Alma Mater Studiorum · Università di Bologna

scuola di ingegneria e architettura

dipartimento di ingegneria dell'energia elettrica e dell'informazione "guglielmo marconi" - dei

corso di laurea magistrale in ingegneria dell'energia elettrica

Sviluppo di Driver con Interfaccia OS-Independent per la piattaforma PULP

Tesi di Laurea in Architetture e Programmazione dei Sistemi Elettronici

Relatore:
Prof. Davide Rossi

Candidato:
Nico Orlando

Correlatori:
Dott. Corrado Bonfanti
Dott. Mattia Sinigaglia
Dott. Giuseppe Tagliavini

Anno Accademico 2020/2021
III Sessione

A tutti coloro
che durante questo
viaggio mi sono
stati vicini ...

Indice

Elenco delle figure

Abstract

I dispositivi endpoint per Internet-of-Things non solo devono funzionare con un inviluppo di potenza estremamente ridotto di pochi milliwatt, ma devono anche essere flessibili nelle loro capacità di elaborazione.
Attraverso un System on Chip (SoC) parallel ultra-low power (PULP) caratterizzato da un'architettura gerarchica con core RISC-V di classe microcontroller (MCU) di piccole dimensioni e potenziato con un sottosistema IO autonomo per un'efficiente trasferimento dati da un'ampia gamma di periferiche, si riesce a soddisfare i requisiti dei nodi endpoint IoT.
In questo elaborato ci si è concentrati sullo sviluppo dei driver per le periferiche di comunicazione SPI e I^2C in modo che queste potessero rispettare le specifiche Hard Real-Time dei sistemi RTOS. Per poter sfruttare FreeRTOS e PULP-OS, presenti nel SDK, i driver sono stati realizzati in modo da essere OS-Indipendent. Per fare questo si sono implementate delle Microcontroller Software Interface Standard in grado di poter creare un ulteriore livello di astrazione per il software, estraendo le funzionalità degli RTOS e definendo interfacce comuni per driver e periferiche.
Quello che si è ottenuto è la realizzazione di un SDK in grado di poter utilizzare due RTOS differenti in base alle esigenze applicative.

Capitolo 1

Introduzione

Gli ultimi anni hanno visto una crescita esplosiva di piccoli dispositivi alimentati a batteria che rilevano i parametri dell'ambiente in cui sono e comunicano in modalità wireless i dati rilevati dopo alcune elaborazioni, riconoscimenti o classificazioni dei dati. Questi dispositivi, che prendono il nome di dispositivi endpoint IoT, Internet of Things, condividono la necessità di un'estrema efficienza energetica e inviluppi di potenza di pochi milliwatt. Nelle architetture hardware dei microcontrollori destinate a tali applicazioni, la parte di acquisizione dati è implementata da un sottosistema di sensing, realizzato con sensori a bassa potenza. I dati acquisiti vengono quindi elaborati digitalmente con microcontrollori a bassa potenza, MCU, e trasmessi attraverso un sottosistema di comunicazione wireless, costituito da ricetrasmettitori radio TX/RX a bassa energia e implementati a batteria. Sebbene il sottosistema di sensing, il sistema di acquisizione dati e i ricetrasmettitori, siano talvolta suddivisi su più di un chip, il mercato è orientato verso una soluzione a chip singolo completamente integrata. L'intero nodo IoT è alimentato da batterie con fattore di forma ridotto , composte da un sottosistema di alimentazione e conversione. Nonostante una riduzione di quasi 10 volte della potenza del ricetrasmettitore in pochi anni, la sua quota nel budget energetico complessivo della maggior parte dei nodi di sensori wireless e dei dispositivi indossabili rimane dominante. In questo scenario, conviene

