza complessiva del sistema edge.

LTC3350 è un circuito integrato nel Dynagate, responsabile della gestione dell'alimentazione di backup attraverso supercondensatori ad alta corrente. Il suo chip si trova all'interno della board del Dynagate, ed è collegato all'unità di elaborazione centrale tramite bus $I^2C$. In situazioni di interruzione di corrente, LTC3350 assicura che il Dynagate continui a funzionare.

Durante l'operazione normale, LTC3350 monitora costantemente voltaggio e corrente di diverse componenti del sistema di alimentazione. Ha anche capacità di controllare la salute dei supercondensatori, assicurando che vengano sostituiti per tempo in caso di degrado della batteria. Questo contribuisce a rendere il sistema più robusto e resiliente nelle sue applicazioni.

LTC3350 svolge parte delle sue operazioni in automatico, come attivare l'alimentazione di backup alla necessità, e regolare il caricamento dei supercondensatori per un utilizzo efficiente. Per sfruttare le sue funzionalità di monitoraggio, e personalizzare i suoi parametri di funzionamento è necessario interfacciarlo con il sistema operativo: qui entra in gioco il driver che ho sviluppato.

Il driver, che chiamerò `ltc3350-driver` in casi di ambiguità, permette all'utente del Dynagate di comunicare con LTC3350, leggendo i valori da esso misurati, impostando allarmi per monitorarne le condizioni e programmando parametri di gestione alimentazione. In parallelo alla creazione del driver, ho implementato un'applicazione al livello utente che ne utilizza e dimostra le features, e che ha permesso il testing del driver e del gateway.

Il Dynagate è prodotto da Eurotech, un'azienda globale specializzata in PC Embedded e piattaforme software per l'IoT. Abinsula ha supportato la creazione di una distribuzione Linux specializzata per i suoi gateway, chiamata Eurotech Everyware Linux.

Abinsula è uno dei principali attori italiani nel campo delle soluzioni embedded, IoT, web e mobile. Fondata nel 2012, si è affermata al livello mondiale nel campo automotive(applicazioni embedded nel campo automobilistico) grazie al suo alto livello di specializzazione. Collabora da anni con aziende estere, a cui offre supporto per il rilascio di nuovi prodotti, ma lavora anche alla manutenzione e al miglioramento di sistemi preesistenti.

In questo contesto, Eurotech ha richiesto ad Abinsula lo sviluppo del driver per LTC3350, in vista di un rilascio successivo del sistema operativo del Dynagate. Mi sono inserita nel team di sviluppo di Eurotech Everyware Linux, in cui ho potuto studiare, progettare e sviluppare il lavoro di cui parlerò in questa relazione.

## 1.1 Contenuto di questa relazione

Nel seguente documento esporrò nel dettaglio il background necessario alla comprensione del progetto, e le diverse fasi di sviluppo del prodotto finale.

- Il prossimo capitolo riassume le informazioni più importanti sullo sviluppo di un driver del kernel di Linux: le caratteristiche fondamentali del sistema Linux, la sua gestione dei dispositivi attraverso drivers, e altri dettagli del suo funzionamento che sono stati necessari allo sviluppo del driver, quale l'utilizzo e la gestione delle interruzioni.

- Il terzo capitolo descrive il protocollo $I^2C$ e il circuito LTC3350, spiegando come funziona e in che modo comunica con l'unità di elaborazione centrale.

- Il quarto capitolo contiene una breve analisi dei software preesistenti per LTC3350, delle loro criticità, e dei driver per bus $I^2C$ esemplari per il funzionamento di un device periferico.

- Il quinto capitolo riassume le tecnologie utilizzate per l'implementazione e il funzionamento del driver, tra cui lo Yocto Build System e le librerie del kernel.
- Il sesto capitolo riassume il processo di individuazione dei requisiti del prodotto, ed esamina alcune scelte implementative.
- Il settimo capitolo descrive i metodi con cui ho implementato le funzionalità del driver, con esempi di codice significativi.
- L'ottavo capitolo descrive i test svolti per valutare il funzionamento del driver, e poi alcuni esperimenti che ne sfruttano a pieno le funzionalità.
- Il nono capitolo conclude questa relazione con considerazioni sui risultati e le prospettive future del progetto.

# Capitolo 2

# Contesto

## 2.1 Linux

«Il kernel di Linux è il sistema operativo più flessibile esistente», sostiene Greg Kroah-Hartman, autore di *Linux Kernel in a Nutshell* e uno dei più importanti sviluppatori di Linux, in quanto manutentore della branch *stable*. Linux può essere personalizzato per un'ampia gamma di sistemi diversi, dai più piccoli sistemi embedded alla maggior parte dei supercomputer più grandi al mondo.[1]

Secondo Alberto Liberal de los Rios, «il kernel di Linux è uno dei progetti open source più grandi e di maggior successo mai realizzati.» La sua evoluzione costante è alimentata da una comunità attiva di migliaia di sviluppatori in tutto il mondo.

Sostiene che il vantaggio principale di Linux è la sua abilità di riutilizzare i componenti; è reso scalabile dalla sua modularità e configurabilità.[2]

Linux è utilizzabile su piattaforme hardware estremamente limitate. Il suo kernel può essere molto piccolo e compatto: è possibile far stare un'immagine del kernel, inclusi alcuni programmi di sistema, in un floppy disk da 1.44 MB. Il suo codice open source permette a migliaia di aziende e individui di contribuire al suo miglioramento, e di utilizzarlo gratuitamente.

L'utilizzo di Linux e di varie componenti open source nei sistemi embedded si chiama "Embedded Linux". Si può trovare nell'elettronica di consumo, come i dispositivi smart, nell'IVI (sistemi di intrattenimento e informazione su veicoli), nelle apparecchiature di rete, nella gestione di macchinari industriali, nell'automazione industriale, nella navigazione, nel software di volo per veicoli spaziali e in molti strumenti medici.[2]

## 2.2 Il kernel di Linux

Il kernel è il nucleo che determina il funzionamento dell'intero sistema Linux. È il livello di software inferiore eseguito sul sistema. Gestisce l'hardware, esegue i pro-

grammi utente, mantiene la sicurezza e integrità dell'intero sistema, pur essendone una parte relativamente piccola.

Il kernel di Linux è monolitico. È un singolo programma, per quanto ampio e complesso, che svolge la maggior parte delle funzioni del sistema operativo, in contrasto ai sistemi operativi che prevedono un *microkernel*, il quale svolge un set di funzioni relativamente piccolo.[3]

### 2.2.1 Kernel Mode e User Mode

I sistemi operativi basati su Unix, come Windows e Linux, nascondono i dettagli a basso livello del computer alle applicazioni eseguite dall'utente.

Quando un programma vuole utilizzare una risorsa hardware, deve effettuare una richiesta al sistema operativo. Il kernel valuta la richiesta, e, se decide di accettare, interagisce con i componenti hardware al posto dell'applicazione utente. Questo garantisce una maggiore tolleranza agli errori del sistema, impedendo di accedere direttamente ai dispositivi hardware e limitando gli accessi in memoria.

Per ottenere questo risultato, l'esecuzione della CPU è divisa al livello hardware in due modelli di esecuzione: una modalità non privilegiata per i programmi utente, chiamata User Mode, e una privilegiata per il kernel, detta Kernel Mode.

### 2.2.2 Drivers e Moduli

Linux può accomodare un'ampia varietà di dispositivi I/O fisici, prodotti da un grande numero di aziende. C'è diversità nelle interfacce hardware per i dispositivi così come nei chip per gestirli. Circa 70% del kernel di Linux è software device-specific, secondo un sondaggio riportato in *Operating Systems, Principles & Practice.*[3]

La portabilità di Linux è data dai driver device *dynamically loadable*, ovvero caricabili a runtime. Si tratta di software per gestire un dispositivo, un'interfaccia o un chipset, aggiunto al sistema operativo dopo il suo avvio, per gestire i dispositivi presenti in una specifica macchina.

Tipicamente, lo sviluppatore del dispositivo produce il codice del driver, ma non sempre, e non necessariamente per tutti i sistemi operativi. Il sistema operativo invoca il driver quando ha bisogno di leggere o scrivere dati sul device.

Il sistema operativo si avvia con un numero limitato di drivers. Per i dispositivi attaccati fisicamente al computer, come nel caso di LTC3350, è il produttore del computer ad indicare quali driver devono essere caricati tramite il Device Tree. Al suo avvio, il kernel verifica quali dispositivi sono collegati al computer e carica i driver appropriati.

**Moduli** I device drivers sono un esempio di modulo: una feature del kernel che realizza i vantaggi di modularità e portabilità delle componenti di un sistema operativo. Un modulo è un file oggetto sul cui codice si può effettuare il *linking* (ed eventuale *unlinking*) a runtime.

Oltre ad un device driver, un modulo può implementare un filesystem o altre funzionalità dello strato superiore del kernel. Il modulo non è eseguito come un processo specifico, ma è eseguito in Kernel Mode al posto del processo corrente, come qualsiasi altra funzione del kernel.

L'utilizzo di moduli ha vantaggi sia dal punto di vista di portabilità, che dall'efficienza in tempo e spazio. Infatti, anche se un modulo può basarsi su alcune caratteristiche hardware specifiche, è indipendente dalla piattaforma - un driver per LTC3350 funzionerà ugualmente se il circuito si trova su un PC, su un gateway, o su un supercomputer. Inoltre, una volta svolto il linking, il codice oggetto di un modulo è equivalente al codice sorgente del kernel statically linked, pertanto non è necessario alcun passaggio di messaggi esplicito quando sono invocate le funzioni del modulo. L'unlinking di un modulo permette anche di liberare spazio in memoria, in modo trasparente all'utente.

## 2.3 Struttura di un sistema Linux

Le componenti principali di un sistema Linux embedded sono il Bootloader, il Kernel, l'interfaccia delle chiamate di sistema, la libreria a runtime di C, le librerie di sistema, e il Root filesystem.

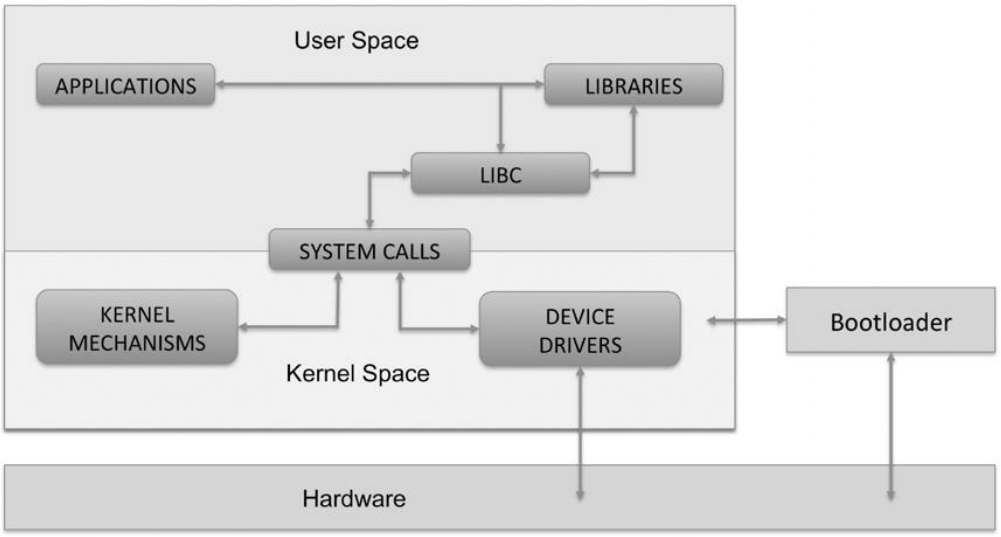



Figura 2.1: Visione ad alto livello di un sistema Linux embedded

### 2.3.1 Il bootloader

Linux necessita di una piccola quantità di codice machine-specific per inizializzare il sistema. Il bootloader ha il ruolo di configurare il sistema di memoria, caricare agli indirizzi corretti il kernel, il device tree, e opzionalmente il disco di RAM, e di configurare la command-line del kernel. Di solito, il bootloader configura anche una console seriale per il kernel.

**U-Boot** Il Dynagate ha un'architettura ARM64, e il bootloader standard per le architetture ARM si chiama U-Boot. Queste sono le sue caratteristiche principali:

- **Piccola taglia**: una configurazione di U-Boot dovrebbe essere più piccola di 128 KB.
- **Veloce**: limita al massimo il numero di devices inizializzati.
- **Portabile**: utilizzabile in una vasta gamma di boards.
- **Configurabile**: permette di scegliere le features necessarie alla board.
- **Debuggabile**: permette di identificare e correggere errori facilmente.

Nel caso di questo progetto, la configurazione del bootloader è stata effettuata dai programmatori di Abinsula prima dell'inizio del tirocinio.

### 2.3.2 Altri elementi

**System Calls e C Runtime Library** La chiamata di sistema è l'interfaccia principale tra un'applicazione e il kernel, ed è l'unico modo in cui un'applicazione in userspace può interagire con il kernel. La netta separazione tra kernelspace e userspace impedisce alle applicazioni di accedere liberamente alle risorse interne del kernel, assicurando la sicurezza e la stabilità del sistema.

Le chiamate di sistema tendono a non essere invocate direttamente, ma attraverso funzioni "wrapper" della libreria Runtime di C.

**Librerie condivise** Le librerie condivise sono caricate dai programmi all'inizio della loro esecuzione, e sono poi utilizzate automaticamente dai programmi eseguiti successivamente. Un esempio di queste librerie è *libc*, che contiene il supporto di base per il linguaggio C e servizi per l'accesso alle chiamate di sistema.

**Root filesystem** Tutti i files contenuti nella gerarchia dei file si trovano nel root filesystem, il cui punto di accesso è '/'. Contiene tutte le librerie, le applicazioni e i dati.

## 2.4 Linux Device and Driver Model

I sistemi operativi liberi permettono a tutti di vedere il loro funzionamento interno. Chiunque ne sia capace può esaminare, comprendere e modificare il sistema operativo. Tuttavia, il kernel di Linux resta un insieme di codice grande e complesso, contando ora più di trenta milioni di righe[4], e i drivers possono essere un punto di ingresso per approcciare il codice senza essere sopraffatti dalla sua complessità.[5]

### 2.4.1 Il ruolo del driver

Il ruolo del device driver è quello di un intermediario tra l'hardware e lo userspace: agisce come una scatola nera, che fornisce un'interfaccia di programmazione ben definita allo spazio utente, nascondendo il funzionamento dell'hardware.

Le applicazioni del livello utente agiscono tramite chiamate di sistema standard, che sono indipendenti dal driver. Il suo scopo è di mappare queste chiamate di sistema alle operazioni specifiche che agiscono sul dispositivo.

La distinzione tra il meccanismo e la politica è un'idea fondamentale nel design di Unix. La maggior parte dei problemi di programmazione possono essere divisi in due parti:

1. **il meccanismo**: quali funzionalità devono essere offerte.
2. **la politica**: in che modo queste funzionalità devono essere usate.

Il programmatore che lavora su un driver deve effettuare delle scelte che garantiscano la sua flessibilità, offrendo un meccanismo invece di imporre una politica. Se le due questioni sono risolte da parti diverse del programma, o addirittura in programmi diversi, il pacchetto software è più semplice da sviluppare ed adattare a bisogni specifici. All'interno del progetto di cui stiamo trattando, questa idea è rispecchiata nella divisione tra il driver in sé, e l'applicazione che lo utilizza.

Il programmatore di driver deve fare particolare attenzione a questo concetto: il codice kernel deve permettere di accedere all'hardware, ma non forzare alcun tipo di politica sull'utente. Lo scopo è di rendere l'hardware disponibile, e non di prescrivere in che modo l'utente lo deve utilizzare.

È comune affrontare la questione di politica rilasciando il driver insieme con programmi utente per aiutare la configurazione e l'accesso al device target. Oltre all'inclusione di una applicazione, la configurazione del driver avviene attraverso il device tree: una struttura dati utilizzata nel kernel Linux per descrivere l'hardware.

### 2.4.2 Concetti del Device and Driver Model

Il Device Model unificato è stato introdotto nella versione 2.6 del kernel come un meccanismo universale per rappresentare i dispositivi, e descrivere la loro topologia nel sistema, organizzando dispositivi e drivers in **bus**.[2]

In informatica, si chiama bus un circuito per la trasmissione di dati, costituito da un insieme di linee dati, trattate come un singolo segnale logico.[6] Ad esempio, il **system bus** connette il processore alla memoria. In questo contesto, il bus è un canale tra il processore e uno o più dispositivi.

Un esempio di bus è una connessione $I^2C$: composta da una linea di clock e una linea dati, permette una comunicazione tra il processore, tramite il suo circuito controller, e un dispositivo client(vedi 3.2).

Nel Linux Device and Driver Model, si definiscono:

- **device**, o dispositivo, un oggetto fisico o virtuale attaccato ad un bus.
- **bus**, anch'esso un dispositivo, che collega altri dispositivi.
- **driver**, un'entità software che può effettuare il *probing* ed essere associato ad un dispositivo. Effettua alcune operazioni di gestione del dispositivo.

Il probing è il processo attraverso il quale il driver identifica e stabilisce una connessione con uno specifico dispositivo, hardware o virtuale, del sistema. Si può intuire che siccome un bus è un dispositivo, esistono driver per gestire i bus.

### 2.4.3 Tipi di device drivers

Nel contesto del Device and Driver Model, possiamo considerare tre tipi di drivers: bus core drivers, bus controller drivers, e device drivers.

Linux prevede una distinzione tra tre tipi fondamentali di dispositivo: i character devices, cioè dispositivi a cui si può accedere come un flusso di bytes, block devices, che possono contenere un filesystem, e network interfaces, dispositivi che abilitano lo scambio di dati con altri hosts.

**Bus Core Driver** Esiste un bus core driver generico per ogni tipologia di bus supportato dal kernel(USB, PCI, $I^2C$, ecc.). Il bus core driver definisce una struttura dati che rappresenta una tipologia di bus. In questo progetto, ho usato il bus core driver I2C.

**Bus Controller Driver** Per ogni tipo di bus possono esistere molti dispositivi controllers diversi da produttori diversi, e ognuno ha il suo driver. Esistono decine di driver per diversi produttori di bus $I^2C$. Il bus controller driver di questo progetto è `I2C-IMX`.

**Device drivers** Il device driver ha il dovere di inizializzare le struttura dati `device_driver`, può leggere i dati di configurazione dal device tree, e implementare funzionalità aggiuntive. Il driver che ho implementato è un device driver per LTC3350.

**Esempio** Ad ogni device driver è associata una struttura dati che contiene le informazioni necessarie al suo caricamento.

```
static const struct i2c_device_id smbalert_ids[] = {
  { "smbus_alert", 0 },
  { /* LIST END */ }
};
MODULE_DEVICE_TABLE(i2c, smbalert_ids);

static struct i2c_driver smbalert_driver = {
  .driver = {
    .name = "smbus_alert",
  },
  .probe    = smbalert_probe,
  .remove   = smbalert_remove,
  .id_table = smbalert_ids,
};
```

Listato 2.1: creazione device driver

In questo esempio, mostro la struttura dati che rappresenta smbalert-driver, che è descritto nel dettaglio nella sezione 4.3.1. La tabella degli id, o id table, permette di verificare la compatibilità tra device e drivers. La macro `MODULE_DEVICE_TABLE` registra la tabella in strutture dati generali del kernel, permettendo di registrare il driver. Per ogni struttura dati di un driver, è necessario indicare il suo nome, e funzioni per gestirlo.

La funzione `probe()` è la funzione di probing legata ad un driver, che identifica e inizializza il dispositivo ad esso associato. La funzione `remove()` ha lo scopo opposto: di eliminare le strutture dati di un dispositivo e "scollegarlo" dal driver. Esistono altre funzioni di questo tipo, tra cui la funzione `alert()` di cui si parla in 4.3.2 e nel capitolo 7.

### 2.4.4 Device Tree

L'Open Firmware Device Tree, o semplicemente Device Tree (albero dei dispositivi), è una struttura dati e un linguaggio per descrivere l'hardware. Specificamente, è una descrizione dell'hardware leggibile dal sistema operativo.

Il DT è un albero formato da nodi con nome, ciascuno dei quali può avere un numero arbitrario di proprietà, che rappresentano dati arbitrari.

Esistono convenzioni d'uso, chiamate **bindings**, che permettono di definire i bus, le interrupt lines, le connessioni GPIO, e i device periferici. Il DT è rappresentato come un insieme di file di testo nel Linux kernel source tree.