aumentare la complessità dell'analisi e del filtraggio dei dati vicino al sensore fornendo più potenza di calcolo al sottosistema di elaborazione. Questo approccio può ridurre drasticamente la quantità di dati wireless trasmessi. Per questo motivo, la disponibilità di hardware che permette una elaborazione digitale potente, flessibile ed efficiente dal punto di vista energetico, in prossimità dei sensori, gioca un ruolo chiave nella rivoluzione IoT. Un approccio per ottenere una maggiore efficienza energetica e prestazioni è quello di dotare MCU di sistemi Digital Signal Processing, DSP, i quali raggiungono prestazioni molto elevate durante l'elaborazione dei dati, ma non sono flessibili come un processore e sono difficili da programmare. Un'efficienza energetica ancora maggiore può essere raggiunta con acceleratori dedicati per l'elaborazione del segnale, come ad esempio le Fast Fourier Transformations, FFT. Queste sono implementate mediante un blocco hardware dedicato, che è controllato da un MCU. Una tale combinazione di MCU e acceleratore FFT è superiore in termini di prestazioni, ma anche molto specializzato e quindi non molto flessibile e scalabile. Attraverso la piattaforma PULP, Parallel Ultra Low Power, si ha una piattaforma flessibile, scalabile ed efficiente dal punto di vista energetico con core programmabili che consumino solo un paio di milliwatt. Sebbene l'efficienza energetica di un blocco hardware personalizzato non possa mai essere raggiunta con core programmabili, è possibile costruire una piattaforma multicore flessibile e scalabile con un'efficienza energetica molto elevata. Le operazioni ULP, Ultra Low Power, possono essere ottenute sfruttando il regime di tensione vicino alla soglia, near-threshold, NT, in cui i transistor diventano più efficienti dal punto di vista energetico. La perdita di prestazione può essere compensata sfruttando il calcolo parallelo. Tali sistemi possono superare gli equivalenti single-core in quanto possono funzionare a una tensione di alimentazione inferiore per ottenere lo stesso throughput. In PULP viene utilizzato un Instruction Set Architecture, ISA, di tipo open source in quanto può potenzialmente ridurre la dipendenza da un singolo provider IP e consente allo stesso tempo di poter applicare estensioni specifiche alle istruzioni in base all'applicazione. Pertanto, PULP è

compatibile con una microarchitettura basata su RISC-V ISA, che raggiunge prestazioni e densità di codice simili agli MCU all'avanguardia basati su un ISA proprietario, come i core della serie ARM Cortex-M.

Lo scopo del mio lavoro è stato quello di implementare inizialmente dei test bare-metal per verificare il corretto funzionamento del Interrupt Controller. Per fare questo si sono utilizzate le periferiche di comunicazione SPI e I^2C, con le quali si è andato a scrivere/leggere rispettivamente su una memoria Flash ed EEPROM. Dopodichè ho realizzato un abstraction layers per i driver delle periferiche di comunicazione SPI e I^2C, in modo tale che queste potessero essere utilizzate su due RTOS differenti, cioè FreeRTOS e PULP-OS. Per verificare i due abstraction layers ho realizzato test sincroni e asincroni.

I capitoli seguenti descrivono nel secondo capitolo la piattaforma PULP, il SoC PULP e le sue periferiche, come il uDMA, SPI e I^2C. In questo capitolo si sono anche mostrati i comandi per utilizzare, attraverso il uDMA, la periferica di comunicazione SPI e I^2C oltre che la descrizione del protocollo per SPI e I^2C, e del Interrupt. Nel terzo capitolo invece si sono descritti gli RTOS utilizzati, con particolare attenzione alla gestione dei event task in PULP per i due RTOS. Nel quarto capitolo si è descritto l'abstraction layer per la periferica di comunicazione SPI e I^2C, con i rispettivi test fatti per validare il funzionamento di questi driver.

Capitolo 2

PULP Platform

PULP (Parallel Ultra Low Power) è una piattaforma di elaborazione multi-core open source, che mira all'alta efficienza energetica, sviluppata dalla collaborazione tra il gruppo di ricerca dell'Energy-Efficient Embedded Systems (EEES) dell'Università di Bologna e l'Integrated Systems Laboratory (IIS) dell'ETH Zürich.