Gli scopi principali del Device tree sono i seguenti:

1. **Identificare la macchina**: può essere necessario per il kernel riconoscere la board durante la fase di accensione, in modo da eseguire operazioni machine-specific.
2. **Configurazione a runtime**: il DT è il modo principale, se non unico, di passare dati di configurazione dal firmware al kernel.
3. **Popolazione della lista dei device**: Una volta identificata la board, e svolta una prima configurazione, il kernel può procedere con il setup delle interruzioni e la popolazione del Linux device model.

## 2.5 Interruzioni

Un'interruzione è definita come un evento che altera la sequenza di istruzioni eseguite dal processore. Questi eventi corrispondono a segnali elettrici generati da circuiti hardware sia dentro che fuori il chip della CPU.[1]

Le interruzioni sono spesso suddivise in sincrone e asincrone. Le prime sono anche chiamate eccezioni, e sono generate dalla CPU solo dopo aver terminato l'esecuzione di un'istruzione. Le seconde sono generate da altri device hardware, come device I/O, in momenti arbitrari rispetto ai segnali di clock della CPU. Le eccezioni sono causate da errori di programmazione o condizioni anomale che devono essere gestite dal kernel, e le interruzioni hardware sono causate da eventi esterni, quali premere un pulsante.

**Segnali di interruzione** Quando il processore riceve un segnale di interruzione, ferma l'esecuzione del processo corrente ed inizia una nuova attività. Il codice eseguito in seguito non è un processo separato, ma è una sequenza di istruzioni che prende il posto del processo corrente, esterno al flusso di esecuzione normale. Questo codice si chiama *kernel control path*. Per effettuare questo *switch*, la CPU salva il valore corrente del program counter nello stack della Kernel Mode. Al suo posto, scrive l'indirizzo di un pezzo di codice dedicato alla gestione dell'interruzione.

Un'interruzione può essere maskable o nonmaskable. Nel primo caso può essere disattivata tramite scritture su registri. Tutte le interruzioni generate da dispositivi I/O sono maskable. Invece, le interruzioni nonmaskable non possono essere disattivate, e sono utilizzate solo per eventi critici, come fallimenti nell'hardware.

**Interrupt request** Una Interrupt ReQuest (IRQ), o richiesta di interruzione, è un segnale inviato da un dispositivo hardware al processore, per richiedere un'attenzione immediata. Ad una interrupt request corrisponde un identificatore numerico, chiamato numero IRQ, che permette al sistema operativo di capire quale dispositivo ha generato l'interruzione. Questo è reso più complicato dal fatto che diversi dispositivi possono condividere la stessa linea IRQ.

**Interrupt controller** Ogni dispositivo hardware in grado di causare interruzioni ha una linea di output designata per interrupt requests, che sono collegate ad un circuito hardware chiamato Interrupt Controller. Questo è il punto di incontro tra le linee IRQ e la CPU, poiché verifica i segnali attivi sulle linee IRQ e li notifica alla CPU. Un esempio linea IRQ è la linea `#SMBALERT` di LTC3350(vedi 3.3.3).

**Interrupt vector** Ogni interruzione o eccezione è identificata da un numero da 0 a 255, talvolta chiamato *interrupt vector*, o vettore di interruzione. Una tabella di sistema chiamata *Interrupt Descriptor Table*(IDT) associa ad ogni vettore l'indirizzo di un *interrupt handler*, il codice che gestisce l'interruzione corrispondente.

### 2.5.1 Interrupt handler

Come detto sopra, il *kernel control path* è la sequenza di istruzioni eseguite in kernel mode per gestire un'eccezione o un'interruzione. La prima parte di questo codice salva il contenuto dei registri nello stack della Kernel Mode, e la sua ultima parte ripristina lo stato precedente dei registri, e riporta la CPU in User Mode.

I kernel control path possono avere un'esecuzione concorrente tramite *interleaving*, infatti un interrupt handler può essere interrotto da un altro interrupt handler. Questa scelta di implementazione permette una gestione delle linee di interrupt più efficiente, perché l'interrupt controller richiede un riconoscimento dalla CPU per riabilitare la linea IRQ.

Un gestore di eccezioni ha un funzionamento generalmente semplice, e spesso si limita a mandare un segnale Unix al processo che ha generato l'eccezione. Ad esempio, il tentativo di effettuare una divisione per zero causa una "floating-point exception", che corrisponde al segnale `SIGFPE`. Un accesso ad un indirizzo di memoria non valido causa l'eccezione "invalid memory reference", o "segmentation fault", che corrisponde al segnale `SIGSEGV`.[7]

Gli interrupt handlers, o gestori di interruzioni, hanno un comportamento più complesso. Questo è dovuto, in primo luogo, al fatto che diversi dispositivi possono condividere la stessa linea IRQ, per cui l'interrupt vector può non essere sufficiente a determinare le operazioni da svolgere. Di conseguenza, l'interrupt handler è associato a diverse *interrupt service routines* (ISR), ciascuna delle quali è una funzione dedicata ad uno specifico dispositivo che usa la linea IRQ.

Tutti gli interrupt handlers devono salvare il numero IRQ e i valori dei registri nello stack della Kernel Mode, notificare all'Interrupt Controller di star gestendo l'interruzione, eseguire le ISR associate ai dispositivi connessi alla linea IRQ, poi terminare, ripristinando lo stato precedente.

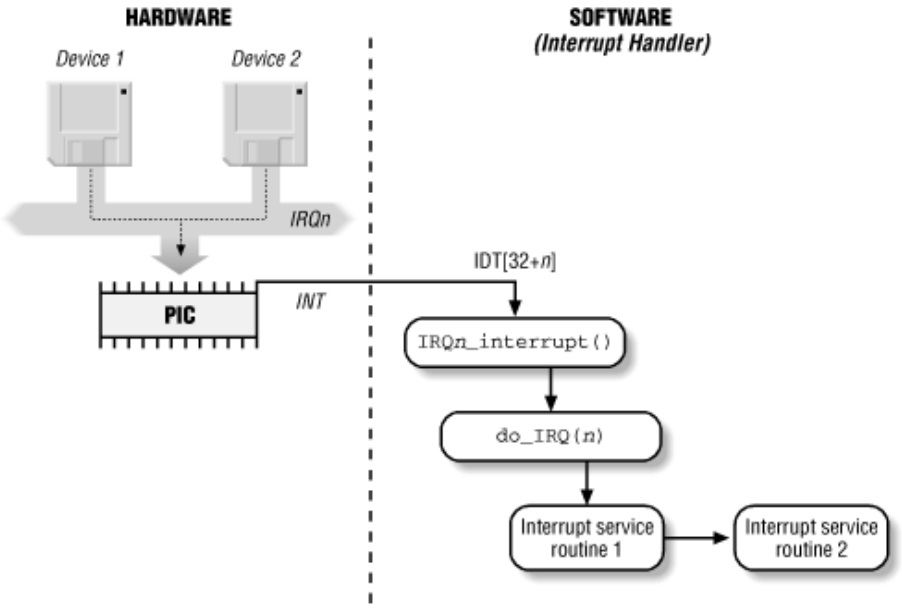



Figura 2.2: Gestione di interruzioni [1]

**Livelli di urgenza** Il gestore di interruzioni ha una esecuzione estremamente privilegiata, ma può dover eseguire operazioni con livelli di criticità diversi. Durante l'esecuzione del gestore, i segnali sulla linea IRQ corrispondente sono ignorati. Inoltre, non è possibile eseguire procedure bloccanti come operazioni I/O nel gestore, a rischio di bloccare l'intero sistema. Le azioni da svolgere dopo un'interruzione sono divise in tre livelli di urgenza:

- **Critiche**, come notificare l'interrupt controller di star gestendo l'interruzione. Queste azioni sono veloci, e sono eseguite immediatamente nell'interrupt handler, con le interruzioni maskable disattivate.
- **Non critiche**, come aggiornare strutture dati accessibili solo dal processore. Anche queste sono azioni veloci, e sono eseguite immediatamente, con le interruzioni attive.
- **Non critiche differibili**, cioè azioni che possono essere riinviate o delegate ad un altro processo. Queste azioni possono includere operazioni I/O, o l'interazione con un altro processo, come inviare il testo digitato sulla tastiera al processo che gestisce il terminale. Possono essere posticipate senza effetti sull'operazione del kernel, e sono effettuate da funzioni separate chiamate "bottom halves", cioè metà inferiori.

**ISR** L'interrupt handler chiama tutte le ISR associate alla sua linea di interrupt dopo aver salvato lo stato dei registri nello stack del kernel. Le ISR svolgono operazioni specifiche al dispositivo la cui interruzione stanno gestendo, il che include la pianificazione di operazioni non urgenti.

**Bottom halves** Le bottom halves sono funzioni kernel di bassa priorità, solitamente legate alla gestione di interruzioni, che sono eseguite quando il kernel finisce di eseguire operazioni più urgenti e sceglie il nuovo processo da eseguire.

### 2.5.2 Richiedere una linea IRQ

Per permettere ad un dispositivo di utilizzare una linea IRQ, il suo driver deve invocare la funzione `request_irq()`. Questa funzione aggiunge un nuovo descrittore nella lista degli IRQ. Vedremo questo processo più nel dettaglio quando analizzeremo il driver `smbalert-driver`, che ci permette di gestire le interruzioni causate da `SMBUSALERT#`, da parte del dispositivo `LTC3350`.

## 2.6 Dynagate

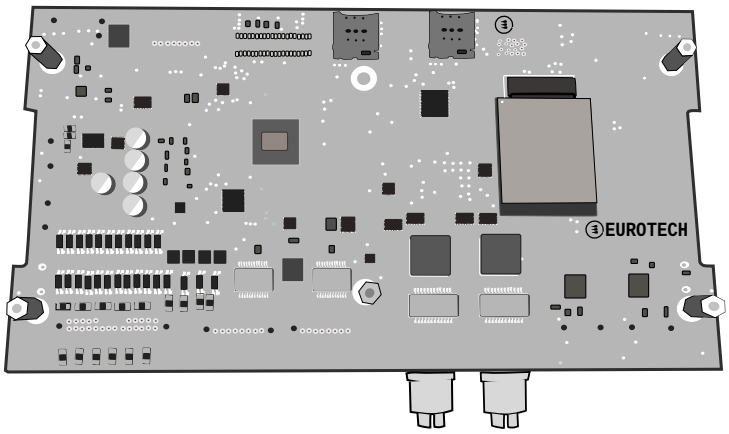



Figura 2.3: Disegno della parte di sopra del Dynagate

Il Dynagate è un calcolatore che permette di connettere un sistema embedded ad internet, nel campo automobilistico o industriale. È indicato in applicazioni moderatamente robuste: è in grado di funzionare anche in condizioni di temperatura e pressione difficili, ha una buona resistenza ai guasti e, in particolare, è in grado di funzionare anche in caso di interruzioni momentanee di corrente.

È un dispositivo progettato per l'edge computing, essendo un punto di connessione tra i dispositivi IoT(Internet of Things), e le reti esterne, come il cloud ed altre infrastrutture di rete.

Il processore di questo Gateway è basato sulla famiglia di processori AM335x Cortex-A8, con 1GB di RAM e 4GB di memoria flash permanente. Offre connettività tramite un modem LTE, con supporto per Micro-SIM, Wi-Fi, Bluetooth e Ethernet. Possiede un GNSS interno per fornire localizzazione precisa. È progettato per la capacità di ricevere input da telecamere poste su un veicolo, e elaborare i dati tramite AI Computer Vision, ad esempio per riconoscere gli oggetti sulla strada.

Il sistema operativo del dispositivo è Eurotech Everyware Linux, che è una distribuzione di Linux personalizzata per gestire i gateway Eurotech.

# Capitolo 3

# LTC3350 e protocollo $I^2C$

LTC3350 è il circuito integrato che monitora e gestisce l'alimentazione di backup nel dynagate. Oltre a gestire il caricamento e l'utilizzo dei supercondensatori, permette di personalizzare il funzionento del circuito, e di monitorare diverse valori, tramite il suo collegamento all'unità di elaborazione centrale. LTC3350 è collegato al processore tramite bus $I^2C$. Questo capitolo contiene una descrizione del protocollo $I^2C$, nella sua variante SMBus, e di LTC3350 stesso.

## 3.1 Circuiti integrati e i loro collegamenti

Un circuito integrato, anche detto IC o chip, è un pezzo quadrato di silicone, tipicamente grande 5mm x 5mm, in cui sono state depositate delle porte logiche. La caratteristica fondamentale di un circuito integrato è di contenere molti transistor in un singolo chip di silicio.[8]

Un circuito integrato è solitamente inserito in un package, di plastica o ceramica, alle cui estremità sono inserite delle file di pin che possono essere attaccati o saldati ad una board. Il pin rappresenta un input o un output fisico che porta un segnale elettrico. Ogni pin è collegato all'input o l'output di una porta logica nel chip, o alla corrente, o a terra.[9] Anche se un pin può effettuare una sola funzione alla volta, può essere configurato internamente per svolgere funzioni diverse, attraverso il *pin multiplexing*.[2]

Le componenti di una scheda sono collegate tra loro tramite **linee**, dei percorsi che trasportano segnali elettrici o alimentazione.[11] Sono solitamente fili o tracce di rame su una scheda a circuito stampato, che permettono la comunicazione o il trasferimento di energia all'interno della scheda.

Le linee di segnale trasportano impulsi digitali o segnali analogici tra componenti. Le linee digitali trasportano dati binari, e rappresentano informazioni codificate. Due esempi significativi sono le linee di clock, che permettono la sincronizzazione tra i circuiti, e le linee dati, che trasportano informazione. Diciamo che una linea è

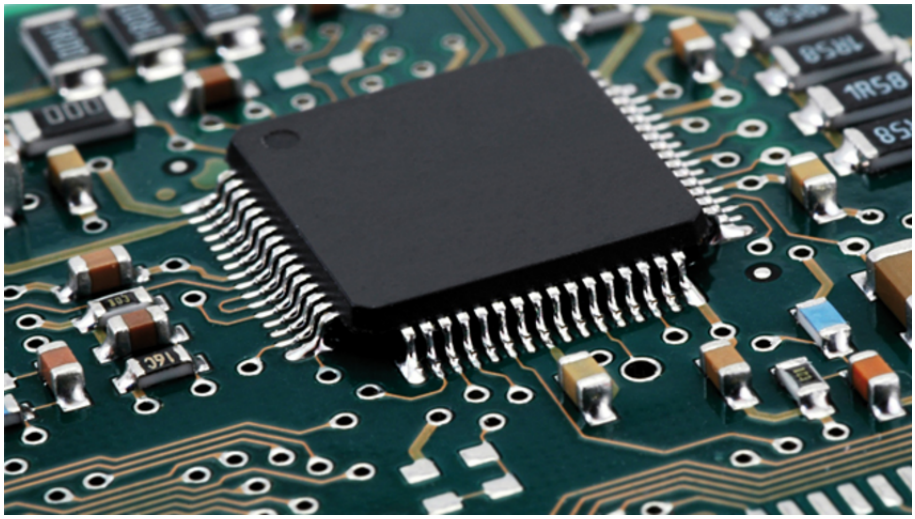


Figura 3.1: Un chip in silicio [10] (Fotolia/Sabine)

alta quando attraverso di essa passa una certa quantità di corrente, bassa quando la sua corrente è vicina a zero.

## 3.2 I²C e protocollo SMBus

I²C, pronunciato *i-quadro-c*, è un protocollo di comunicazione seriale che usa una linea dati (SDA) e una linea di clock (SLC).

Il protocollo prevede due tipi di device: i device controller, che hanno il ruolo di arbitrare la comunicazione, e i device target o client. I termini utilizzati storicamente per indicare i dispositivi controller e target sono rispettivamente master e slave, presenti ancora in alcune funzioni di libreria, ma questa relazione utilizzerà la nomenclatura moderna.

Il protocollo I²C supporta l'utilizzo, su un singolo bus, di uno o più device controller, e di uno o più device target, che possono mandare e ricevere comandi e dati. La comunicazione avviene attraverso **pacchetti** di byte con indirizzi unici per ciascun device target.

### 3.2.1 Storia

I²C sta per Inter-Integrated Circuit protocol, cioè protocollo di comunicazione tra circuiti integrati. E' stato sviluppato nel 1982 da NXP Semiconductor, per collegare dispositivi controller come microcontrollori e processori, a dispositivi target come data converters e altre periferiche.

Il suo successo è dovuto al fatto che necessita di sole due linee di segnale per un singolo bus, che possono essere usate per collegare numerosi device.[12]

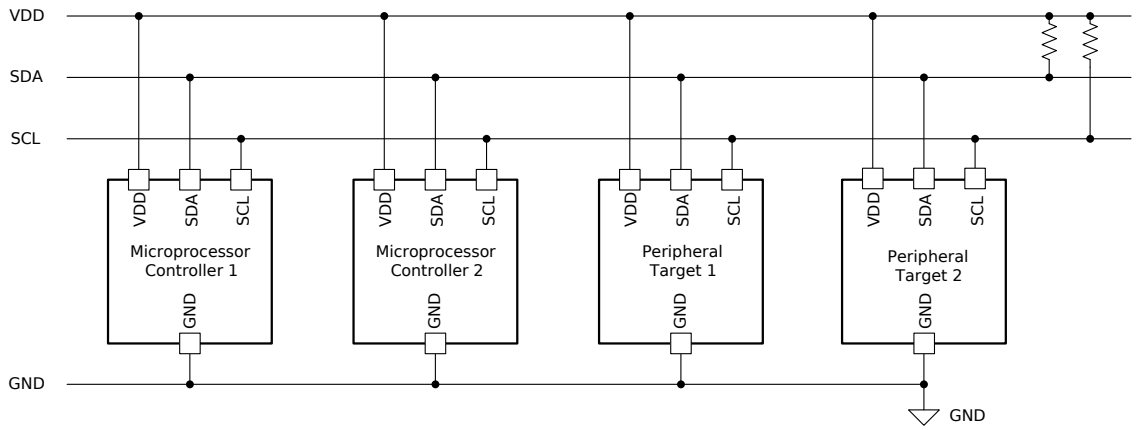



Figura 3.2: Implementazione fisica tipica di I²C [12]

### 3.2.2 Livello fisico

Nella figura 3.2 si possono vedere le due linee di segnale, SCL e SDA, la linea di alimentazione VDD, e il collegamento a terra(GND). Al bus sono collegati due dispositivi controller e due dispositivi target.

I²C è un protocollo di comunicazione half-duplex, per cui i device e i controller mandano dati uno alla volta. In un protocollo full-duplex, come SPI, i dati possono essere mandati e ricevuti allo stesso tempo.

Un controller I²C dà inizio e fine alla comunicazione, quindi non si presentano conflitti. Inoltre, ad ogni device target è associato un indirizzo unico nel bus, per poter inviare i messaggi ad un device specifico.

### 3.2.3 Il protocollo I²C

Sia la linea SDA che quella SCL sono bidirezionali. In I²C, i dati sono trasferiti in messaggi, suddivisi in *frames*. Ogni messaggio contiene un address frame che contiene l'indirizzo del target, e uno o più data frames che contengono i dati da trasferire.

**Start e Stop condition** Quando il bus non è utilizzato, entrambe le linee di segnale sono alte. La comunicazione è iniziata da un controller con una `START` condition, in cui porta in basso prima la linea SDA, e poi la linea SCL. Questa sequenza indica che il device controller sta prenotando il bus, fermando gli altri controller.

Quando ha terminato la comunicazione, rilascia la linea SCL, che torna alta, e dopo la linea SDA. Questo indica una `STOP` condition, che permette agli altri controller di iniziare una comunicazione.

I²C usa una sequenza di zero e uno per la comunicazione seriale. SDA è usato per i bit di dato e SCL come linea di clock che dà il tempo alla sequenza. Un uno logico è mandato quando SDA rilascia la linea, rendendola alta. Uno zero è mandato

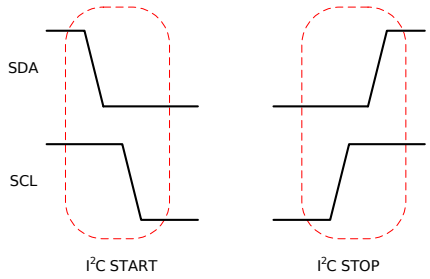



Figura 3.3: Start e Stop Condition [12]

quando SDA abbassa la linea, portandola ad un livello basso. Gli zero e uno sono ricevuti quando SCL sale e scende. Per un bit valido, SDA non deve cambiare tra il fronte di salita del clock e il fronte di discesa. Altrimenti, i cambiamenti di SDA possono essere interpretati come una start o una stop condition.

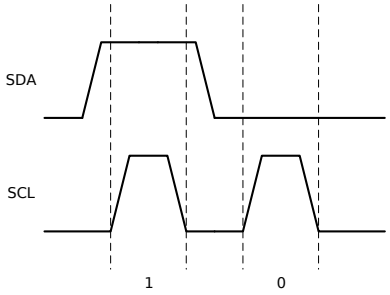



Figura 3.4: Segnali digitali di zero e uno [12]

**Messaggio** Un messaggio I²C si svolge nei seguenti passaggi:

1. Il device controller avvia una `START` condition.
2. Il device controller invia un frame di 8 bit, che contiene 7 bit di indirizzo e un bit di lettura/scrittura. In questo frame, il controller specifica con quale device target vuole comunicare, inviando i bit corrispondenti al suo indirizzo, e se intende effettuare un'operazione di lettura o scrittura.
3. Un nono bit è utilizzato per verificare che la comunicazione ha avuto successo. Il device target porta SDA ad un livello basso per la durata di un ciclo di clock, per indicare il successo.
4. Il frame di indirizzo è seguito da uno o più frame di dati, che sono mandati un byte alla volta.
5. Quando la comunicazione è terminata, il controller manda la `STOP` condition.

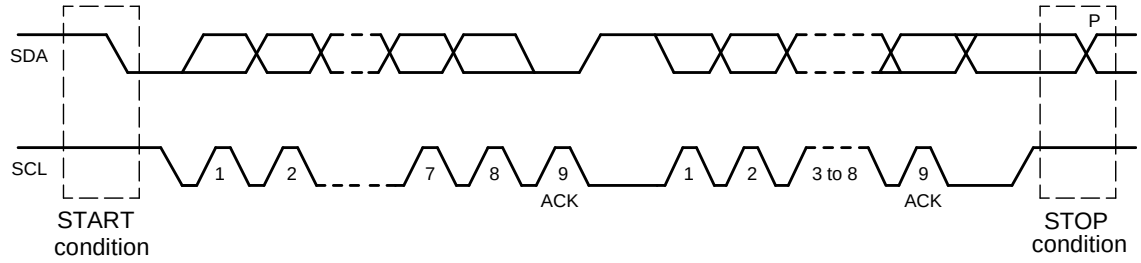



Figura 3.5: Invio di un messaggio [13]

### 3.2.4 SMBus

Il System Management Bus(SMBus) è un protocollo con interfaccia a due linee basato su I²C. È pensato per offrire un bus di controllo per attività di monitoraggio energetico e di sistema, permettendo ad un chip device di dare informazioni sulle sue caratteristiche, riportare errori, accettare opzioni di controllo e dare informazioni di stato. SMBus ha alcune funzionalità aggiuntive rispetto ad I²C, che lo rendono appropriato per l'utilizzo in un dispositivo di monitoraggio e gestione energetica come LTC3350.

- **Gestione degli errori e timeout**: SMBus permette di riconoscere e gestire errori legati al malfunzionamento di un device, pone limiti inferiori sulla frequenza di clock, e pone un timeout sulla durata di un messaggio, in modo da recuperare il funzionamento in seguito al fallimento di un dispositivo.

- **Protocolli SMBus**: i protocolli SMBus sono un modo di inviare comandi ai dispositivi target. Il protocollo I²C si occupa solo di descrivere in che modo spostare i byte da un dispositivo ad un altro, ma SMBus prescrive una struttura per i messaggi.

- **Funzioni di power management**: SMBus include caratteristiche specifiche per leggere lo stato della batteria e altre informazioni di sistema. Inoltre offre delle ottimizzazioni per un utilizzo minore della batteria.

- **Comunicazione da target a controller**: in I²C, il dispositivo target avvia sempre la comunicazione, ma SMBus offre dei protocolli per permettere al target di segnalare di voler comunicare con il controller.

SMBus è stato proposto nel 1995 da Intel per permettere ad un caricabatterie di interagire con il sistema.[14] Questa applicazione prevede una Smart Battery, che contiene dei circuiti che offrono informazione sul caricamento allo Smart Battery Charger. Le informazioni trasmesse includono il voltaggio e la corrente di caricamento, la segnalazione di condizioni di allarme, e altre informazioni sulla temperatura e lo stato chimico della batteria. [15]

### 3.2.5 Protocolli di lettura/scrittura

Ci sono diversi protocolli SMBus, che permettono di leggere e scrivere dati. Il produttore del device può scegliere tra quindici protocolli di comando possibili - LTC3350 implementa `read word` e `write word`. In entrambi questi protocolli, si specifica un *command code*, che può specificare un'istruzione o un indirizzo di 8 bit, da utilizzare a piacere nell'implementazione del dispositivo.

**Write word** Il messaggio che contiene il comando write word è lungo quattro bytes (escludendo i bit di start/stop/ack). Contiene il byte di indirizzo, con il bit di scrittura, seguito dal byte di command code. I restanti due byte contengono i dati da scrivere sul dispositivo. Il target invia un ACK alla ricezione di ogni byte, come previsto dal protocollo I²C, e il messaggio è preceduto e terminato rispettivamente da una `START` condition e una `STOP` condition.

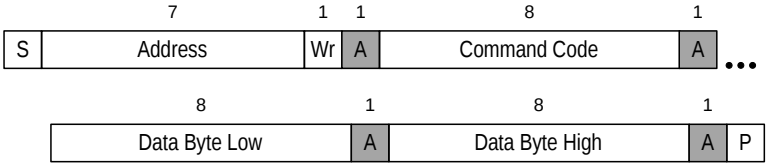



Figura 3.6: protocollo write word [16]

**Read word** Il protocollo `read word` prevede lo scambio di due messaggi. Per prima cosa, il controller deve scrivere un comando sul device target. Il primo messaggio è settato in scrittura e contiene il byte di indirizzo e il byte del command code. Il secondo messaggio contiene di nuovo l'indirizzo, con il bit settato in lettura, e poi i due byte mandati dal target al controller.[16]

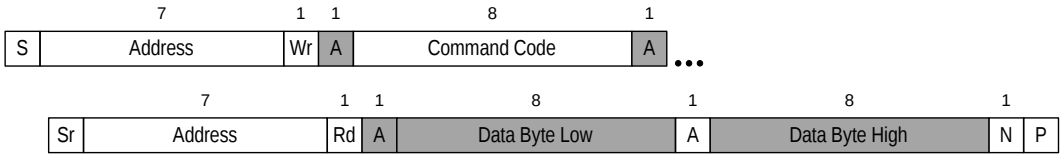



Figura 3.7: Protocollo read word [16]

### 3.2.6 Protocollo SMBus Alert

In I²C, il target device non può avviare la comunicazione, ma SMBus offre la possibilità per il target di segnalare al controller di voler dare inizio alla comunicazione. Questo può avvenire attraverso Host Notify, un protocollo che non vediamo, oppure tramite SMBus Alert. SMBus supporta una terza linea di segnale opzionale, chiamata `SMBALERT#`, che è una linea di interruzione condivisa tra tutti i device.

Se un target device vuole richiedere una comunicazione con il controller, porta la linea `SMBALERT#` al livello basso. Il device processa l'interruzione, e allo stesso tempo accede a tutti i device tramite l'Alert Response Address(ARA), un indirizzo di broadcast riservato. Il device che ha attivato la linea risponde con il proprio indirizzo, mentre gli altri device ignorano il messaggio. In questo modo, il controller può interagire direttamente con il device e gestire la causa del segnale. [16]

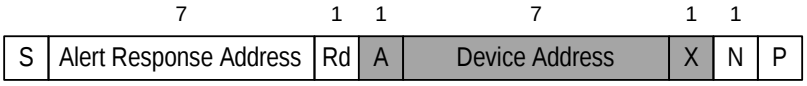



Figura 3.8: Un dispositivo target risponde all'ARA

## 3.3 Analog Devices LTC3350

Un elemento di un sistema embedded può avere la necessità di funzionare anche in caso di una temporanea assenza di corrente, o di effettuare delle operazioni di spegnimento in modo da evitare inconsistenze nel suo stato e malfunzionamenti futuri.

Per questo motivo, il gateway su cui ho lavorato era in possesso di due supercondensatori, in grado di fornire abbastanza corrente da mantenere un funzionamento regolare per 100 secondi(in questo caso specifico), o di tenerlo acceso per abbastanza tempo da terminare i suoi processi e svolgere uno spegnimento normale.

Il device LTC3350 ha lo scopo di gestire il caricamento delle batterie e il passaggio all'alimentazione di backup, e di monitorare lo stato del sistema.

### 3.3.1 Descrizione

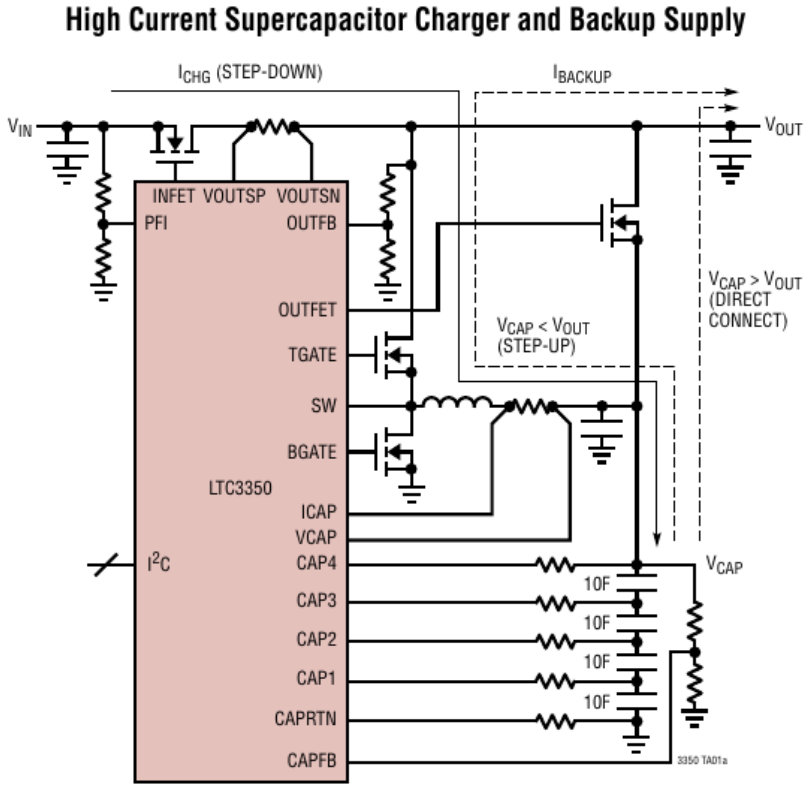



Figura 3.9: Utilizzo tipico di LTC3350

LTC3350 è un gestore dell'alimentazione di backup e un monitor di sistema integrato, per supercondensatori ad alta corrente.

In step-down mode, permette un caricamento con corrente e voltaggio costanti, con un limite programmabile di corrente. In step-up mode, fornisce energia dallo stack di supercondensatori all'alimentazione di backup. Offre un sistema

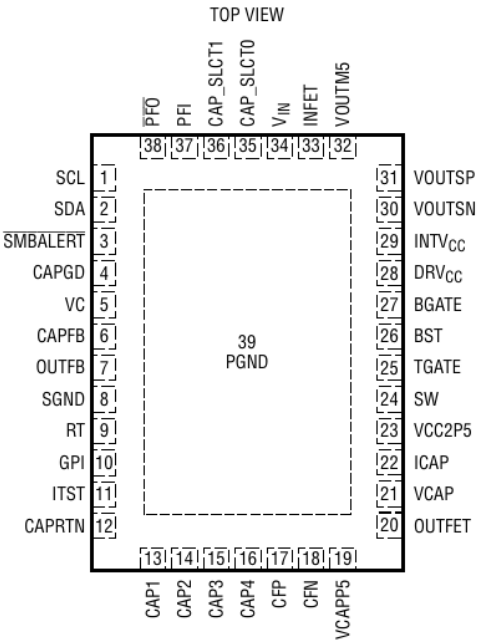
di bilanciamento e protezione da sovratensioni per uno stack da uno a quattro supercondensatori.

Monitora i voltaggi, le correnti, e la temperatura del chip. Contiene un general purpose input pin, per misurare un parametro aggiuntivo. Può misurare la capacitanza e la resistenza dello stack, fornendo informazioni sul loro stato di salute.

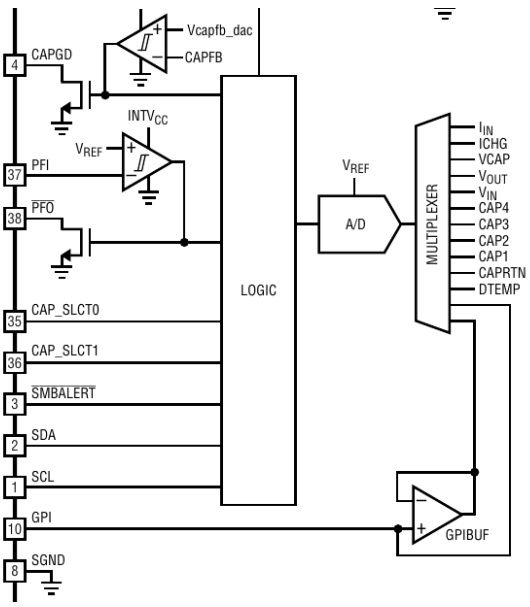
### 3.3.2 Configurazione dei Pin

Il device ha 39 pin, il cui schema è nella figura 3.10a. Alcuni di questi pin sono rilevanti a questo lavoro perchè ne possiamo monitorare il voltaggio, ma tre sono i più importanti per lo sviluppo del driver:

1. **SCL (Pin 1)**: pin di clock per la porta seriale $I^2C$.
2. **SDA (Pin 2)**: pin di scambio di dati bidirezionale per la porta seriale $I^2C$.
3. $\overline{SMBALERT}$ **(Pin 3)**: pin di interruzione.




(a) Configurazione dei pin




(b) sezione del diagramma a blocchi

### 3.3.3 $I^2C$, SMBus e SMBALERT

Il dispositivo contiene una porta $I^2C$/SMBus, che permette di configurarlo e leggere i risultati delle misure.

Ad ogni registro accessibile tramite il bus è associato un sotto-indirizzo di 8 bit, che può essere utilizzato come command code del formato read word/write word. Quindi, quando il controller vuole leggere o scrivere su un registro, deve specificare il suo indirizzo nel byte di command code del protocollo. I registri misurano 16 bit.

L'indirizzo dell'LTC3350 sul bus $I^2C$ è `0b0001001`, ovvero 0x07. Quindi, un protocollo write word sul registro uno di LTC3350 avrebbe la seguente forma:

- START condition.
- Indirizzo di LTC3350: 0x07.
- Bit di scrittura.
- Sotto-indirizzo o command code: 0x01.
- I 16 bit da scrivere sul registro.
- STOP condition.

Tramite queste letture e scritture, è possibile impostare dei limiti ai valori misurati, in modo da ricevere un allarme, ad esempio, quando il chip eccede una determinata temperatura. E' anche possibile impostare un allarme che si attiva al verificarsi un determinato evento di stato, come il collegamento all'alimentazione.

Seguendo il protocollo descritto in 3.2.6, il pin `SMBALERT#` è portato al livello basso al verificarsi di una condizione di allarme. Quando il controller legge dal Alert Response Address, che vale `0b0001100`, cioè 12, LTC risponde con il suo indirizzo, e rilascia il segnale `SMBALERT`, che torna alto.

Tramite questa interfaccia è possibile accedere a diversi registri, tra cui un registro che descrive gli allarmi attivi, e due che descrivono lo stato del sistema, oltre a tutti i registri che rappresentano le misure e i limiti su di esse.

### 3.3.4 Misurazioni

**Analog to Digital Converter** LTC ha un Analog to Digital Converter(ADC) integrato, che misura automaticamente, in modo periodico, i valori di 11 canali. I valori ottenuti sono scritti in un registro di 16 bit come numeri in complemento a due. I due bit più bassi sono sub-bits, cioè output con troppo rumore per essere utilizzati in una singola conversione, ma utilizzabili in misure ripetute. Alcuni risultati dell'ADC sono usati dal dispositivo per il bilancio e la protezione dello stack.

**Misura di Capacitanza e Resistenza** LTC3350 può misurare la capacitanza e la resistenza equivalente in serie(ESR) dello stack di condensatori. Questo test può richiedere diversi minuti e non è eseguito automaticamente, ma dev'essere iniziato attraverso un comando esplicito, o programmato, tramite una scrittura sui registri. Inoltre, le prime misure possono non essere accurate, ed è consigliato di effettuare i test diverse volte al primo powerup del sistema, per ottenere risultati di qualità.

È possibile impostare la frequenza delle misurazioni attraverso il registro `cap_esr_per`.

Il test di capacitanza è eseguito solo dopo che i supercondensatori hanno finito di caricare. Il test disabilita temporaneamente il caricatore, e poi fa scaricare i capacitori di 200mV. Il tempo di scaricamento è misurato ed usato per calcolare la capacitanza, con il risultato scritto nel registro `meas_cap`.

Il test ESR è eseguito immediatamente dopo il test di capacitanza. Il controller di commutazione, cioè il circuito che controlla il flusso di corrente, viene acceso e spento più volte. Questa azione provoca variazioni nella corrente di carica e tensione nel condensatore, che sono misurate. Queste sono usate per calcolare l'ESR relativo alle resistenza applicate al circuito.

Le misure di capacitanza e ESR possono fallire per diversi motivi, e in questo caso i bit `mon_cap_failed` o `mon_esr_failed` saranno attivi nel registro `mon_status`.

### 3.3.5 Registri

Nella tabella 3.1, si trova la lista completa dei registri. LTC3350 ha 38 registri, di cui 17 in sola lettura, che rappresentano varie misure e informazioni di stato del dispositivo, e 21 in lettura e scrittura, che permettono di impostare gli allarmi e le opzioni di configurazione.

In seguito la descrizione di alcuni dei registri più interessanti.

**Enable/mask alarms register** Il registro `msk_alarms` consente di scegliere quali allarmi attivare, tramite scrittura su uno dei suoi sedici bit. Se il bit corrispondente ad un allarme su questo registro è uno, e si verifica la condizione corrispondente, allora viene inviato un SMBus Alert.

**Alarm register** Il registro `alarm_reg`, di sola lettura, indica quali allarmi sono stati attivati. Permette di identificare la causa di un SMBus Alert: il controller deve leggere questo registro per individuare la condizione di allarme. Ad ognuno dei sedici bit corrisponde un allarme, e un bit del registro `msk_alarms`.

**Clear alarms register** Il registro `clr_alarms` permette di ripulire un allarme che si è verificato, una volta che è terminata la condizione di allarme. Scrivere uno su un bit di questo registro ripulisce il corrispondente allarme sul registro degli allarmi.

**VCAP voltage reference DAC setting** Il VCAP DAC è un convertitore digitale-analogico a 4 bit integrato nel circuito. La sua funzione è di impostare una tensione di riferimento per il punto di retroazione di VCAP, chiamato CAPFB. Questo registro permette di adattare dinamicamente la tensione di carica dei supercondensatori per compensare l'invecchiamento e la perdita di capacità, prolungando l'efficacia del sistema di backup.

**Monitor status register** Il registro `mon_status` contiene dei bit che contengono informazioni sul sistema di monitoraggio di capacitanza e ESR. Questi bit sono attivati e disattivati in corrispondenza di alcuni eventi legati alle misurazioni. Scrivere sul registro `msk_mon_status`, fa attivare una SMBus alert quando il bit corrispondente di `mon_status` si attiva.