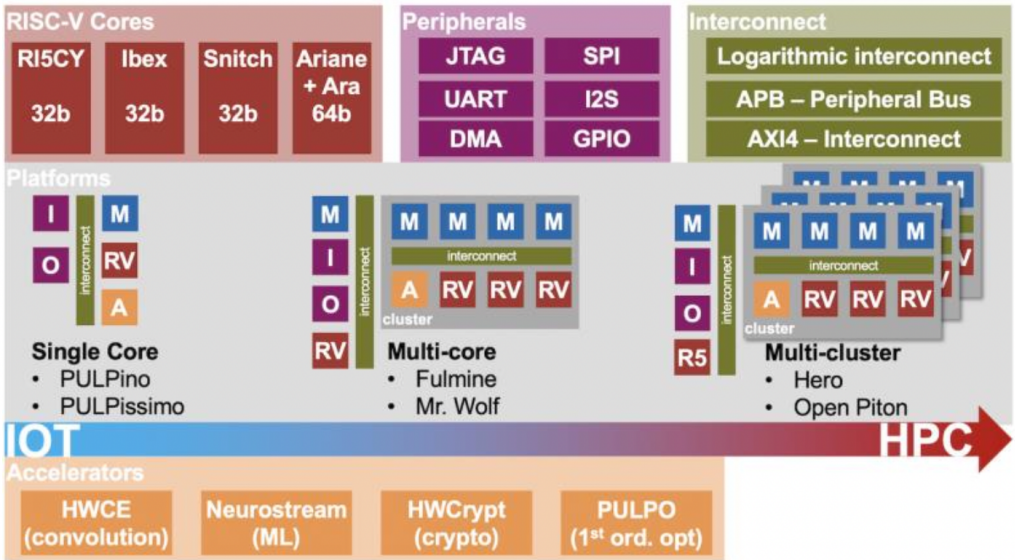

Figura 2.1: PULP Platform

La piattaforma PULP è costituita da un MCU single-core, che prende il nome di PULPissimo, e da un MCU multi-core, che prende invece il nome di PULP. Entrambi gli MCU sono compatibili con il set di istruzioni open-source di RISC-V. L' obbiettivo di questi MCU è quello di poter soddisfare le esigenze computazionali delle moderne applicazioni IoT e disporre di una elaborazione performante in funzione dei dati provenienti dai sensori. Rispetto alle altre famiglie di MCU single-core, l'architettura multi-core di PULP consente di soddisfare le esigenze computazionali delle applicazioni IoT, senza superare il consumo tipico di pochi mW.

2.1 PULP

PULP è un SoC, System-on-a-Chip, composto da:

- Fabric Controller (FC), RI5CY single-core a 32 bit, che rispetta le specifiche del RISC-V ISA, il quale viene utilizzato per funzioni di controllo, comunicazioni e sicurezza. Questo può essere visto come un classico MCU.

- Un Cluster, composto da 4 a 16 RI5CY single-core a 32 bit, che rispetta le specifiche del RISC-V ISA, con un'architettura ottimizzata per l'esecuzione di algoritmi vettorializzati e parallelizzati.

La Fig. 2.2 mostra il core RI5CY utilizzato. Questo ha una pipeline a 4 stadi e Instruction Set Architecture (ISA) RV32IMC + estensioni tramite le quali è possibile eseguire istruzioni specifiche per migliorare performance di algoritmi DSP e machine learning. Queste estensioni includono hardware-loop, pointer post-increment, Multiply And Accumulate (MAC) e vettorizzazione SIMD. La 2.3 mostra la memory-pap per PULP.

L'esecuzione parallela nel cluster PULP richiede un'architettura di memoria scalabile per ottenere velocità quasi ideali nelle applicazioni parallele, riducendo al contempo la potenza. Di seguito viene descritta l'architettura di memoria in un cluster PULP.