**Control status register** Il registro `ctl_reg` permette di comandare una misurazione di capacitanza e ESR, di terminare una misurazione attiva, di attivare un input di buffer per il valore del pin di GPI, o di modificare la scala di misurazione della capacitanza.

| Registro | indirizzo | R/W | bits | Descrizione |
|---|---|---|---|---|
| clr_alarms | 0x00 | R/W | 15:0 | Clear alarms register |
| msk_alarms | 0x01 | R/W | 15:0 | Enable/mask alarms register |
| msk_mon_status | 0x02 | R/W | 9:0 | Enable/mask monitor status alerts |
| cap_esr_per | 0x04 | R/W | 15:0 | Capacitance/ESR measurement period |
| vcapfb_dac | 0x05 | R/W | 3:0 | VCAP voltage reference DAC setting |
| vshunt | 0x06 | R/W | 15:0 | Capacitor shunt voltage setting |
| cap_uv_lvl | 0x07 | R/W | 15:0 | Capacitor undervoltage alarm level |
| cap_ov_lvl | 0x08 | R/W | 15:0 | Capacitor overvoltage alarm level |
| gpi_uv_lvl | 0x09 | R/W | 15:0 | GPI undervoltage alarm level |
| gpi_ov_lvl | 0x0A | R/W | 15:0 | GPI overvoltage alarm level |
| vin_uv_lvl | 0x0B | R/W | 15:0 | VIN undervoltage alarm level |
| vin_ov_lvl | 0x0C | R/W | 15:0 | VIN overvoltage alarm level |
| vcap_uv_lvl | 0x0D | R/W | 15:0 | VCAP undervoltage alarm level |
| vcap_ov_lvl | 0x0E | R/W | 15:0 | VCAP overvoltage alarm level |
| vout_uv_lvl | 0x0F | R/W | 15:0 | VOUT undervoltage alarm level |
| vout_ov_lvl | 0x10 | R/W | 15:0 | VOUT overvoltage alarm level |
| iin_oc_lvl | 0x11 | R/W | 15:0 | IIN overcurrent alarm level |
| ichg_uc_lvl | 0x12 | R/W | 15:0 | ICHG undercurrent alarm level |
| dtemp_cold_lvl | 0x13 | R/W | 15:0 | Die temperature cold alarm level |
| dtemp_hot_lvl | 0x14 | R/W | 15:0 | Die temperature hot alarm level |
| esr_hi_lvl | 0x15 | R/W | 15:0 | ESR high alarm level |
| cap_lo_lvl | 0x16 | R/W | 15:0 | Capacitance low alarm level |
| ctl_reg | 0x17 | R/W | 3:0 | Control register |
| num_caps | 0x1A | R | 1:0 | Number of capacitors configured |
| chrg_status | 0x1B | R | 11:0 | Charger status register |
| mon_status | 0x1C | R | 9:0 | Monitor status register |
| alarm_reg | 0x1D | R | 15:0 | Active alarms register |
| meas_cap | 0x1E | R | 15:0 | Measured capacitance value |
| meas_esr | 0x1F | R | 15:0 | Measured ESR value |
| meas_vcap1 | 0x20 | R | 15:0 | Measured capacitor one voltage |
| meas_vcap2 | 0x21 | R | 15:0 | Measured capacitor two voltage |
| meas_vcap3 | 0x22 | R | 15:0 | Measured capacitor three voltage |
| meas_vcap4 | 0x23 | R | 15:0 | Measured capacitor four voltage |
| meas_gpi | 0x24 | R | 15:0 | Measured GPI pin voltage |
| meas_vin | 0x25 | R | 15:0 | Measured VIN voltage |
| meas_vcap | 0x26 | R | 15:0 | Measured VCAP voltage |
| meas_vout | 0x27 | R | 15:0 | Measured VOUT voltage |
| meas_iin | 0x28 | R | 15:0 | Measured IIN current |
| meas_ichg | 0x29 | R | 15:0 | Measured ICHG current |
| meas_dtemp | 0x2A | R | 15:0 | Measured die temperature |

Tabella 3.1: Registri

# Capitolo 4

# Stato dell'Arte

## 4.1 Software per LTC3350

LTC3350 è un dispositivo per il quale non è stato creato molto software: questo è dovuto alla sua utilità come hardware,al di fuori delle sue capacità di monitoraggio, ma anche alla specificità delle sue applicazioni. Analog Devices, il suo produttore, offre un software development kit per l'LTC3350, che permette al programmatore di accedere alle funzioni del chip estremamente a basso livello.

Il software development kit[17] include una lista e una descrizione dei registri in un file HTML, una libreria per C e una libreria per Python, con due esempi di utilizzo.

Lo scopo di questo SDK è di aiutare gli sviluppatori futuri a sviluppare applicazioni che utilizzano LTC3350. Tuttavia, le sue librerie presentano delle criticità che hanno reso necessario lo sviluppo di un driver.

**Applicazione in userspace** Svolgere interazione diretta con l'hardware in userspace comporta l'assenza di astrazione e di portabilità, e l'incapacità di utilizzare funzionalità offerte dal sistema operativo, come la gestione della concorrenza e le interruzioni.

**Richiede conoscenze pregresse** Per l'utilizzo del SDK, è previsto che lo sviluppatore al livello applicativo conosca il protocollo $I^2C$ e la configurazione della propria seriale, e voglia implementare le funzioni di lettura e scrittura sul bus. Il driver astrae completamente questi aspetti. Tutte le informazioni necessarie per utilizzarlo si trovano sulla datasheet di LTC3350.

**Specifico ai linguaggi di programmazione** Esistono due API per due linguaggi di programmazione, ma il driver permette di scrivere applicazioni in qualsiasi linguaggio di programmazione che sappia leggere e scrivere da file.

**Gestione di SMBus alert assente** Il kernel di Linux permette la gestione di interruzioni in modo semplice ed elegante, ed è ottimale per la gestione di SMBus alert. Ancora una volta, il kernel può gestire l'interruzione in modo trasparente all'utente e notificarlo tramite meccanismi consolidati e testati. L'API fornita non ha supporto per gli alerts. Nella sezione di implementazione, parlerò più nel dettaglio dei vantaggi del mio sistema.

## 4.2 Driver hwmon

Il sottosistema di hardware monitoring del kernel di Linux contiene drivers per dispositivi simili a LTC3350, come ad esempio LTC4222, che offre funzionalità di lettura e scrittura da registri simili a quelli del nostro device. Tuttavia, si tratta di chip più semplici, che effettuano solo operazioni hardware monitoring, come la misura dei voltaggi, e non hanno la stessa complessità di questo sistema. Inoltre, non implementano gli alerts.

Lo standard hwmon offre macro e funzioni per semplificare l'allocazione di strutture dati e le interazioni con gli oggetti kernel, ma lo standard hwmon è pensato per dispositivi con funzionalità più limitate. Un esempio tipico di driver hwmon è il driver per i sensori di temperatura, che permette di accedere alle temperature dei chip e segnala eventualmente temperature estreme. Oppure, un chip può misurare e comunicare le tensioni di ingresso e di uscita, o la corrente che lo attraversa.

**lm-sensors** `lm-sensors` è un pacchetto software utilizzato in sistemi Linux per monitorare in tempo reale vari parametri hardware nel sistema, come la temperatura dei componenti, la velocità delle ventole e le tensioni di alimentazione. Nel suo utilizzo più semplice, il software effettua una ricerca dei device hwmon, e quando viene chiamato stampa i valori dei loro attributi. Perché funzioni, è necessario che gli attributi seguano uno standard di nomenclatura.

| Voltage Parameter | Description |
|---|---|
| `in[0-*]_min` | Voltage min value. |
| `in[0-*]_lcrit` | Voltage critical min value. |
| `in[0-*]_max` | Voltage max value. |
| `in[0-*]_crit` | Voltage critical max value. |
| `in[0-*]_input` | Voltage input value. |

Tabella 4.1: Nomi standard per le tensioni di ingresso.

Nel caso di LTC3350, non esiste una corrispondenza diretta tra i registri e i tipici attributi hwmon. Soprattutto, ci sono registri che non corrispondono ad attributi standard, come il monitor status, e ci sono registri che permettono di influenzare il funzionamento del circuito, come `vshunt` e `ctl_reg`.

Ho anche pensato che seguire gli standard di nomenclatura avrebbe reso più difficile per lo sviluppatore di una applicazione al lato utente capire le corrispondenze tra registri e attributi. Per questo motivo ho deciso di non garantire la compatibilità del driver con lm-sensors.

## 4.3 Modulo SMBUS

Come visto in nella sezione 3.1.1, il protocollo SMBus aggiunge delle funzionalità al protocollo $I^2C$, che permettono al dispositivo target di segnalare delle condizioni al dispositivo controller. Questi protocolli, SMBus Alert e SMBus Host Notify, non sono presenti in tutti i dispositivi $I^2C$, ma sono totalmente opzionali. Di conseguenza, gli sviluppatori del kernel hanno deciso di situare la loro gestione in un modulo separato, chiamato I2C-SMBUS.[18]

Il modulo contiene l'implementazione di funzioni utili all'uso del protocollo SMBus e un driver per gestire SMBus Alert, chiamato `smbalert-driver`. Per scegliere se usarlo nel mio progetto, ho studiato il suo funzionamento interno, che è un esempio interessante di gestione delle interruzioni e di comunicazione con periferiche tramite driver.

Il modulo offre due modi per gestire gli alert: uno è di utilizzare `smbalert-driver` per la gestione delle interruzioni, l'altro è di gestire le interruzioni dal bus controller driver.[19]

### 4.3.1 smbalert-driver setup

Utilizzare l'interrupt handler di `smbalert-driver` è la soluzione più semplice, ed è l'unica se non si vuole andare a modificare il driver del bus controller. `smbalert-driver` gestisce sia la fase di richiesta della linea di interruzione, che l'interrupt handling.

Durante la fase di registrazione dell'adapter, cioè del driver per il bus controller, il kernel verifica tramite il device tree, se il bus sia capace di ricevere interruzioni chiamate `smbus-alert`.

```
1 &i2c3 {
2   clock-frequency = <400000>;
3   pinctrl-names = "default";
4   pinctrl-0 = <&pinctrl_i2c3>;
5   status = "okay";
6   interrupts-extended = <&gic GIC_SPI 37 IRQ_TYPE_LEVEL_HIGH
     >,
7     <&gpio1 10 IRQ_TYPE_LEVEL_LOW>;
8   interrupt-names = "irq", "smbus_alert";
```

Listato 4.1: Rappresentazione in device tree di bus $I^2C$

Nell'esempio riportato sopra, il bus $I^2C$ supporta due interrupt request: una generica, chiamata `irq`, sulla linea 37, e una dedicata a SMBus Alert, chiamata `smbus_alert`, sulla linea 10. Quando ho implementato il driver, è stato il mio dovere quello di aggiornare l'entry nel device tree del bus $I^2C$, per descrivere l'esistenza di questa linea IRQ.

Se il modulo I2C-SMBUS è stato caricato, ed è presente un'interruzione chiamata `smbus_alert`, è chiamata la funzione `i2c_new_smbus_alert_device`.

```
1  struct i2c_client *i2c_new_smbus_alert_device(
2      struct i2c_adapter *adapter,
3      struct i2c_smbus_alert_setup *setup)
4  {
5    struct i2c_board_info ara_board_info = {
6      I2C_BOARD_INFO("smbus_alert", 0x0c),
7      .platform_data = setup,
8    };
9
10   return i2c_new_client_device(adapter, &ara_board_info);
11 }
```

Listato 4.2: creazione device smbus_alert

Questa funzione crea un nuovo dispositivo "virtuale" il cui scopo sarà di permettere al controller di scrivere all'Alert Response Address(ARA). Non esiste davvero un dispositivo client all'indirizzo `0x0c`, ma durante l'esecuzione di SMBus Alert, il controller effettuerà una lettura da questo indirizzo. Creare un dispositivo virtuale permette di utilizzare le funzioni di libreria per leggere da un indirizzo riservato.

La funzione `i2c_new_smbus_alert_device` prende come parametri la struct vuota `setup` e un riferimento all'adapter, cioè al driver del bus controller. Registra un nuovo dispositivo client di tipo `smbus_alert`, all'indirizzo `0x0c`. Restituisce un riferimento a questo nuovo dispositivo, il cui driver è proprio `smbalert_driver`.

Come per ogni altro driver, è chiamata la funzione di probing. Questa, oltre alle normali inizializzazioni di strutture dati, svolge due operazioni interessanti.

```
1  static int smbalert_probe(struct i2c_client *ara,
2          const struct i2c_device_id *id)
3  {
4    ...
5    struct i2c_adapter *adapter = ara->adapter;
6    int res, irq;
7
8    /* Allocazione strutture dati */
9    ...
10   /* Ricerca numero irq sul device tree */
11   ...
12   INIT_WORK(&alert->alert, smbalert_work);
13   alert->ara = ara;
```

```
14
15   if (irq > 0) {
16     res = devm_request_threaded_irq(&ara->dev, irq,
17             NULL, smbus_alert,
18             IRQF_SHARED | IRQF_ONESHOT,
19             "smbus_alert", alert);
20     if (res)
21       return res;
22   }
23   i2c_set_clientdata(ara, alert);
24   dev_info(&adapter->dev, "supports SMBALERT#\n");
25   return 0;
26 }
```

Listato 4.3: Funzione di probing di `smbalert-driver`

La prima è l'inizializzazione di una *work queue*, cioè di una coda di lavoro. Una *work queue* è un meccanismo tramite cui è possibile effettuare lo *scheduling* dei task per eseguirli come processi. Permette di rinviare l'esecuzione di un task di gestione di un'interruzione ad un momento in cui non stanno essendo svolte operazioni critiche. Ogni volta che avviene lo scheduling di una nuova task, è chiamata la funzione `smbus_alert()`.

La seconda è la richiesta di allocare una linea di interruzione per il device ARA. Viene chiamata la funzione `devm_request_threaded_irq` con i seguenti parametri:

1. **dev**, il dispositivo che richiede la linea IRQ, ovvero il device ARA.
2. **irq**, linea IRQ da allocare, cioè il numero scritto nel device tree.
3. **handler**, l'interrupt handler da svolgere immediatamente durante la interrupt service routine, in questo caso **null**, perchè non ci sono operazioni urgenti da svolgere. Viene chiamato un handler che termina immediatamente.
4. **thread_fn**, una funzione da svolgere come bottom-half della gestione dell'interruzione. La sua esecuzione sarà riinviata. Si chiama `smbus_alert()` ed è definita nel modulo I2C-SMBUS.
5. **irqflags**, le flag che indicano la tipologia di interrupt. `SHARED` indica che la linea di interruzione è condivisa tra diversi dispositivi, e `ONESHOT` indica che la linea di interruzione sarà disattivata (masked) finchè non è eseguito il thread di gestione interrupt.
6. **devname**, il nome del dispositivo ARA, cioè `smbus_alert`.
7. **dev_id**, un cookie che contiene dati utili alla gestione dell'interruzione, chiamato `alert`.

La richiesta di IRQ ha successo, e d'ora in poi, ogni volta che la linea `SMBALERT#` sarà portata al livello basso, l'interrupt handler chiamerà la funzione `smbus_alert()` in un nuovo thread.

### 4.3.2 Gestione con smbalert-driver

Il gestore interruzioni degli alert deve essere eseguito in un scheduled thread perchè non si possono svolgere chiamate SMBus nel kernel code path, come visto in 2.5.1, Interrupt Handling. La `smbus_alert()` è chiamata con parametri il numero IRQ, e la struttura dati alert che avevamo passato come cookie.

Secondo il protocollo, il dispositivo master legge dall'Alert Response Address. I dispositivi con pending alerts rispondono in ordine del loro indirizzo, dal più basso al più alto. Dopo aver risposto, il dispositivo rilascia la linea `SMBUSALERT#`. Il contenuto del messaggio letto è l'indirizzo del dispositivo che ha causato l'alert.

```
1  static irqreturn_t smbus_alert(int irq, void *d)
2  {
3    struct i2c_smbus_alert *alert = d;
4    struct i2c_client *ara = alert->ara;
5
6    for (;;) {
7      s32 status;
8      struct alert_data data;
9      status = i2c_smbus_read_byte(ara);
10     if (status < 0)
11       break;
12
13     data.data = status & 1;
14     data.addr = status >> 1;
15     data.type = I2C_PROTOCOL_SMBUS_ALERT;
16
17     dev_dbg(&ara->dev, "SMBALERT# from dev 0x%02x, flag %d\n"
     ,
18       data.addr, data.data);
19
20     /* Notify driver for the device which issued the alert */
21     device_for_each_child(&ara->adapter->dev, &data,
22               smbus_do_alert);
23   }
24
25   return IRQ_HANDLED;
26 }
```

Listato 4.4: gestione interruzione

L'handler chiama la funzione `smbus_do_alert` per ogni dispositivo figlio del bus controller, cioè per ogni target device attaccato al bus $I^2C$, passando come parametro l'indirizzo del device che ha attivato il segnale.

`smbus_do_alert()` controlla se il device su cui è stata chiamata la funzione corrisponde al device che ha segnalato l'alert, e in quel caso chiama la sua funzione `alert()`. Vedremo nel capitolo 7, Implementazione, la funzione di alert di LTC3350.

### 4.3.3 Bus controller driver

Pochi driver per bus controller decidono di implementare una gestione personalizzata delle interruzioni per SMBus Alert. Uno di questi è `i2c-stm32f7`, driver per STMicroelectronics STM32F7 $I^2C$ controller.[20][21]

STM32F7 è un microcontroller, cioè un piccolo computer su un singolo circuito integrato, la cui architettura permette una gestione specifica di alcune interruzioni. Dal punto di vista hardware, ha dei registri per gestire le interruzioni da SMBus Alert, il che ha creato la necessità per un gestore di interruzioni personalizzato.

**Setup** La funzione di probing di `stm32f7` verifica se è presente la proprietà `smbus-alert` sul suo nodo del device tree. Se è presente, invoca la propria funzione `enable_smbus_alert`. Questa invoca `i2c_new_smbus_alert_device()` del modulo I2C-SMBUS.

Al ritorno con successo da questa funzione, il driver `stm32f7` attiva gli alert scrivendo sui propri registri dedicati.

**Gestione interruzioni** `stm32f7` prevede un meccanismo interno per riconoscere le interruzioni: l'avvenimento di un errore o di un alert causa, tramite meccanismi hardware, la scrittura su uno dei suoi registri interni. Di conseguenza, quando riceve un'interruzione, deve controllare i valori dei suoi registri per poter riconoscere la causa.

```
1 if (status & STM32F7_I2C_ISR_ALERT) {
2   dev_dbg(dev, "<%s>: SMBus alert received\n", __func__);
3   writel_relaxed(STM32F7_I2C_ICR_ALERTCF, base +
     STM32F7_I2C_ICR);
4   i2c_handle_smbus_alert(i2c_dev->alert->ara);
5   return IRQ_HANDLED;
6 }
```

Listato 4.5: IRQ handler

Nel blocco di codice, il driver verifica che sia avvenuto un alert tramite lettura da registro. Tramite una scrittura, segnala di star gestendo l'interruzione, poi chiama `i2c_handle_smbus_alert` del modulo I2C-SMBUS.

La funzione mette in coda un nuovo task, che svolge il ruolo della bottom-half del gestore di interruzioni. Questa task chiamerà a sua volta la funzione `alert()` del driver device client.

# Capitolo 5

# Tecnologie utilizzate

La realizzazione di questo progetto ha reso necessario imparare ad utilizzare le tecnologie appropriate, che includono i metodi per effettuare la compilazione e l'utilizzo del kernel, ma anche le librerie e i driver necessari per programmarlo.

## 5.1 Comunicazione seriale

Il Dynagate su cui ho lavorato era una board che non include un display integrato. Per utilizzarla, ho dovuto utilizzare la comunicazione seriale, un metodo di trasmissione dei dati in cui le informazioni vengono inviate un bit alla volta su un singolo canale di comunicazione. In questo caso, ho utilizzato una connessione USB con il mio PC, e il terminal multiplexer `screen` per leggere e scrivere sul Dynagate dal terminale del mio computer.

## 5.2 Yocto build system

Lo Yocto Project è un'iniziativa open source per lo sviluppo di sistemi operativi Linux personalizzati per qualsiasi architettura hardware. Fornisce una serie di strumenti, metodi e best practices che consentono agli sviluppatori di creare immagini di Linux specializzate a dispositivi IoT ed Embedded, o a qualsiasi architettura che necessiti di una distribuzione Linux personalizzata.

Il suo build system si basa su *Bitbake*, un motore che gestisce il processo di compilazione e assemblaggio delle applicazioni e delle componenti di sistema. La sua unità software è la ricetta, un file scritto nel linguaggio di configurazione `.bb`, che definisce come costruire un particolare pacchetto software, specificando le dipendenze e le istruzioni di compilazione. Questo approccio modulare consente di mantenere e riutilizzare facilmente le ricette tra progetti diversi.