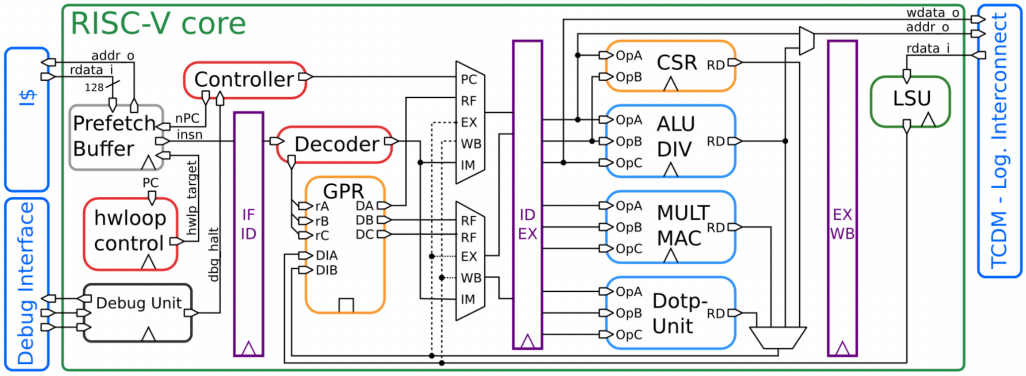

Figura 2.2: RISCY core overview

Un cluster PULP supporta un numero configurabile di core con una cache di istruzioni condivisa, che viene sostituita da una memoria scratchpad, la quale è meno costosa in termini di accesso. Le cache delle istruzioni del fabric controller (1kB) e del cluster (4kB) memorizzano automaticamente le istruzioni secondo necessità. La cache delle istruzioni del cluster è condivisa tra tutti i core del cluster. Generalmente, i core del cluster eseguiranno la stessa area di codice su dati diversi, quindi la cache di istruzioni del cluster condivisa sfrutta questo per ridurre gli accessi alla memoria per il caricamento delle istruzioni.
Vediamo quale è la differenza tra una memoria cache e una memoria scratchpad. In un sistema multiprocessore la gestione della cache diventa un problema pressante dato che bisogna garantire che le cache non contengano dati vecchi, quindi ogni volta che un processore modifica un dato che risulta presente anche in una cache di un altro processore il blocco di dati nella cache del processore va invalidato, questo genera un elevato flusso di dati tra i processori che rallenta il sistema. La scatchpad invece è una memoria direttamente indirizzabile quindi se un processore vuole accedere a dei dati in una scatchpad di un altro processore può farlo direttamente senza dover invalidare la cache del processore proprietario della scatchpad. Questo semplifica la realizzazione hardware e riduce il numero di sincronizzazioni tra