Un'altra caratteristica distintiva dello Yocto Build System è la sua architettura a strati, che consente agli sviluppatori di estendere e personalizzare il sistema senza

modificare direttamente il codice sorgente di base. I layer possono contenere ricette, configurazioni e metadati specifici per un determinato hardware o applicazione.

```
.
├── meta-browser
├── meta-clang
├── meta-eurotech
│   ├── classes
│   ├── common
│   ├── conf
│   ├── COPYING.MIT
│   ├── Documentation
│   ├── dynamic-layers
│   ├── files
│   ├── meta-eurotech-nxp
│   │   ├── meta-dynagate-10-14
│   │   │   ├── classes
│   │   │   ├── conf
│   │   │   ├── COPYING.MIT
│   │   │   ├── files
│   │   │   ├── meta-dynagate-10-14-sato
│   │   │   ├── README.md
│   │   │   ├── recipes-bsp
│   │   │   ├── recipes-connectivity
│   │   │   ├── recipes-core
│   │   │   ├── recipes-extended
│   │   │   ├── recipes-kernel
│   │   │   ├── recipes-navigation
│   │   │   ├── recipes-support
│   │   │   └── wic
│   │   └── meta-reliagate-10-14
│   └── meta-eurotech-ti
├── meta-freescale
├── meta-freescale-distro
├── meta-imx
├── meta-iotedge
├── meta-openembedded
├── meta-rauc
├── meta-rust
├── meta-selinux
└── meta-virtualization
```

Figura 5.1: Struttura directories

Come si può vedere nella figura 5.1, le directories che contengono le ricette sono organizzate in strati, il cui schema segue le diverse parti di un sistema, e si suddivide tra diverse architetture. Grazie a questo sistema, ho inserito il mio driver nel codice sorgente del kernel, e ho compilato il kernel utilizzando le opzioni di configurazione che erano state impostate dai miei collaboratori nelle versioni precedenti.

## 5.3 Linguaggi di programmazione e librerie

Il kernel di Linux è scritto quasi interamente in C, per il suo equilibrio tra prestazioni, portabilità e controllo a basso livello. Il driver su cui ho lavorato non fa eccezione, così come tutte le librerie che utilizza. Per l'applicazione in userspace, è stato possibile scegliere tra più linguaggi di programmazione, ma ho scelto il C per il

maggiore controllo a basso livello. Una piccola parte del software di testing è scritta in python per agevolare la manipolazione di stringhe e la scrittura su files di output.

### 5.3.1 Librerie del kernel Linux

La Linux Kernel API (Application Programming Interface) è un insieme di funzioni, macro, strutture dati e convenzioni che permette agli sviluppatori di eseguire operazioni in ambiente kernel. Le librerie standard del kernel forniscono alternative ottimizzate e sicure per operazioni come la gestione per la memoria, la manipolazione di stringhe, operazioni su bit e strutture dati. Queste formano la Core API, l'interfaccia che permette di eseguire operazioni di base condivise tra tutti i sottosistemi.

**Driver API** Ogni sottosistema ha la propria API con una libreria di funzioni specifiche. In questo progetto ho utilizzato la libreria del sottosistema $I^2C$ e SMBus, la hardware monitoring API, e la Linux filesystem API.

**Sottosistema $I^2C$ e SMBus** Il sottosistema $I^2C$ e SMBus offre un'interfaccia di programmazione sia il lato controller che per il lato target del protocollo. L'interfaccia prevede due tipi di driver e due tipi di device. Un *Adapter Driver* $I^2C$ è un'astrazione dell'hardware controller, è legato ad un platform device, ed espone una `struct i2c_adapter` per rappresentare ciascun bus $I^2C$ che gestisce. Su ogni bus ci saranno dispositivi $I^2C$ rappresentati dalla `struct i2c_client`. Questi device saranno legati ad una `struct i2c_driver` che segue il Linux driver model nel modo standard.[22]

**Hardware Monitoring kernel API** La pagina sulla documentazione ufficiale di Linux della Hardware Monitoring API legge «Questo documento non descrive cosa sia un driver o un Device di Hardware Monitoring.»[23]

Secondo Linux Driver Development[2], il sottosistema hwmon è pensato per sensori utilizzati per monitorare e controllare il sistema stesso, come il controllo delle ventole e il monitoraggio della temperatura. Questa API offre funzioni per registrare dispositivi e dichiarare attributi sysfs.

### 5.3.2 Sysfs

Sysfs è un filesystem virtuale che esporta le informazioni sui dispositivi e i driver dal kernel device model allo userspace. Offre un modo di esportare le strutture dati del kernel, i loro attributi e le relazioni tra di esse allo spazio utente.[2]

**kernel object** Il kernel object, o *kobject* è la struttura fondamentale che tiene insieme il device model.[5] Il suo scopo originale fu di permettere il reference counting

degli oggetti: dà un meccanismo per tenere traccia del ciclo di vita degli oggetti del kernel. Quando non ci sono più riferimenti ad un oggetto, può essere eliminato.

Ogni oggetto esposto tramite sysfs ha, al di sotto, un kobject che interagisce con il kernel per crearne una rappresentazione visibile.

Un kobject è quasi sempre integrato in una struttura dati più grande. Volendolo interpretare in termini di programmazione object-oriented, i kobject possono essere visti come una classe top-level e astratta, da cui sono derivate molte altre classi.[24]

Viceversa, per ogni kobject registrato nel sistema, è creata una directory in sysfs. Questa directory è creata come subdirectory del kobject genitore, esprimendo le gerarchie degli oggetti allo userspace. Le directory top-level in sysfs rappresentano i sottosistemi a cui gli oggetti appartengono, ad esempio `bus/`, `device/`, o `module/`.[25]

**Attributi** Gli attributi di un kobject possono essere esportati sotto forma di files regolari nel filesystem. Sysfs "traduce" le operazioni I/O sui file a dei metodi definiti per gli attributi, dando un modo di leggere e scrivere dagli attributi del kernel. Gli attributi devono essere file di testo ASCII. Nella sezione 7.1.2 vedremo come questo si applica agli attributi di un driver hwmon.

## 5.4 Driver I²C e SMBus

Come anticipato nella sezione 2.4.3, ci sono diversi passaggi, e diversi driver, che contribuiscono all'impostazione di un dispositivo. Analizzo questi passaggi usando come esempio LTC3350, nella sua gestione delle interruzioni.

Il driver per un dispositivo può essere compilato come modulo e caricato alla necessità, come nel caso del collegamento di un dispositivo esterno. Nel caso di un circuito integrato come LTC3350, è preferibile inserire la sua configurazione nel Device Tree, in modo da caricarlo in automatico.

Durante l'inizializzazione del kernel, il device tree è de-serializzato in una rappresentazione a runtime efficace. E' chiamata la funzione `of_platform_populate()`, che attraversa i nodi del device tree e ne crea "platform devices", cioè dispositivi integrati nel sistema, che non sono connessi e riconosciuti dinamicamente. Questi dispositivi sono collegati ad un bus virtuale chiamato "platform bus".

**I2C IMX** Un esempio di platform device è l'I²C bus controller. Per ogni bus I²C, viene registrato un controller driver, ad esempio `i2c-imx`. Questo è un adapter, ovvero gestisce la comunicazione tra il bus controller e i dispositivi client. Contiene insieme di funzioni specifiche che effettuano letture e scritture sugli indirizzi I/O hardware del controller I²C. Quando viene riconosciuto un bus I²C, viene chiamata la funzione `probe()`.

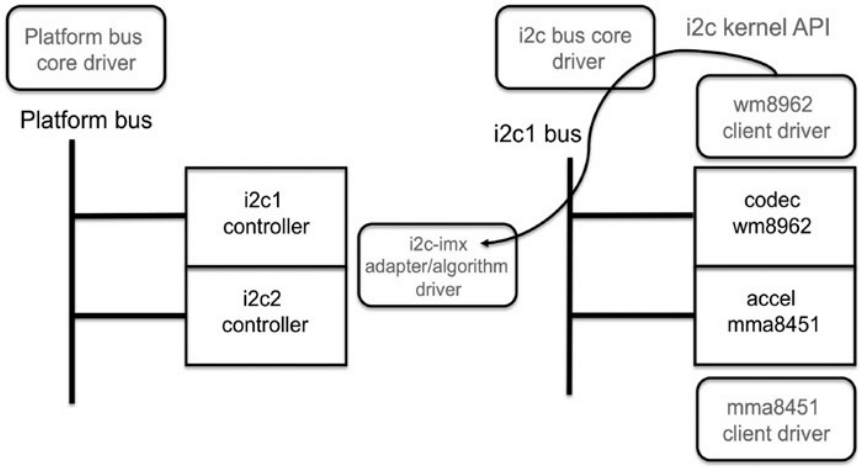



Figura 5.2: Bus e driver corrispondenti[2]

Stampe su `dmesg`:

```
[2.786565] i2c_dev: i2c \dev entries driver
[3.347636] imx-i2c 30a20000.i2c: GPIO lookup for consumer sda
[3.347710] imx-i2c 30a20000.i2c: GPIO lookup for consumer scl
[3.373648] i2c i2c-0: IMX I2C adapter registered
[3.378815] imx-i2c 30a30000.i2c: GPIO lookup for consumer sda
[3.347710] imx-i2c 30a20000.i2c: GPIO lookup for consumer scl
[3.535141] i2c i2c-1: IMX I2C adapter registered
[3.540496] imx-i2c 30a40000.i2c: GPIO lookup for consumer sda
[3.540571] imx-i2c 30a40000.i2c: GPIO lookup for consumer scl
[3.543217] i2c i2c-2: supports SMBALERT#
[3.579951] i2c i2c-2: IMX I2C adapter registered
```

Listato 5.1: Stampe su dmesg di inizializzazione adapter I²C

Queste stampe mostrano la registrazione di tre IMX I2C adapters per tre bus I²C. La registrazione di un adapter contiene diverse operazioni non rilevanti all'ambito del progetto, tra cui il GPIO lookup per le linee SDA e SCL dei bus I²C. Come si può vedere, nella registrazione dell'adapter per il terzo bus I²C, viene chiamato il smbalert-driver, descritto nella sezione 4.3.1, che effettua la stampa `supports SMBALERT#`, come mostrato nel listato 4.3.

**I²C bus core** Il core driver del sistema I²C è solitamente caricato automaticamente in quanto driver built-in. Il suo ruolo è di offrire una interfaccia tra il driver del client e il driver del controller. La I²C Core API è un insieme di funzioni usate dall'I²C client device driver per lo scambio di dati con un dispositivo collegato al bus I²C.

La funzione di probing del controller driver chiama `i2c_add_numbered_adapter()`, che lo inserisce nelle strutture dati del bus core driver, e registra il bus come dispositivo. Questo significa inizializzare tutte le strutture dati necessarie alla gestione del dispositivo.

```
&i2c3 {
  clock-frequency = <400000>;
  status = "okay";
  interrupts-extended = <&gic GIC_SPI 37
   IRQ_TYPE_LEVEL_HIGH>,
    <&gpio1 10 IRQ_TYPE_LEVEL_LOW>;
  interrupt-names = "irq", "smbus_alert";
```

Listato 5.2: Configurazione del device tree

**Client device** Il core I²C itera sui nodi figli del bus controller, verificando le corrispondenze tra le stringhe "compatible" del device tree e le compatibilità indicate nella board info dei driver client.

```
ltc3350: ltc3350@09 {
  compatible = "ltc3350";
  reg = <0x09>;
}
```

Listato 5.3: Compatibilità di LTC3350 su device tree

Trovato un driver compatibile, il nuovo device I2C è istanziato, ed è chiamata la sua funzione `probe()`, che sarà descritta nel capitolo Implementazione.

# Capitolo 6

# Analisi e progettazione

Questo progetto consiste nell'implementazione di una feature all'interno di un sistema più grande, il kernel di Linux, e di un suo esempio di utilizzo.

L'azienda che ha commissionato questo progetto, Eurotech, ha fatto richieste specifiche sulle funzionalità necessarie. Lo sviluppatore potrebbe essere tentato di includere nel driver solamente le capacità che sono state richieste esplicitamente, ma ho scelto di rendere il driver abbastanza versatile da poter essere utilizzato in qualsiasi tipo di progetto.

Di conseguenza, mi sono assicurata che il driver soddisfacesse le richieste del cliente, ma anche eventuali necessità di chi si trovasse ad utilizzare LTC3350 in un altro sistema, o in versioni successive del Dynagate. Quindi ho scritto due gruppi di requisiti: quelli di Eurotech, che ha commissionato il prodotto, e quelli dell'utilizzatore di driver generico, cioè chiunque voglia integrare un LTC3350 nel suo sistema Linux.

## 6.1 Requisiti di Eurotech

Secondo le richieste del cliente, era necessario costruire un driver che permettesse di:

- Usare LTC3350 per il monitoraggio della salute dei supercondensatori.
- Controllare che il voltaggio del capacitore non superi una certa soglia, ma solo se la temperatura è minore di 65 gradi.
- Monitorare la temperatura dell'interno del chip, e mandare un allarme nel caso superi i 110 gradi.
- Effettuare misure periodiche di ESR e capacitanza dello stack dei supercondensatori, mandando un allarme qualora si superassero certi limiti.

Per via della specificità, implementare un sistema che svolga solo queste operazioni causerebbe due problemi: in primo luogo, se le richieste del cliente cambiassero

nella prossima versione del prodotto, sarebbe necessario andare a modificare il driver per aggiungere anche solo una misura. In secondo luogo, il driver sarebbe utile solamente per il Dynagate, e non potrebbe essere riutilizzato in altri sistemi.

Per questo motivo ho fatto la scelta, supportata dal mio tutor, di ampliare la lista dei requisiti in modo da soddisfare le richieste del cliente ma anche di scrivere un modulo che possa essere riutilizzato in sistemi diversi, e da sviluppatori diversi, per obbiettivi diversi. L'obbiettivo è stato di permettere allo sviluppatore di usare tutte le funzionalità di LTC3350, senza limitare il suo utilizzo, ma anche di offrire un'applicazione pronta per eseguita.

## 6.2 Requisiti

Elenco i requisiti funzionali del prodotto completo, includendo driver e applicazione. Il prodotto deve:

1. Istanziare il dispositivo all'interno del Device Driver Model.
2. Leggere dal device tree le proprietà del dispositivo.
3. Configurare il device con le proprietà lette, che includono `Capacitor Overvoltage Level`, `Maximum Die Temperature`, `ESR High Level`, `ESR Measurement Period`, `CAP-ESR Initial Measurements`.
4. Configurare il device per eseguire delle misure periodiche di capacitanza e ESR.
5. Scrivere e leggere sui registri di LTC3350 tramite bus $I^2C$.
6. Esporre i registri come attributi sysfs, permettendo letture e scritture dall'utente.
7. Rispondere agli SMBUS alert del dispositivo.
8. Notificare il ricevimento degli alerts.
9. Permettere l'implementazione di un programma in user space che sfrutti le funzionalità sopra.
10. Leggere e scrivere sui registri come su files regolari.
11. Attendere il ricevimento di alerts.
12. Azzerare gli allarmi.
13. Ottenere uno status report completo di LTC3350.
14. Convertire valori dalle unità di misura standard alle unità dei registri.

D'ora in poi mi riferirò all'applicazione principale in C come `ltc-monitor` e allo script di testing in python come `ltcsensors`. Il driver è chiamato semplicemente `ltc3350` nel kernel, e mi riferirò al suo file sorgente come `ltc3350.c`, e ad esso semplicemente come "il driver" o `ltc3350-driver`.

La creazione di un'applicazione è stata necessaria sia per mantenere una separazione tra meccanismo e politica, come visto in 2.4.1, il ruolo del driver, ma anche per testare il driver. Da un lato è un'applicazione funzionante e pronta all'uso, dall'altro è un template per lo sviluppo di un'applicazione personalizzata per un sistema specifico.

## 6.3 Progettazione

Le decisioni più importanti che ho svolto al livello architetturale, sono state quelle di suddividere le responsabilità e le funzionalità tra il driver e l'applicazione, nell'ottica degli usi diversi che ne farà il prossimo sviluppatore nella catena. Nello specifico, il driver è pensato per funzionare su qualsiasi sistema, e non è previsto che sia necessario alterarlo. L'applicazione, invece, pur essendo funzionante, è pensata per essere modificata o presa come esempio per sviluppare applicazioni adatte alla necessità del prodotto specifico: il driver è pensato per qualsiasi calcolatore che includa LTC3350, ma l'applicazione è stata pensata e testata solo per il Dynagate.

Una decisione importante che ho svolto è stata quella di delegare le conversioni dei valori all'applicazione, piuttosto che effettuarle in driver. In generale, i driver dei circuiti per l'hardware monitoring tendono ad effettuare automaticamente le conversioni in unità di misura del sistema internazionale. In questo caso, però, le conversioni non dipendevano solo da costanti, ma anche dalla specifica configurazione dell'LTC3350, quindi da quali resistenze gli sono state applicate nell'hardware.

Un'altra motivazione per questa scelta è stata la precisione delle misure. L'arrotondamento legato alla conversione ad unità di misura può comportare una perdita di informazione, soprattutto siccome il kernel di Linux offre supporto limitato per l'utilizzo di floating point numbers.

La conversione viene svolta nell'applicazione, per cui l'utente può scegliere se leggere i valori ricevuti da LTC3350 o i valori convertiti. Chi sviluppasse un'applicazione con lo stesso driver può copiare e incollare le mie funzioni, eventualmente sostituendo le costanti con quelle appropriate per il suo dispositivo hardware.

### 6.3.1 Esempio di caso d'uso

Di seguito un esempio di caso d'uso interessante perchè evidenza il rapporto tra le diverse componenti del sistema.

**Nome:** Alert

**Descrizione:** LTC3350 segnala di aver registrato una condizione di allarme all'utente.

**Attore primario:** LTC3350.

**Attore secondario:** Utente.

**Precondizioni:** Driver inizializzato.

**Sequenza degli eventi principali:** LTC3350 rileva la condizione di allarme e la segnala tramite `#SMBALERT`. Il `smbalert-driver` rileva e gestisce l'interruzione. Chiama la funzione di alert di `ltc3350-driver`, che notifica `ltc-monitor`. Questo stampa il messaggio di alert.

**Postcondizioni:** stampa su terminale.

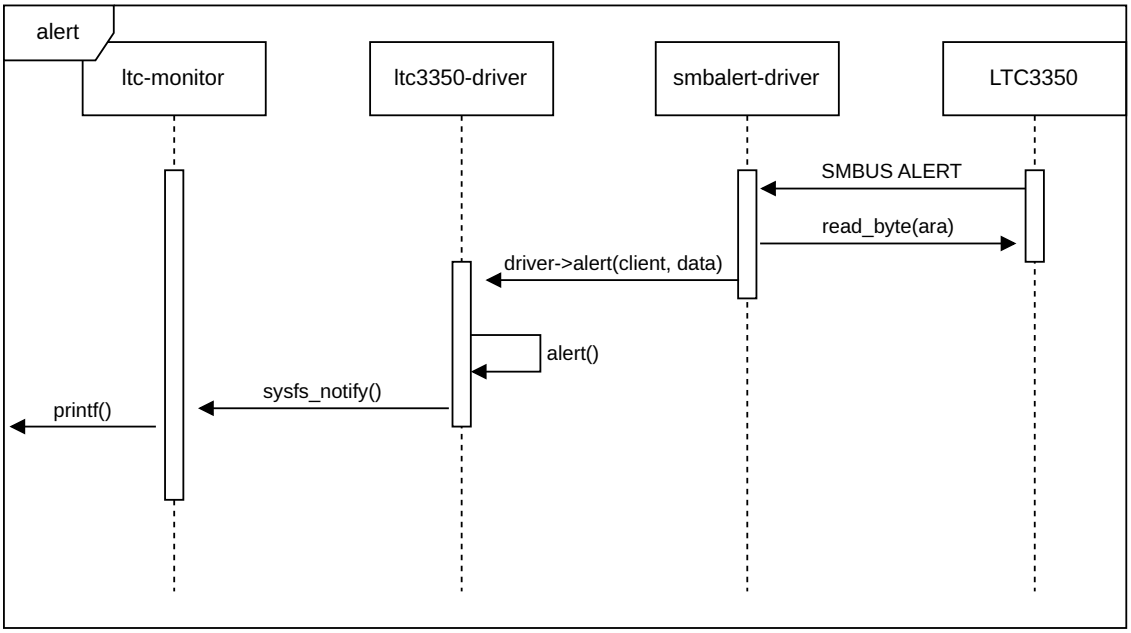



Figura 6.1: Diagramma di sequenza per alert

### 6.3.2 Lo standard hwmon e perchè non l'ho seguito

Altre decisioni importanti sono state legate al rapporto con il Kernel di Linux e con lo standard Hwmon(hardware monitoring). Cercando degli strumenti per costruire il driver, la libreria hwmon mi è sembrata la più appropriata. Per pubblicare un driver nel kernel nella categoria `hwmon`, è necessario seguire una serie di linee guida, che contengono i nomi degli attributi e alcune caratteristiche della struttura. Per quanto le funzioni della libreria fossero utili per il lavoro, LTC3350 non è soltanto un sensore di voltaggi e correnti, ma permette un monitoraggio più ampio, permette di

agire sui supercondensatori, ed ha un numero di registri di molto maggiore rispetto, ad esempio, ad un sensore di temperatura. Di conseguenza, ho scelto un approccio diverso da quello consigliato, nominando gli attributi come i registri, in modo da rendere il driver più facile da utilizzare.

Questo ha impedito la compatibilità con la libreria "lm-sensors". Ho deciso di scrivere uno script che offrisse le stesse funzionalità, utilizzando le misure registrate dal driver: da qui viene **ltcsensors**, che permette di visualizzare le misure in tempo reale e scriverle su file per analizzarle.

### 6.3.3 Proprietà

Ho cercato di attenermi ai seguenti principi di progettazione per garantire la qualità del progetto.

**Incapsulamento** L'incapsulamento è una caratteristica dei linguaggi di programmazione ad oggetti, cioè la proprietà di mantenere al loro interno sia attributi che metodi. Un driver, tuttavia, gode della stessa proprietà: ha uno stato interno e un comportamento. Ad esempio, la funzione di alert e la funzione di probing possono essere viste come metodi, in quanto ad esso specifiche e contenute nella sua struttura dati. I driver fanno uso di funzioni e variabili statiche per un sottile livello di information hiding.

```
static struct i2c_driver ltc3350_driver = {
  .driver = {
    .name = "ltc3350",
    .of_match_table = of_match_ptr(ltc3350_match),
  },
  .probe_new  = ltc3350_probe,
  .id_table = ltc3350_id,
  .alert = ltc3350_alert,
};
```

Listato 6.1: Struttura dati driver

Più importante è il fatto che offre un'interfaccia alle altre componenti software che ci interagiscono, siano esse del sistema operativo o applicazioni al livello utente: il suo funzionamento interno e la sua implementazione non sono necessarie per il suo utilizzo. L'interfaccia offerta al resto del kernel è quella del driver che gestisce un determinato dispositivo all'interno del Device and Driver Model. L'interfaccia offerta al livello utente è quella di un insieme di attributi sysfs, trattati come file, su cui è possibile svolgere operazioni. In entrambi i casi, i moduli del sistema possono essere modificati indipendentemente.

**Coesione funzionale** Tutte le funzioni del driver contribuiscono ad un obbiettivo finale: dare al livello utente le funzionalità del dispositivo in modo trasparente. Il

driver cerca di essere il più possibile universale: la sua implementazione dipende solo da LTC3350. Tutto ciò che può dipendere dal sistema in cui è integrato è lasciato alle applicazioni utente.

**Efficienza delle prestazioni** Per quanto riguarda l'occupazione di spazio, il codice sorgente del driver è grande 22KB, e l'intera patch, inclusa la documentazione, misura 44KB. Il suo file oggetto è grande circa 500 KB. L'intera immagine compilata del kernel è grande 32 MB. Le strutture dati allocate per l'inizializzazione e la gestione del driver sono nell'ordine di grandezza delle decine di KB. L'efficienza in tempo sia nel driver che in ltc-monitor si trova nel loro uso dell'attesa passiva: questo sarà approfondito nella sezione sull'implementazione.

**Tolleranza agli errori** Il software effettua una gestione delle eccezioni continua e consistente che assicura il funzionamento del sistema generale anche in caso di errori hardware e dell'utente. Si vedranno esempi nel capitolo 8, Validazione e verifica.

# Capitolo 7

# Implementazione

Lo sviluppo del driver e della sua applicazione è stato iterativo ed incrementale: le funzionalità sono state implementate e testate una alla volta. Ad ogni funzionalità del driver è corrisposto il suo utilizzo all'interno di `ltc-monitor`, e il testing è stato incessante. Questo capitolo contiene un'esposizione dei metodi di implementazione delle funzionalità del prodotto.

## 7.1 Inizializzazione driver

### 7.1.1 Inizializzazione e probing

La sezione 5.4, Driver I$^2$C e SMBus, parla dell'inizializzazione del driver come avviene nel prodotto finito nel Dynagate: cioè attraverso la lettura del device tree. Tuttavia, non è l'unico modo in cui può avvenire il probing del driver, ma esiste un metodo manuale tramite `modprobe`.

```
1 modprobe
2 echo ltc3350 0x07 > /sys/bus/i2c/devices/i2c-2/new_device
```

`modprobe` è un comando utilizzato nei sistemi operativi Linux per caricare e scaricare i moduli del kernel in modo dinamico. I comandi mostrati sopra causano il caricamento del modulo, e creano un nuovo dispositivo I$^2$C associato al driver `ltc3350` sull'adapter I$^2$C numero due, con l'indirizzo `0x09`.

Per trovare il driver `ltc3350`, il kernel fa uso di una struttura dati chiamata `id_table`, diversa dalla `of_match_table` utilizzata per la registrazione dei dispositivi da Device Tree.

```
1 static struct i2c_driver ltc3350_driver = {
2   .driver = {
3     .name = "ltc3350",
4     .of_match_table = of_match_ptr(ltc3350_match),
```

```
5   },
6   .probe   = ltc3350_probe,
7   .id_table = ltc3350_id,
8   .alert = ltc3350_alert,
9 };
10 module_i2c_driver(ltc3350_driver);
```

Listato 7.1: Struttura dati che rappresenta il driver

Il listato 7.1 mostra i punti di accesso al driver:

- Il nome del driver, "ltc3350".
- `of_match_table`, la tabella di compatibilità nel device tree.
- `probe`, la funzione che viene chiamata per l'associazione del driver al dispositivo.
- `id_table`, tabella di compatibilità per registrazione esplicita a runtime.
- `alert`, la funzione chiamata al ricevimento di un SMBus Alert dal dispositivo.

Entrambi i metodi di registrazione del device causano la generazione di una nuova directory sysfs, e l'invocazione della funzione di probing. Il device creato ha esso stesso un kobject associato, che può avere degli attributi.

### 7.1.2 Letture e scritture

Come i kobject, l'attributo non è pensato per essere utilizzato direttamente: contiene solo le proprietà name, owner e mode. Non dà nessun modo di leggere o scrivere il suo valore. E' compito dei singoli sottosistemi definire le strutture dei loro attributi e le funzioni per gestirli. Il driver model descrive `struct device_attribute` come:

```
1 struct device_attribute {
2   struct attribute attr;
3   ssize_t (*show)(struct device *dev, struct
     device_attribute *attr, char *buf);
4   ssize_t (*store)(struct device *dev, struct
     device_attribute *attr,
5                    const char *buf, size_t count);
6 };
```

Listato 7.2: Struttura dati device_attribute

Quindi aggiunge all'attributo due metodi, lo `show` per leggere il valore, e lo `store` per scrivere. Ho usato dei `device_attribute` per rappresentare i registri di LTC3350. Nel driver, gli attributi sono definiti tramite delle macro. Il device è

registrato come parte del device model all'interno della funzione di probing, insieme ai suoi attributi.

Ogni volta che l'utente legge un file corrispondente ad un attributo, viene chiamata la funzione di `show()` che ho definito.

```
static ssize_t ltc3350_value_show(struct device *dev,
                 struct device_attribute *da, char *buf)
{
  struct sensor_device_attribute *attr =
   to_sensor_dev_attr(da);
  struct ltc3350_hwmon_data *data = dev_get_drvdata(dev);
  int value;
  int ret;

  if (IS_ERR(data))
    return PTR_ERR(data);

  ret = ltc3350_get_value(data->client, attr->index, &
   value);
  if (unlikely(ret < 0)) {
    return ret;
  }

  return sysfs_emit(buf, "%d\n", value);
}
```

Listato 7.3: Funzione show()

Nella funzione 7.3, il driver chiama la funzione `ltc3350_get_value()`, la quale a sua volta chiama `i2c_smbus_word_data()`, che effettua le operazioni necessarie comando SMBus read word (3.2.5).

Questo è interessante perchè significa che:

1. E' possibile svolgere qualsiasi operazione all'interno di una funzione show/store associata ad un attributo.

2. Ho la scelta completa di quali attributi implementare: non è necessario che corrispondano ai registri.

3. Sysfs permette di svolgere qualsiasi operazione tramite lettura o scrittura da file.

4. Le operazioni di lettura e scrittura da bus vengono svolte solamente quando avviene una lettura del file, che rappresenta l'attributo sysfs, che corrisponde ad una lettura da bus $I^2C$.

Molti dispositivi che utilizzano lo standard Hwmon scelgono di far corrispondere un attributo a ciascun allarme. In questo caso, ho scelto di evitare, siccome LTC3350

permette di attivare 16 allarmi diversi. E' comunque facile impostarli scrivendo sul file corrispondente al registro di selezione allarmi. Ho deciso di implementare un attributo per ogni registro, perchè rende semplice scrivere una applicazione che utilizza le funzionalità di LTC3350.

```
root@dynagate-10-14-utiliboot:SD:/sys/bus/i2c/devices/i2c-2/2-0009/
    hwmon/hwmon4# ls
alarm_reg     ctl_reg          ichg_uc_lvl   meas_iin     meas_vout
power         vin_uv_lvl       cap_esr_per   device       iin_oc_lvl
meas_vcap     mon_status       subsystem     vout_ov_lvl  cap_lo_lvl
dtemp_cold_lvl meas_cap        meas_vcap1    msk_alarms   uevent
vout_uv_lvl   cap_ov_lvl       dtemp_hot_lvl meas_dtemp   meas_vcap2
msk_mon_status vcap_ov_lvl     vshunt        cap_uv_lvl   esr_hi_lvl
meas_esr      meas_vcap3       name          vcap_uv_lvl  chrg_status
gpi_ov_lvl    meas_gpi         meas_vcap4    num_caps     vcapfb_dac
clr_alarms    gpi_uv_lvl       meas_ichg     meas_vin     of_node
vin_ov_lvl
```

Listato 7.4: Directory del dispositivo con attributi corrispondenti

In 7.4 è possibile osservare che esiste una directory per il device hwmon, che contiene un attributo per ogni registro. Ad esso si aggiungono degli attributi che descrivono il nome del device, il suo sottosistema, un link simbolico alla directory del device, e altri due file di gestione, `power` e `uevent`. All'interno della funzione `show()`, è anche possibile effettuare una conversione dei valori.

Per una fase del progetto, ho scritto le funzioni in modo da convertire i valori automaticamente, ma poi ho spostato la funzionalità nell'applicazione. Una volta che si implementa un cambiamento nella funzione di show, non si può "tornare indietro", l'informazione del vero valore letto dal bus è persa. La funzione `sysfs_emit()` stampa il numero letto come stringa ASCII sul file dell'attributo corrispondente.

La funzione `store()` è analoga alla `show()`, ma utilizza il comando write word.

**Dal lato utente** L'applicazione `ltc-monitor` legge gli attributi sysfs come se fossero file regolari, usando funzioni quali `readdir()` ed `fopen()`, in modo completamente trasparente al funzionamento interno. Il comando `ltc-monitor show` effettua la seguente stampa.

```
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor show
cap_ov_lvl      : 12534   2300 mV     ichg_uc_lvl      : 0       0
gpi_ov_lvl      : 0       0 mV        msk_alarms       : -8190   -8190
meas_iin        : 1541    1541        vin_uv_lvl       : 0       0 mV
num_caps        : 1       1           cap_uv_lvl       : 0       0 mV
vcapfb_dac      : 15      15          gpi_uv_lvl       : 0       0 mV
iin_oc_lvl      : 0       0           dtemp_hot_lvl    : 12907   110 C
meas_vcap4      : -7      -1 mV       clr_alarms       : 0       0
ctl_reg         : 0       0           vout_ov_lvl      : 0       0 mV
meas_vcap2      : 11961   2194 mV     vshunt           : 14745   2705 mV
meas_cap        : 66      15 F        mon_status       : 513     513
meas_vcap       : 3007    4438 mV     vout_uv_lvl      : 0       0 mV
meas_ichg       : 23      23          meas_vout        : 2221    4908 mV
chrg_status     : 5409    5409        cap_lo_lvl       : 166     40 F
vcap_ov_lvl     : 0       0 mV        meas_esr         : 441     34 mR
```

```
meas_gpi        : -15     -2 mV      meas_dtemp      : 10542   43 C
msk_mon_status  : 16      16         meas_vcap3      : 0       0 mV
cap_esr_per     : 6       6          dtemp_cold_lvl  : 0       -251 C
vcap_uv_lvl     : 0       0 mV       meas_vcap1      : 12110   2222 mV
vin_ov_lvl      : 0       0 mV       esr_hi_lvl      : 512     40 mR
alarm_reg       : -32768  -32768     meas_vin        : 2242    4954 mV
```

Listato 7.5: stampa lista degli attributi

A fianco dei valori letti dal bus, si può vedere la conversione dei valori corrispondenti a misure di unità fisiche. Ad esempio, `meas_dtemp` mostra che il chip ha la tiepida temperatura di 43 gradi.

**Strutture dati** La funzione di probing utilizza la Hwmon API per la registrazione del device, e inizializza due strutture dati. La prima contiene i dati legati al dispositivo hwmon, che saranno accessibili nelle funzioni di lettura e scrittura. La seconda contiene i dati legati all'oggetto del client $I^2C$, e sarà disponibile nella funzione di alert. E' necessario definire queste due strutture dati, perchè al device corrisponde un kobject che deve essere disponibile nella funzione di alert, ed è necessario accedere al client $I^2C$ per svolgere le operazioni di lettura su bus.

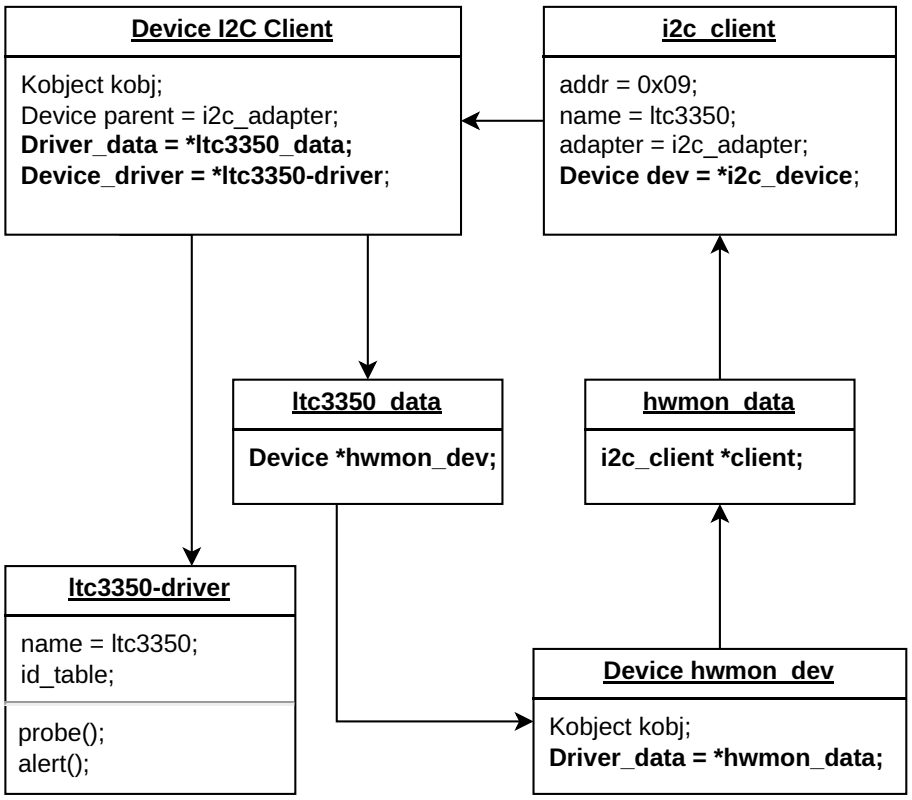



Figura 7.1: Rapporti tra oggetti

Nella figura 7.1, ho mostrato le relazioni tra le strutture dati coinvolte nella descrizione dei diversi device, indicando con le frecce i puntatori.

### 7.1.3 Lettura da device tree

In seguito alla registrazione del device, la funzione di probing effettua anche una configurazione da device tree: legge i valori di alcune proprietà per specificare le impostazioni di partenza del dispositivo.

```
ltc3350: ltc3350@09 {
  compatible = "ltc3350";
  reg = <0x09>;

  capacitor-overvoltage-level = <12534>; // 2300 mV
  maximum-temperature = <12907>; // 110 C
  esr-high-level = <512>; // 40 mR
  capacitance-low-level = <166>; // 40 F
  cap-esr-measurement-period = <17280>; // 2 days
  cap-esr-initial-measurements = <10>;
};
```

Listato 7.6: rappresentazione nel device tree di LTC3350

Ho dovuto scegliere quali caratteristiche rappresentare come proprietà del device tree: la scelta è ricaduta su quali sarebbero state più utili per il committente. Ho deciso di aggiungere l'opzione del numero di misurazioni di capacitanza ed ESR da svolgere alla prima attivazione del dispositivo.

Queste misurazioni diventano più accurate, più volte vengono effettuate, e possono non essere affidabili fino a 72 ore dall'accensione. Per questo motivo, ho considerato che aumentare la frequenza delle misurazioni per il primo periodo fosse ottimale.

## 7.2 Alert e interrupt handling

Il meccanismo di alert è stato il più complesso da implementare, ma anche il più utile. L'applicazione che vediamo usa il meccanismo solo per effettuare stampe da terminale, ma un meccanismo simile può permettere di suonare un allarme o spegnere il sistema se si verificano condizioni di pericolo. Abbiamo visto il percorso che porta a chiamare la funzione `alert` nelle sezioni 2.5, 3.2.6, 3.3.3, 4.3 e 6.3.1.

Lo scopo della funzione di alert è di notificare l'utente all'attivarsi di un allarme. Trovare un metodo e capire come funzionasse è stato un grosso ostacolo nello sviluppo del driver. C'erano diversi metodi per ottenere questo risultato. Il primo, ovvio ma estremamente inefficiente, era di effettuare un controllo periodico sui file degli allarmi attivi. Scegliere un periodo di misurazione adeguato avrebbe dato risultati insoddisfacenti: da un lato un allarme può verificarsi una volta ogni diversi mesi, quindi renderebbe inutile controllare con troppa frequenza. Dall'altro, è necessario comunicare istantaneamente l'avvenimento di una interruzione di corrente.

Un suggerimento che ho trovato è stato quello di utilizzare Netlink Sockets, un meccanismo che permette alle operazioni in userspace di ricevere messaggi asincroni dal kernel. Crea un canale bidirezionale che permette lo scambio di messaggi tra i due livelli. Tuttavia, richiede il supporto di molte strutture dati, e offre funzionalità non necessarie. L'obbiettivo è di inviare un segnale che non porta alcuna informazione aggiuntiva al processo giusto, quindi non è necessario un vero scambio di messaggi.

Il meccanismo dei segnali sembra essere la soluzione perfetta: ma richiede di avere informazioni sul processo a cui inviare il segnale. Il driver è totalmente indipendente dall'applicazione che lo utilizza: non solo non è possibile conoscere il PID del processo, ma neanche le caratteristiche dell'eseguibile.

La funzione `sysfs_notify()` non è presente nella Linux Filesystem API[26], come `sysfs_emit()`, nè nella documentazione di sysfs[25], nè in alcuno dei libri da cui ho studiato il kernel di Linux. E' presente però nella funzione di alert del driver `STTS751`, sensore locale di temperatura digitale a basso voltaggio.[27]

```
1  static void ltc3350_alert(struct i2c_client *client, enum
       i2c_alert_protocol, unsigned int data){
2    int alarm_value;
3    int monitor_value;
4    int ret1, ret2;
5    struct kobject *kobj;
6    struct device *hwmon_dev;
7    struct ltc3350_data *client_data = dev_get_drvdata(&client->dev);
8    hwmon_dev = client_data->hwmon_dev;
9
10   dev_dbg(&client->dev, "Alert from device at address 0x%02x\n", client
       ->addr);
11   ret1 = ltc3350_get_value(client, ALARM_REG, &alarm_value);
12   ret2 = ltc3350_get_value(client, MON_STATUS, &monitor_value);
13
14   if (unlikely(ret1<0 || ret2 < 0))
15     dev_err(&client->dev, "Error reading value of ALARM_REG or
       MON_STATUS\n");
16
17   ltc3350_show_alarms(alarm_value, monitor_value, client);
18
19   if (hwmon_dev == NULL) {
20     dev_err(&client->dev, "Error getting hwmon_dev. Sysfs notifications
        disabled.\n");
21   } else {
22     kobj = &hwmon_dev->kobj;
23     sysfs_notify(kobj, NULL, "alarm_reg");
24     sysfs_notify(kobj, NULL, "mon_status");
25   }
26 }
```

Listato 7.7: funzione di alert

La funzione in 7.7 si limita a leggere i valori dei registri che indicano gli allarmi attivi, effettuare delle stampe su dmesg, e chiamare la funzione `sysfs_notify()`. La funzione prende come parametri un kobject, una stringa `dir`, e una stringa `attr`. Il

kobject è quello corrispondente al dispositivo hwmon, e ho scelto di notificare per entrambi gli attributi. Questa funzione non ha effetti visibili, ma segnala l'avvenimento di una condizione non precisata sul file corrispondente all'attributo indicato.