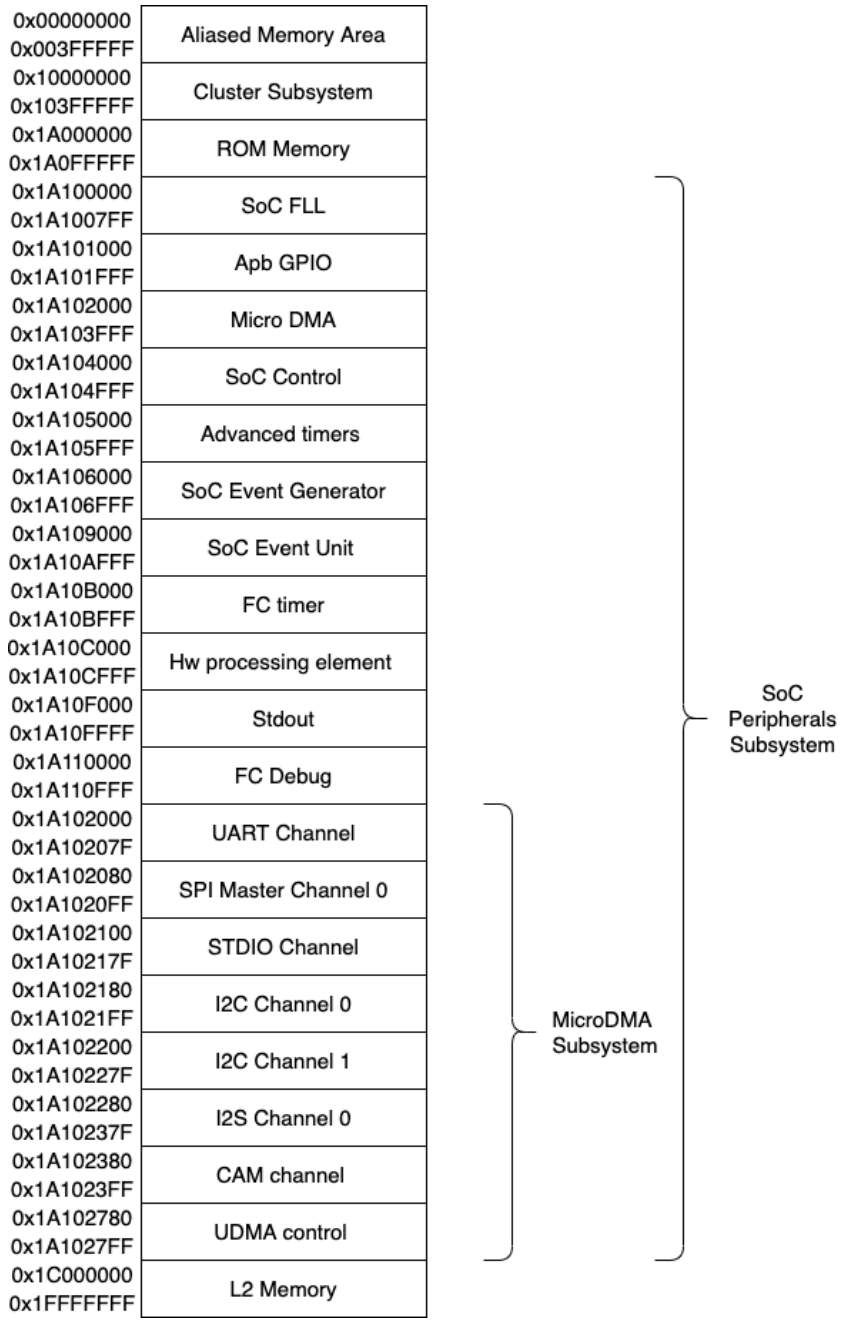

Figura 2.3: PULP Memory Map

i processori. Entrambe le porte di memoria hanno una latenza variabile e possono arrestare la pipeline.