### 7.2.1 Applicazione

Dal lato utente, il processo rimane in attesa di avvenimenti sui file a cui è interessato, e attende tramite la funzione `poll()` sul file descriptor. L'evento che risveglia la funzione è `POLLPRI`, e indica "una qualche condizione eccezionale sul file descriptor".[28]

La `poll()` implementa un'attesa passiva molto efficiente e minimale, che permette una segnalazione immediata ed efficiente delle alerts.

```
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor await &
[1] 984
You will be notified in the event of an alarm, or a change in monitor
    status.
Polling for alerts...
[ 2189.131680] ltc3350 2-0009: ALARM_CAP_LO
New data ltc-monitor
MONITOR STATUS:
Waiting programmed time to begin a capacitance/ESR measurement.
Capacitance measurement has completed.
ESR Measurement has completed.
The device is connected to power outlet.
ALARMS:
stack capacitance low alarm
Measured capacitance value: 177. Capacitance Low Level: 166
CHARGER STATUS:
The synchronous controller is in step-down mode (charging)
The capacitor voltage is above power good threshold
The charger is in constant current mode
```

Listato 7.8: ltc-monitor si mette in attesa, e riceve un alert

## 7.3 Misure periodiche di capacitanza e ESR

Secondo Eurotech, le misure di capacitanza e ESR sono tra le più importanti(vedi 6.1), ma non sono svolte in automatico, e necessitano di essere programmate esplicitamente per essere svolte(vedi 3.3.4).

Inoltre, non sono totalmente accurate nelle prime 72 ore, e diventano più accurate man mano che vengono svolte. Per questo motivo, il driver svolge automaticamente una quantità di misure nelle prime ore di accensione del dispositivo, di default. Questa opzione può essere modificata nel device tree. La proprietà `cap-esr-initial-measurements` del device tree permette di specificare il numero di misure iniziali in seguito alla prima accensione del dispositivo.

La documentazione completa dei bindings di LTC3350 nel device tree è disponibile a [29], LTC3350 device tree bindings.

```
i2c {
    interrupts-extended = <&gic GIC_SPI 37
   IRQ_TYPE_LEVEL_HIGH>,
      <&gpio1 10 IRQ_TYPE_LEVEL_LOW>;
    interrupt-names = "irq", "smbus_alert";

    ltc3350: ltc3350@09 {
      compatible = "ltc3350";
      reg = <0x09>;

      capacitor-overvoltage-level = <12534>; // 2300 mV
      maximum-temperature = <12907>; // 110 C
      esr-high-level = <192>; // 15 mR
      capacitance-low-level = <249>; // 60 F
      cap-esr-measurement-period = <12>;
      cap-esr-initial-measurements = <10>;
    };
};
```

Listato 7.9: Device tree entry corrispondente a ltc3350

### 7.3.1 Utilizzo di ltc-monitor

```
Usage: ltc-monitor <command>
Accepted commands:
  show
  await
  write file value [measurement unit]
  read [-c] file
  clear
  status
```

Listato 7.10: Utilizzo di ltc-monitor

L'applicazione ltc-monitor permette di chiamare cinque comandi.

- **show** mostra il contenuto di tutti i registri.
- **await** mette il processo in attesa finché non arriva un alert, momento in cui mostra uno status report.
- **write** permette di scrivere un valore su un registro, eventualmente effettuando la conversione da una unità di misura.
- **read** permette di leggere il valore di un registro, eventualmente effettuando una conversione nell'unità di misura appropriata.

- **clear** resetta il valore del registro `alarm_reg`, quindi azzerando gli allarmi e permettendo loro di essere attivati nuovamente.
- **status** effettua uno status report completo del dispositivo, che include gli allarmi attivi, lo stato delle misure e dell'alimentazione.

## 7.4 Script in python

Ho scritto uno script in python che effettua la lettura periodica in tempo reale dei valori di alcuni registri interessanti e ne salva il valore nei file .csv, in modo da poterci fare un'analisi più approfondita. Anche questo script è un esempio di quanto sia facile interagire con i valori dei registri semplicemente grazie alla conoscenza del linguaggio di programmazione.

```
usage: ltcsensors [-h] [-s] [-l LENGTH] [-f FILE]

Offers readings of ltc3350 sensors

options:
  -h, --help            show this help message and exit
  -s, --silent
  -l LENGTH, --length LENGTH
  -f FILE, --file FILE
```

Listato 7.11: ltcsensors.py help message

Nel capitolo 8 su Validazione e Verifica si vedranno i risultati delle misure svolte grazie a questo script.

# Capitolo 8

# Validazione e verifica

L'obbiettivo di questa fase di validazione e verifica del software è stato di garantire che il driver funzioni come previsto, sia stabile e non comprometta le prestazioni del sistema.

## 8.1 Verifica statica e test funzionali

Sviluppando applicazioni al livello utente, siamo generalmente abituati a testare il loro funzionamento immediatamente: le modifiche che posso svolgere ad un file .css in un progetto di programmazione web sono visibili istantaneamente sul browser. La programmazione del kernel di Linux, invece, richiede diversi passaggi tra la scrittura del codice e i test. Nello sviluppo di questo driver, il primo livello di verifica statica è stata la compilazione: questa richiedeva circa un quarto d'ora, anche se era terminata prima quando venivano trovati errori nel driver. Anche così, la scrittura del codice è dovuta essere particolarmente attenta, e passavano diverse ore tra un test e l'altro. In seguito alla compilazione, l'immagine del sistema operativo doveva essere spostata sulla memoria del Dynagate, una scheda SD, e doveva essere svolto il riavvio del sistema.

Ho implementato le funzionalità una alla volta, e le ho testate una alla volta.

### 8.1.1 Lettura e scrittura

Per le operazioni di lettura e scrittura, oltre a verificare che avvenisse una scrittura, è stato necessario assicurarsi che i valori fossero corretti. Per questo è stata essenziale la conversione in unità di misura del sistema internazionale dei valori letti.

Durante questa fase ho individuato un difetto nella gestione dei numeri negativi. Infatti, in una misurazione, ho rilevato che il valore `meas_gpi`, cioè il voltaggio del pin GPI, era di 65522, corrispondente a circa 12 V. Non era plausibile che la misura fosse accurata perché nella documentazione il pin è indicato con un voltaggio compreso tra -0.3 V e 5.5 V. Sulla documentazione di LTC3350, è segnato che i

| Attributo | LSB | Valore | Attributo | LSB | Valore |
|---|---|---|---|---|---|
| cap_ov_lvl | 12534 | 2300 mV | ichg_uc_lvl | 0 | 0 |
| gpi_ov_lvl | 0 | 0 mV | msk_alarms | 0 | 0 |
| meas_iin | 1638 | 1638 | vin_uv_lvl | 0 | 0 mV |
| num_caps | 1 | 1 | cap_uv_lvl | 0 | 0 mV |
| vcapfb_dac | 15 | 15 | gpi_uv_lvl | 0 | 0 mV |
| iin_oc_lvl | 0 | 0 | dtemp_hot_lvl | 12907 | 110 C |
| meas_vcap4 | 1 | 0 mV | clr_alarms | 0 | 0 |
| ctl_reg | 0 | 0 | vout_ov_lvl | 0 | 0 mV |
| meas_vcap2 | 12377 | 2271 mV | vshunt | 14745 | 2705 mV |
| meas_cap | 182 | 43 F | mon_status | 536 | 536 |
| meas_vcap | 3076 | 4540 mV | vout_uv_lvl | 0 | 0 mV |
| meas_ichg | 13 | 13 | meas_vout | 2221 | 4908 mV |
| chrg_status | 5157 | 5157 | cap_lo_lvl | 249 | 60 F |
| meas_esr | 458 | 35 mR | meas_gpi | -15 | -2 mV |
| meas_dtemp | 10815 | 51 C | msk_mon_status | 0 | 0 |
| meas_vcap3 | 1 | 0 mV | cap_esr_per | 0 | 0 |
| dtemp_cold_lvl | 0 | -251 C | vcap_uv_lvl | 0 | 0 mV |
| meas_vcap1 | 12402 | 2275 mV | vin_ov_lvl | 0 | 0 mV |
| esr_hi_lvl | 192 | 15 mR | alarm_reg | 0 | 0 |
| meas_vin | 2241 | 4952 mV | | | |

Tabella 8.1: Valori degli attributi con conversione

valori dei registri sono numeri con segno in complemento a due, ma le funzioni del driver $I^2C$ li interpretano come numeri positivi senza segno, di conseguenza i numeri negativi erano rappresentati come numeri positivi molto grandi. Questo difetto è stato corretto nella funzione di lettura interna al driver.

Per testare l'applicazione, ho verificato che fosse in grado di gestire input corretti così come input errati dall'utente. Come si può vedere dal Listato 8.1, la gestione degli errori è valida e consistente.

```
// Utilizzo corretto
root@dynagate-10-14-utiliboot:SD:/# ltc-monitor write msk_alarms 32767
Wrote 32767 on msk_alarms

// L'attributo ciao non esiste
root@dynagate-10-14-utiliboot:SD:/# ltc-monitor write ciao 12
write_value Failed to open sysfs file: No such file or directory

// L'attributo "meas_vcap" e' read-only
root@dynagate-10-14-utiliboot:SD:/# ltc-monitor write meas_vcap 22
write_value Failed to open sysfs file: Permission denied

// Tentativo di leggere un attributo che non esiste
root@dynagate-10-14-utiliboot:SD:/# ltc-monitor read dajadjk
```

```
read_file Failed to open sysfs file: No such file or directory
read_integer_value : No such file or directory
-1
```

Listato 8.1: Test funzionalità di lettura e scrittura.

```
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor write cap_esr_per 12
Wrote 12 on cap_esr_per
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor write dtemp_cold_lvl
    -10 C
Wrote 8621 on dtemp_cold_lvl
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor read dtemp_cold_lvl
8621
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor read -c dtemp_cold_lvl
-10 C
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor read -c meas_vcap1
2226 mV
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor read meas_vcap1
12129
```

Listato 8.2: Utilizzo corretto di ltc-monitor per effettuare letture e scritture.

### 8.1.2 Ricezione di alerts

Per testare la ricezione di alerts è stato necessario provocare situazioni di alert, ad esempio mettendo livelli troppo bassi per i limiti, in modo che i valori normali provocassero un alert. Oppure ponendo alert su eventi descritti dal registro di stato, come l'inizio di una misurazione ESR e di capacitanza.

```
root@dynagate-10-14-utiliboot:SD:~# [  350.796802] ltc3350 2-0009:
    ALARM_CAP_LO
[  352.870364] ltc3350 2-0009: Initial capacitance and ESR measurements
     may be inaccurate. Measurement number 0.
[  352.880330] ltc3350 2-0009: ALARM_CAP_LO
```

Listato 8.3: Test ricezione alert.

Nel listato 8.3, è possibile vedere le stampe svolte dal driver alla ricezione di un alert. Questo non garantisce che sia stato inviato un segnale anche all'applicazione utente. Perché succeda, l'applicazione utente deve mettersi in attesa di ricevere segnali dal driver.

```
root@dynagate-10-14-utiliboot:SD:~# ltc-monitor await &
[1] 984
You will be notified in the event of an alarm, or a change in monitor
    status.
Polling for alerts...

[ 2189.121741] ltc3350 2-0009: Initial capacitance and ESR measurements
     may be inaccurate. Measurement number 2.
[ 2189.131680] ltc3350 2-0009: ALARM_CAP_LO
```

```
New data ltc-monitor
MONITOR STATUS:
Waiting programmed time to begin a capacitance/ESR measurement.
Capacitance measurement has completed.
ESR Measurement has completed.
The device is connected to power outlet.
ALARMS:
stack capacitance low alarm
Measured capacitance value: 177. Capacitance Low Level: 166
CHARGER STATUS:
The synchronous controller is in step-down mode (charging)
The capacitor voltage is above power good threshold
The charger is in constant current mode
```

Listato 8.4: Test ricezione alert

Come in 2.1, al comando, l'applicazione `ltc-monitor` si mette in attesa di ricevere un alert. Quando LTC3350 termina una misura di capacitanza ed ESR, rileva un valore di capacitanza troppo basso. Questo è previsto, perché le prime misure di capacitanza tendono a sottostimare il suo valore. In questo caso, tuttavia, l'allarme di bassa capacitanza è stato causato da una misura passata: l'alert è causato dalla terminazione della misura, avvenimento osservato. Gli allarmi che non sono azzerati restano attivi.

Altre informazioni interessanti sono lo stato delle misurazioni periodiche, l'informazione che il device è collegato alla corrente e i supercapacitori sono in carica. L'applicazione mostra un confronto tra la misura corrente del valore che ha generato l'allarme, e il livello critico. In questo esempio, la capacitanza è sopra il livello minimo, quindi l'allarme può essere azzerato.

## 8.2 Test con ltcsensors

Il driver permette di ottenere informazioni in tempo reale sui voltaggi e la carica del sistema di alimentazione di backup. Per sfruttare queste funzionalità, ho scritto uno script in python che permette all'utente di vedere i valori dei voltaggi e della temperatura, chiamato `ltcsensors`. Questi appaiono sullo schermo e sono aggiornati ogni secondo per un intervallo di tempo passato come parametro.

```
root@dynagate-10-14-utiliboot:SD:~# ./ltcsensors.py -l 60
dtemp:    44.0  C    (max = 110.0  C , min = -251.4  C )
gpi:      -3 mV    (max =     0 mV, min =     0 mV)
iin:    3885 mV    (max =     0 mV, min = )
vcap:   4427 mV    (max =     0 mV, min =     0 mV)
vcap1:    2215 mV    (max =  2300 mV, min =     0 mV)
vcap2:    2188 mV    (max =  2300 mV, min =     0 mV)
vcap3:      -0 mV    (max =  2300 mV, min =     0 mV)
vcap4:      -1 mV    (max =  2300 mV, min =     0 mV)
vin:    4955 mV    (max =     0 mV, min =     0 mV)
vout:   4911 mV    (max =     0 mV, min =     0 mV)
```

Listato 8.5: Output ltcsensors.py

Il format è ispirato a lm-sensors, la libreria per il monitoraggio energetico dei dispositivi di hardware monitoring. Lo script salva i risultati in un file .csv. Ho avuto la possibilità di eseguire test del funzionamento su due dynagate diversi, in tre casi: utilizzo normale, utilizzo con misure periodiche di capacitanza ed ESR, e momento di sospensione di corrente.

### 8.2.1 Funzionamento normale

Durante il funzionamento normale, senza misure di capacitanza e ESR, i Dynagate hanno uno stato simile al seguente:

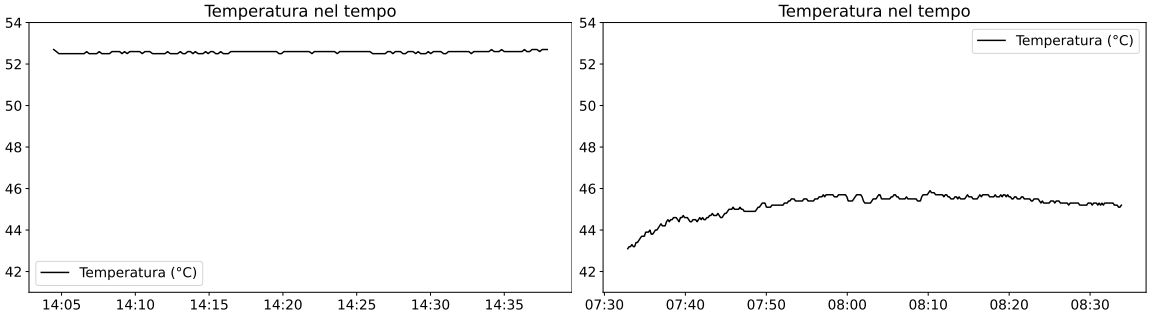
```
SUPERCAPACITORS STATUS REPORT
-----------------------------

MONITOR STATUS:
Capacitance measurement has completed.
ESR Measurement has completed.
The device is connected to power outlet.
ALARMS:
CHARGER STATUS:
The synchronous controller is in step-down mode (charging)
The charger is in constant voltage mode
The capacitor voltage is above power good threshold
REGISTERS VALUES:
cap_ov_lvl          : 12534  2300 mV  ichg_uc_lvl         : 0      0
gpi_ov_lvl          : 0      0 mV     msk_alarms          : 0      0
meas_iin            : 1638   1638     vin_uv_lvl          : 0      0 mV
num_caps            : 1      1        cap_uv_lvl          : 0      0 mV
vcapfb_dac          : 15     15       gpi_uv_lvl          : 0      0 mV
iin_oc_lvl          : 0      0        dtemp_hot_lvl       : 12907  110
    C
meas_vcap4          : 1      0 mV     clr_alarms          : 0      0
ctl_reg             : 0      0        vout_ov_lvl         : 0      0 mV
meas_vcap2          : 12377  2271 mV  vshunt              : 14745  2705
     mV
meas_cap            : 182    43 F     mon_status          : 536    536
meas_vcap           : 3076   4540 mV  vout_uv_lvl         : 0      0 mV
meas_ichg           : 13     13       meas_vout           : 2221   4908
     mV
chrg_status         : 5157   5157     cap_lo_lvl          : 249    60 F
meas_esr            : 458    35 mR    meas_gpi            : -15    -2
    mV
meas_dtemp          : 10815  51 C     msk_mon_status      : 0      0
meas_vcap3          : 1      0 mV     cap_esr_per         : 0      0
dtemp_cold_lvl      : 0      -251 C   vcap_uv_lvl         : 0      0 mV
meas_vcap1          : 12402  2275 mV  vin_ov_lvl          : 0      0 mV
esr_hi_lvl          : 192    15 mR    alarm_reg           : 0      0
meas_vin            : 2241   4952 mV
```

Listato 8.6: Dynagate 1 durante l'operazione normale

I due dynagate su cui ho fatto test avevano tre differenze interessanti: la prima è che il Dynagate 1 è più giovane del Dynagate 2, quindi potrebbe avere un livello di degrado della batteria inferiore. La seconda è che Dynagate 2 era stato spento per anni prima di eseguire questi test, quindi è possibile che ci siano alcune differenze

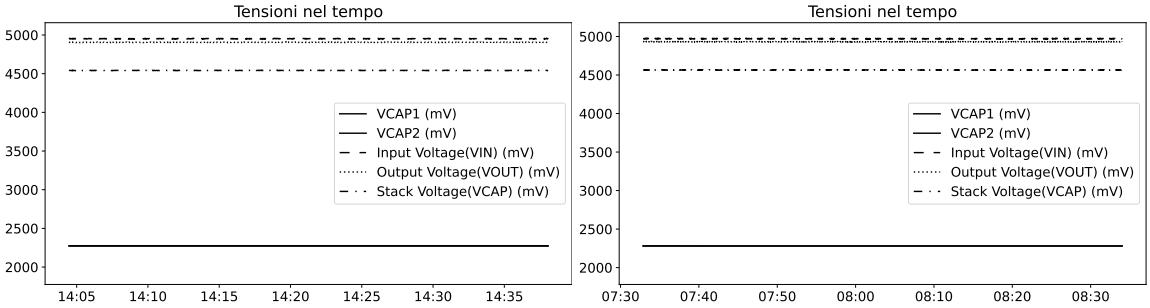
nelle misurazioni di capacitanza e ESR. La terza è il fatto che Dynagate 2 non era inserito in una custodia, per cui ha una dissipazione diversa della temperatura.

L'output 8.6 è stato preso dal Dynagate 1, che tende ad avere una temperatura più alta(51 °C). In 8.1 vediamo i grafici ottenuti dalle misure di temperatura dei due Dynagate per un'ora. Quando ho svolto queste misure, il primo dynagate era acceso da diverse ore, mentre il secondo era appena stato acceso. Per questo motivo, possiamo vedere che la temperatura del primo si aggira intorno ai 52 °C, mentre quella del secondo cresce, fino ad avvicinarsi ai 46 °C.




(a) Temperatura Dynagate 1 (b) Temperatura Dynagate 2

Figura 8.1: Temperature a confronto




(a) Tensioni Dynagate 1 (b) Tensioni Dynagate 2

Figura 8.2: Tensioni a confronto

Invece, le differenze nelle misure delle tensioni elettriche sono trascurabili, il funzionamento è uguale. Possiamo fare alcune osservazioni interessanti:

1. La tensione del pin del condensatore 1, indicata con VCAP1, è uguale alla tensione del pin del condensatore 2, indicata con VCAP2. Per questo motivo, le linee si accavallano. Il loro valore si aggira intorno a 2300 mV.