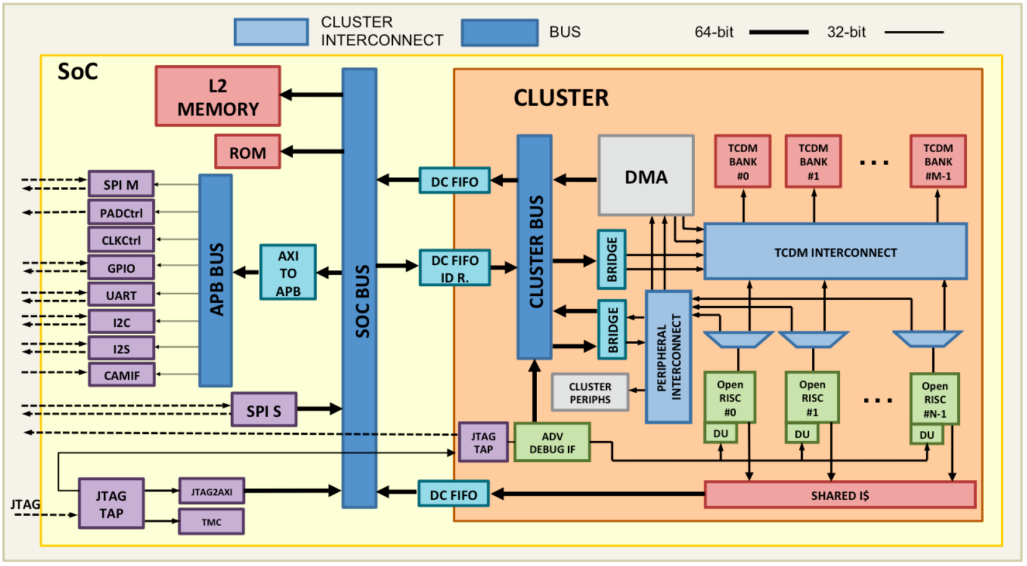

Figura 2.4: PULP Overview

Vengono poi utilizzate le memorie tightly coupled-data-memories (TCDMs), le quali possono essere condivise da più core per consentire l'elaborazione sulla stessa struttura dati senza introdurre overhead hardware.

Tutti gli elementi del SoC condividono l'accesso a un'area di memoria L2, di 512 kB. I core del cluster condividono l'accesso a un'area di memoria L1, 128 kB, (TCDM) e alla cache delle istruzioni.

PULP può anche accedere ad aree di memoria esterne tramite le interfacce HyperBus (Flash o RAM) o quad-SPI (Flash). Poiché il costo energetico e di prestazioni dell'accesso alla memoria RAM esterna, tramite bus esterni, è molto elevato rispetto alla memoria interna, in genere ciò dovrebbe essere evitato il più possibile. Di conseguenza, il codice del programma viene caricato dalla flash esterna all'avvio nell'area di memoria L2.

All'interno di PULP, più unità di DMA consentono trasferimenti auto-

nomi, veloci e a bassa potenza tra la memoria del cluster L1 e L2 e tra la memoria L2 e le periferiche esterne.
In PULP il uDMA fornisce il trasferimento diretto dei dati tra la memoria L2 e le diverse interfacce fornite in PULP. Questo aiuta a ridurre il carico di lavoro del fabric controller. Si possono gestire fino a 11 canali con il uDMA:

- Dalla Camera alla memoria L2;
- Dalla I2S0 alla memoria L2;
- Dalla I2S1 alla memoria L2;
- Dalla I2C0 alla memoria L2, e viceversa;
- Dalla I2C1 alla memoria L2, e viceversa;
- Dalla UART alla memoria L2, e viceversa;
- Dalla SPIM0 alla memoria L2, e viceversa;

La larghezza dei trasferimenti con il uDMA può essere selezionata tra 8, 16 o 32 bit. È possibile trasferire fino a 128kB durante una singola transazione (8kB per SPIM). In generale, le transazioni possono essere bidirezionali ma, a seconda dell'interfaccia, in alcuni casi è disponibile solo una direzione.

2.2 Stack Software in PULP

Il termine Stack Software si riferisce all'insieme di componenti che interagiscono per supportare l'esecuzione di una applicazione. I componenti di uno stack software lavorano insieme per fornire in modo efficiente servizi applicativi all'utente finale. Lo Stack Software è una raccolta di componenti indipendenti che collaborano per semplificare l'implementazione si una applicazione. I livelli inferiori nella gerarchia spesso si interfacciano con l'hardware, mentre i livelli superiori nella gerarchia eseguono task e servizi utente specifici.

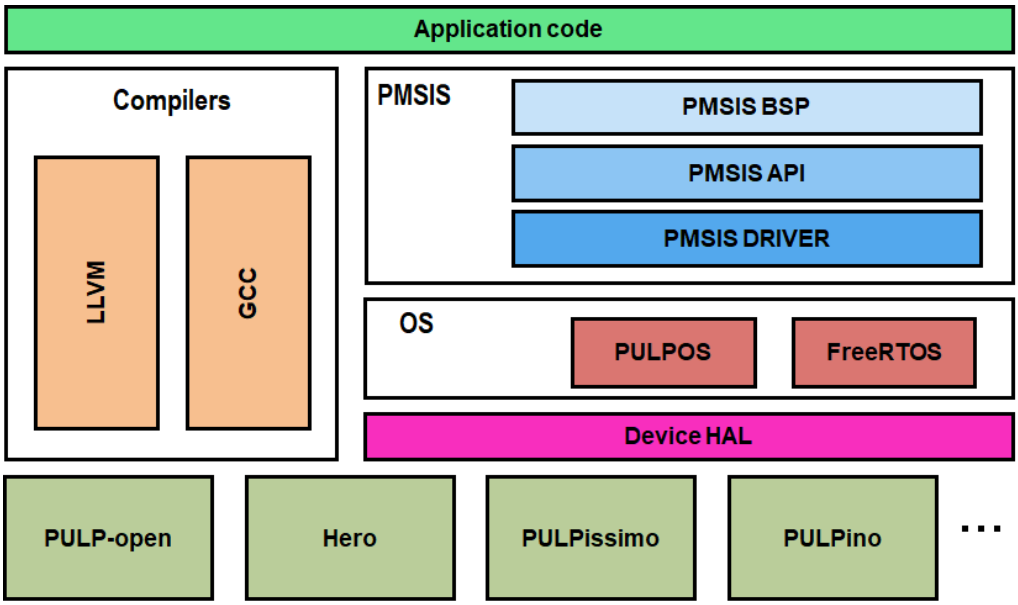

Figura 2.5: Stack Software in PULP

Nella Fig. 2.5 viene mostrato lo Stack Software per la piattaforma PULP. Dalla figura si dimostra che a livello più basso dello stack software è presente hardware che si utilizza. Nel caso della piattaforma PULP i tipi di hardware compatibili possono essere PULP-open, Hero, PULPissimo e PULPino. Poi si ha il Device HAL, Hardware Abstraction Layer. HAL è un livello di programmazione che consente di interagire con un dispositivo hardware a un livello più alto. Qui ci sono ad esempio le definizioni dei vari indirizzi per accedere ai registri delle periferiche oppure funzioni per poter andare a scrivere in registri specifici del core. HAL può essere chiamato dal kernel del sistema operativo o da un driver di dispositivo. Successivamente si hanno gli RTOS, che possono essere FreeRTOS o PULP-OS, con i quali si vanno a gestire i task in esecuzione. L'interfaccia tra l'utente e i componenti appena descritti, viene realizzata mediate le PMSIS, PULP Microcontroller Software Interface Standard. Queste servono per creare un ulteriore livello di astrazione per il software, estraendo le funzionalità di sistema operativo e definendo interfacce comuni per driver e periferiche. Le PMSIS BSP si utilizzano per creare un

interfaccia con device specifici, come la Camera, Display TFT e HyperRam. Nelle PMSIS API sono presenti le API, Application Programming Interface, che si devono utilizzare per l'implementazione dei driver presenti in PMSIS DRIVER. In PMSIS DRIVERS, sono implementati i driver per le periferiche di comunicazione SPI, I^2C e UART. La compilazione del codice può avvenire mediante i compilatori LLVM e GCC.

	Frequency	Cycle-accurate	Effort
Event-based RTL	1Hz-1KHz	Yes	Low
Compiled RTL	500Hz-5KHZ	Yes	Low
Architectural simulators	10KHz-2MHz	No	High
FPGA/Emulators	1-100MHz	Yes	High

Figura 2.6: Different Levels of abstraction

L'intera architettura PULP può essere utilizzata su più piattaforme, ad esempio RTL, FPGA o GVSoC. Il System Verilog è un Hardware Description Language, HDL, con lo scopo di descrivere il design in Register Transfer Level , RTL, ovvero specificare il comportamento di un circuito digitale per ogni ciclo di clock in termini di segnali, registri ed operazioni logiche. La FPGA, Field Programmable Gate Array, è un dispositivo logico programmabile formato da un circuito integrato le cui funzionalità logiche di elaborazione sono appositamente programmabili.