2. La misura di tensione dello stack dei condensatori(VCAP) ha un valore vicino alla somma della tensione dei due condensatori.

3. La tensione in ingresso del circuito è di poco superiore alla tensione di uscita.

4. I valori sono abbastanza costanti da non permettere di vedere le loro oscillazioni su questo grafico.

### 8.2.2 Misurazioni di capacitanza e ESR

Come visto nella sezione 3.3.4, Misura di capacitanza e ESR, questa misurazione è opzionale, e può essere programmata per essere eseguita periodicamente. Per eseguirla, il dispositivo attende che la carica sia completa, poi sospende il caricamento, e fa scaricare lo stack di supercondensatori di 200 mV. Questo può essere osservato con gli strumenti visti fin'ora.

```
  SUPERCAPACITORS STATUS REPORT
-----------------------------
MONITOR STATUS:
Capacitance/ESR measurement is in progress.
The device is connected to power outlet.
ALARMS:
CHARGER STATUS:
The synchronous controller is in step-down mode (charging)
The capacitor voltage is above power good threshold
The charger is temporarily disabled for capacitance measurement
```

Listato 8.7: Status report durante la misurazione di capacitanza e ESR

Nell'esempio 8.7 possiamo osservare lo status report durante una misurazione. Le misure dei registri sono state omesse perché poco interessanti. Osservando il comportamento dei due Dynagate per un'ora, possiamo vedere i risultati delle misure sui valori delle tensioni.

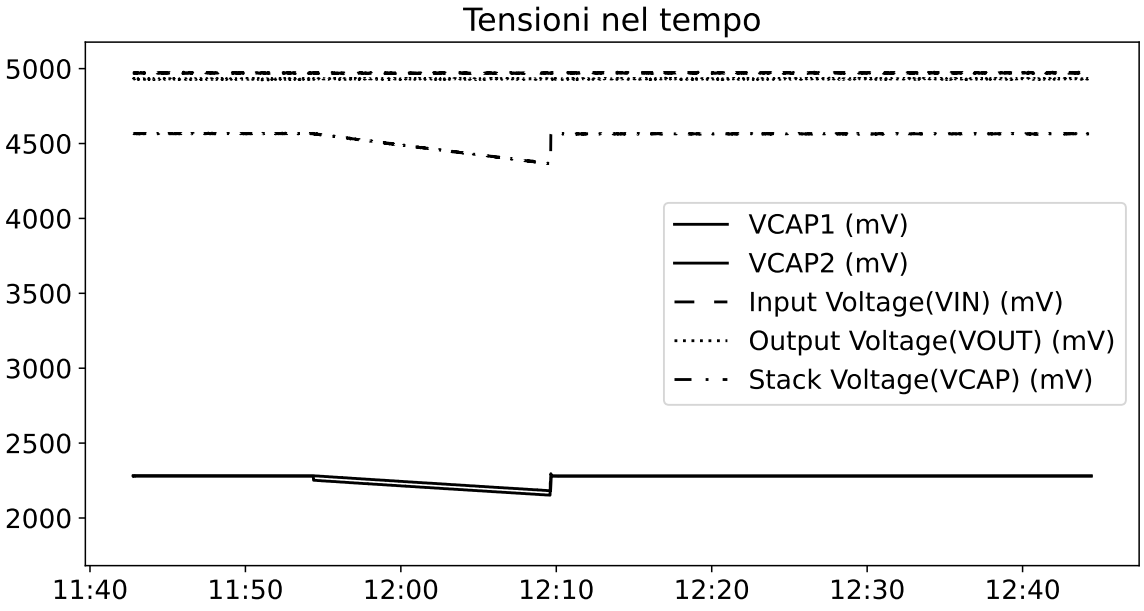



Figura 8.3: Misure periodiche di Capacitanza e ESR in Dynagate 2

Nell'esempio 8.3, il Dynagate 2 esegue la misurazione solo una volta: segue la velocità programmata di una misura all'ora prevista per le prime misure dopo l'accensione. La misurazione ha richiesto un quarto d'ora, in questo caso. Possiamo vedere che la tensione dello stack dei condensatori scende di circa 200 mV. La tensione dei due condensatori non scende alla stessa velocità.

L'esempio del Dynagate 1 prevede misure ogni minuto. Mostra anche due avvenimenti: i momenti in cui ho staccato la spina per eseguire test sulla mancanza di corrente. Si può già osservare che per quanto la tensione di input cali precipitosamente, la tensione di output resta pressoché costante. Nella prossima sezione vediamo la situazione più nel dettaglio.

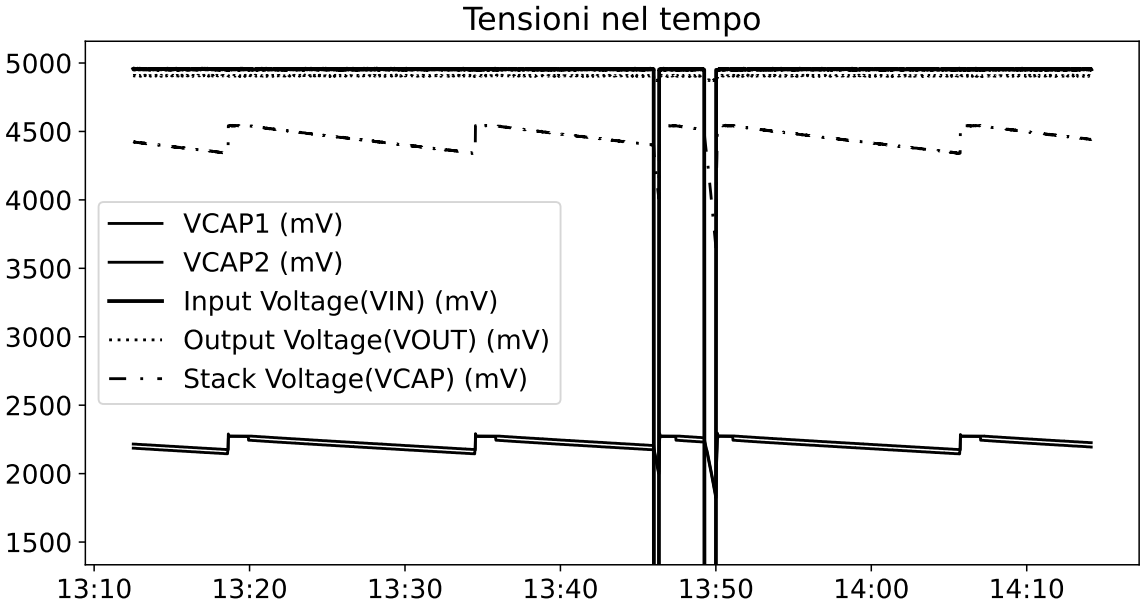



Figura 8.4: Misure periodiche di Capacitanza e ESR in Dynagate 1

### 8.2.3 Interruzioni di corrente

Lo scopo principale di LTC3350 e dello stack di supercondensatori è di provvedere alimentazione di emergenza. Ho testato il suo funzionamento tramite le due applicazioni.

```
SUPERCAPACITORS STATUS REPORT
-----------------------------

MONITOR STATUS:
Waiting programmed time to begin a capacitance/ESR measurement.
The last attempted capacitance measurement was unable to complete.
The last attempted ESR measurement was unable to complete.
The device is no longer connected to power outlet.
ALARMS:
CHARGER STATUS:
The synchronous controller is in step-up mode (backup)
```

```
The charger is in undervoltage lockout
The capacitor voltage is above power good threshold
The charger is in constant current mode
Input voltage is below pfi threshold
REGISTERS VALUES:
cap_ov_lvl          : 12534  2300 mV  ichg_uc_lvl         : 0      0
gpi_ov_lvl          : 0      0 mV     msk_alarms          : -8190
    -8190
meas_iin            : 10     10       vin_uv_lvl          : 0      0 mV
num_caps            : 1      1        cap_uv_lvl          : 0      0 mV
vcapfb_dac          : 15     15       gpi_uv_lvl          : 0      0 mV
iin_oc_lvl          : 0      0        dtemp_hot_lvl       : 12907  110
    C
meas_vcap4          : 0      0 mV     clr_alarms          : 0      0
ctl_reg             : 0      0        vout_ov_lvl         : 0      0 mV
meas_vcap2          : 11544  2118 mV  vshunt              : 14745  2705
     mV
meas_cap            : 0      0 F      mon_status          : 354    354
meas_vcap           : 2877   4246 mV  vout_uv_lvl         : 0      0 mV
meas_ichg           : -2261  -2261    meas_vout           : 2205   4873
     mV
chrg_status         : 6698   6698     cap_lo_lvl          : 249    60 F
temp1_input         : 10820  10820    vcap_ov_lvl         : 0      0 mV
meas_esr            : 0      0 mR     meas_gpi            : -14    -2
    mV
meas_dtemp          : 10820  51 C     msk_mon_status      : 16     16
meas_vcap3          : -1     0 mV     cap_esr_per         : 6      6
dtemp_cold_lvl      : 0      -251 C   vcap_uv_lvl         : 0      0 mV
meas_vcap1          : 11623  2132 mV  vin_ov_lvl          : 0      0 mV
esr_hi_lvl          : 192    15 mR    alarm_reg           : 0      0
meas_vin            : 10     22 mV
```

Listato 8.8: Status report durante interruzione di corrente in Dynagate 1

Possiamo osservare che le stampe descrivono la situazione: il dispositivo non è collegato alla corrente, si trova nella modalità di backup, quindi fornisce alimentazione al sistema. È notevole anche il fatto che la misurazione di Capacitanza e ESR è stata interrotta durante la sua esecuzione, come nella figura 8.4. Questo fa in modo che la misura di capacitanza e la misura di ESR siano settate a zero. Inoltre, la tensione di input è ad un misero 22 mV. Il Dynagate 1 funziona senza corrente per 100 secondi: tempo sufficiente per inviare segnalazioni e spegnersi senza che ci siano danni al sistema. Durante questo tempo, la tensione di output è costante, mentre la tensione dello stack di condensatori raggiunge i 2300 mV.

### 8.2.4 Risultati test

LTC3350 è in grado di fornire alimentazione di backup per 100 secondi dopo essere scollegato dalla corrente, e di comunicare informazioni sul proprio stato al sistema di elaborazione centrale tramite bus $I^2C$. Il driver è in grado di leggere questi valori. Tutti i valori letti tramite il driver sono plausibili, in entrambi i Dynagate, quindi è possibile concludere che non ci sono errori di comunicazione tra LTC3350 e il kernel di Linux.

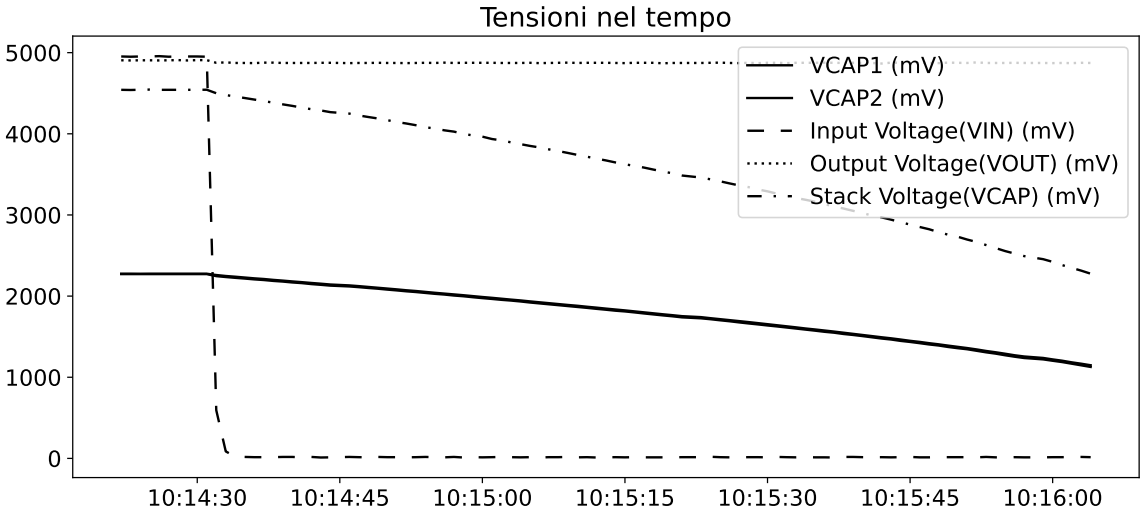



Figura 8.5: Test di interruzione di corrente su Dynagate 1

## 8.3 Documentazione

È essenziale documentare il funzionamento di un driver di Linux, per permettere agli altri sviluppatori di comprenderlo, utilizzarlo e, se necessario, modificarlo. Ho scritto una documentazione in due parti: la prima descrive le caratteristiche del driver e spiega come utilizzarlo[30], la seconda indica come modificare il device tree per l'integrazione di LTC3350 nel sistema.[29] Anche l'applicazione ltc-monitor ha una propria documentazione, che ne descrive l'utilizzo e il metodo di compilazione.[31] Il codice è open-source, ed è pubblicato con la speranza che sia utile in futuro.

# Capitolo 9

# Conclusioni

Il driver prodotto durante questo progetto di tirocinio soddisfa i requisiti, creando un'interfaccia tra il dispositivo LTC3350, il kernel di Linux e lo strato applicativo, e permettendo a diversi tipi di applicazioni di sfruttare le funzionalità di LTC3350.

Considero questo progetto un successo perchè il driver sarà integrato nella prossima versione della distribuzione di Eurotech Everyware Linux per Dynagate. L'applicazione ltc-monitor sarà inviata agli sviluppatori di Eurotech, che potranno decidere se utilizzarla out-of-the-box, oppure come un esempio per sviluppare applicazioni più specifiche alle loro necessità.

**Prospettive future** Sarebbe interessante testare il driver in un sistema diverso dal Dynagate, ad esempio un sistema con un numero diverso di supercondensatori, o con una implementazione diversa dell'hardware di LTC3350.

Una prospettiva per il driver sarebbe la pubblicazione come parte della branch stable del kernel: questo processo richiede molti test e partecipazione da parte di sviluppatori del Kernel con più esperienza. Dovendo cominciare questo progetto da capo, seguirei uno standard diverso dallo standard hwmon: questo renderebbe più semplice la sua integrazione in Linux. Non avendo più accesso al Dynagate, devo relegare questa opera ad uno sviluppatore del futuro, sperando possa prendere ispirazione dal mio lavoro.

**Ringraziamenti** Ci tengo a ringraziare il professor Paolo Milazzo per il suo supporto nella scrittura di questa relazione, e per avermi dato il mio primo trenta, tanti anni fa. Grazie al mio tutor in Abinsula, Davide Salaris, per il suo supporto costante e il suo consiglio, e per le tantissime cose che mi ha insegnato.

Ringrazio anche la mia famiglia e i miei amici per avermi supportato nel mio percorso accademico fino ad oggi, specialmente a chi mi ha dato aiuto e supporto morale durante i momenti più difficili e più importanti.

Grazie al mio amico che mi ha accompagnato dall'inizio della mia carriera accademica fino ad oggi, motivandomi e incoraggiandomi in ogni momento, condividendo

le mie gioie e consolandomi nei miei fallimenti, senza di cui non sarei arrivata a questo traguardo.

Grazie a Sofia. Non posso descrivere quanto la sua presenza sia stata importante negli ultimi anni della mia vita, ma posso ringraziarla per aver letto fino alla fine questa relazione.

# Bibliografia


[1] Daniel P. Bovet e Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005. ISBN: 978-0-596-00565-8.

[2] Alberto Liberal de los Rìos. *Linux Driver Development for Embedded Processors*. Alberto Liberal de los Rìos, 2018.

[3] Michael Dahlin Thomas Anderson. *Operating Systems Principles & Practice*. Recursive Books, Ltd., 2015.

[4] Al Williams. *Linux Kernel From First Principles*. 2023. URL: hackaday.com/2023/08/13/linux-kernel-from-first-principles.

[5] Alessandro Rubini Jonathan Corbet e Greg Kroah-Hartman. *Linux Device Drivers: Where the kernel meets the hardware*. O'Reilly Media, 2005. ISBN: 978-0-596-00590-0.

[6] David Money Harris e Sarah L. Harris. *Sistemi Digitali e Architettura dei Calcolatori*. Trad. da O. Scarabattolo. Zanichelli, 2017.

[7] *Signal(7) - Linux Manual Pages*. Free Software Foundation. 11 Apr. 2020. URL: https://www.man7.org/linux/man-pages/man7/signal.7.html.

[8] Behzad Razavi. *Fundamentals of Microelectronics*. Second Edition. John Wiley & Sons, 2014.

[9] Andrew S. Tanenbaum. *Structured Computer Organization*. Fifth Edition. Prentice Hall, 2005.

[10] Rudolf Sosnowsky e Lukas Dehling. *Computer and automation - Der PC im Chip*. 2015. URL: www.computer-automation.de.

[11] Winfield Hill Paul Horowitz. *The Art of Electronics*. Cambridge University Press, 1989.

[12] Joseph Wu. *A Basic Guide to I2C*. 2022. URL: www.ti.com/lit/an/sbaa565/sbaa565.pdf.

[13] *$I^2C$-bus specification and user manual*. Ver. Rev. 7.0. Document Identifier: UM10204. NXP Superconductors. 10 Gen. 2021. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[14] *System Management Bus Specification*. Ver. Revision 1.0. Intel Corporation. 15 Feb. 1995. URL: https://www.smbus.org/specs/smb10.pdf.

[15] Laszlo Kiraly e Robert Schreiber. *Smart Battery Charger with SMBus Interface*. Linear Technology e Microchip Technology. 1997. URL: https://ww1.microchip.com/downloads/en/AppNotes/00667a.pdf.

[16] Robert V. White. *System Management Bus (SMBus) Specification*. Ver. 3.3. System Management Interface Forum. 12 Mag. 2024. URL: www.smbus.org/specs/SMBus%5C_3%5C_3%5C_20240512.pdf.

[17] Analog Devices. *LTC3350_SDK*. Accessed: 22/10/2024. 2016. URL: www.analog.com/media/en/evaluation-boards-kits/evaluation-software/LTC3350%5C_SDK.zip.

[18] Jean Delvare. *i2c: Add SMBus alert support*. 2010. URL: lwn.net/Articles/374391/.

[19] *SMBus protocol documentation - the Linux Kernel*. 2020. URL: Linux/Documentation/i2c/smbus-protocol.rst.

[20] M'boumba Cedric Madianga. *Driver for STMicroelectronics STM32F7 I2C controller*. 2017. URL: Linux/drivers/i2c/busses/i2c-stm32f7.c.

[21] *Reference Manual for STM32F75 and STM32F74*. STMicroelectronics. 2018. URL: www.st.com/resource/en/reference%5C_manual/dm00124865.pdf.

[22] The kernel development community. *$I^2C$ and SMBus Subsystem*. URL: www.kernel.org/doc/html/latest/driver-api/i2c.html.

[23] The kernel development community e Guenter Rock. *The Hardware Monitoring kernel API*. URL: www.kernel.org/doc/html/latest/hwmon/hwmon-kernel-api.html.

[24] Greg Kroah-Hartman. *Everything you never wanted to know about kobject, ksets, and ktypes*. 2007. URL: www.kernel.org/doc/html/latest/core-api/kobject.html.

[25] Patrick Mochel e Mike Murphy. *sysfs - **The** filesystem for exporting Kernel object*. 2011. URL: www.kernel.org/doc/html/latest/filesystems/sysfs.html.

[26] The kernel development community. *Linux Filesystem API summary*. URL: www.kernel.org/doc/html/latest/filesystems/api-summary.html.

[27] Andrea Merello. *STTS751 sensor driver*. Istituto italiano di tecnologia. 2016. URL: Linux/drivers/hwmon/stts751.c.

[28] *poll(2) - Linux Manual Pages*. Free Software Foundation. 11 Apr. 2020. URL: https://www.kernel.org/doc/man-pages/man2/poll.2.html.

[29] Francesca Pinna. *Device tree bindings del driver ltc3350-driver*. 2024. URL: https://github.com/c-facade/ltc3350-driver/blob/main/Documentation/devicetree/bindings/hwmon/ltc3350.yaml.

[30] Francesca Pinna. *Documentazione di* ***ltc3350-driver***. 2024. URL: https://github.com/c-facade/ltc3350-driver/blob/main/Documentation/hwmon/ltc3350.rst.

[31] Francesca Pinna. *Codice sorgente di* ***ltc-monitor***. 2024. URL: https://github.com/c-facade/ltc-monitor.