È possibile classificare quattro tipi di simulatori hardware, riassunti nella Fig. 2.6. I simulatori RTL basati su eventi, come Modelsim, e compilati, richiedono uno sforzo minimo per simulare direttamente il codice HDL e sono accurati per il ciclo. Sfortunatamente, sono troppo lenti per simulare carichi di lavoro complessi e modelli di sistema completi. L'emulazione può essere anche eseguita con motori di emulazione commerciali, che sono sistemi multi-FPGA proprietari ed estremamente costosi e non possono essere facilmente utilizzati in una configurazione hardware-in-the-loop, cioè con ingressi e uscite a dispositivi esterni reali, come sensori, attuatori e memorie. Infine, l'emulazione può essere eseguita anche con schede single-FPGA, che sono

ugualmente veloci e consentono all'architettura emulata di interfacciarsi con sensori e attuatori, a condizione che le singole risorse FPGA siano sufficienti per emulare il design mirato. GVSOC è un simulatore di set di istruzioni in grado di simulare PULP, e consente l'esecuzione di programmi su una piattaforma virtuale senza alcun limite hardware.

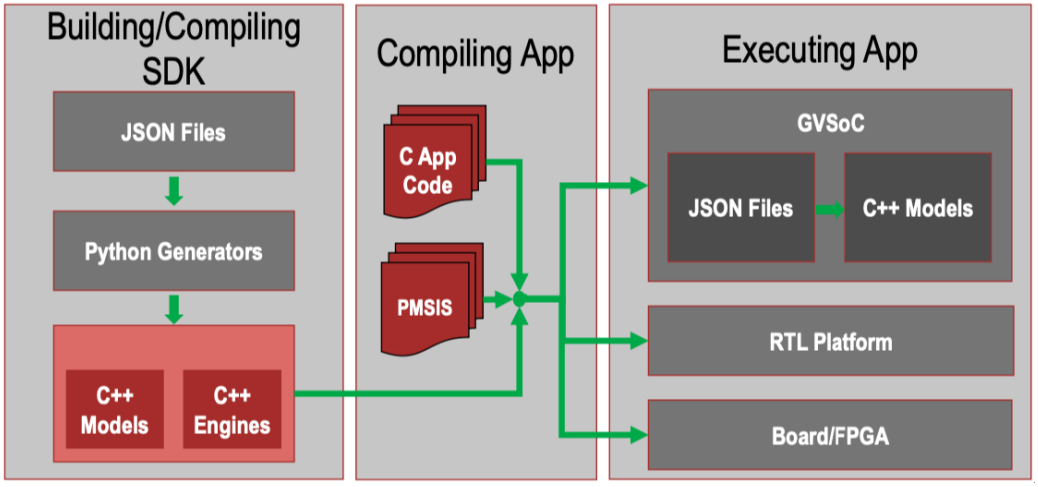

Figura 2.7: PULP Software Environment

La Fig. 2.7 mostra cosa succede durante l'esecuzione del codice su una piattaforma che può essere RTL, FPGA o GVSoC. Attraverso i file di configurazione del target, scritti in formato json, si vanno a impostare le informazioni sulla particolare architettura che si sta utilizzando. Ad esempio il file pulp.json contiene le specifiche dei core che si stanno utilizzando, l'indirizzo base della memory map, la versione del RISC-V, l'indirizzo di boot e il numero di core del cluster. Il file udma.json invece conterrà le specifiche del uDMA, il numero di periferiche ad esso connesso e gli indirizzi di memoria. Nella sezione di Python Generation ci sono degli script in Python che fanno il build del chip che si simulerà sulle varie piattaforme, il quale avrà le specifiche definite nel target. Dopodichè c'è la parte di compilazione, la quale compila application code e le librerie di PMSIS. Infine si andrà a simulare il programma in GVSoC, RTL o FPGA. La piattaforma GVSoC è composta

da due blocchi. Il blocco ENGINE contiene le informazioni sul clock, trace, eventi e timer, mentre il blocco MODEL contiene i modelli dei dispositivi, come le memorie, che si possono utilizzare in GVSoC. Il JSON Files contiene informazioni che possono andare a modificare l'esecuzione del programma.

2.3 μDMA Subsystem

In questo capitolo si mostrerà come viene implementato il uDMA e come questo gestisce il trasferimento dei dati da/verso le periferiche.

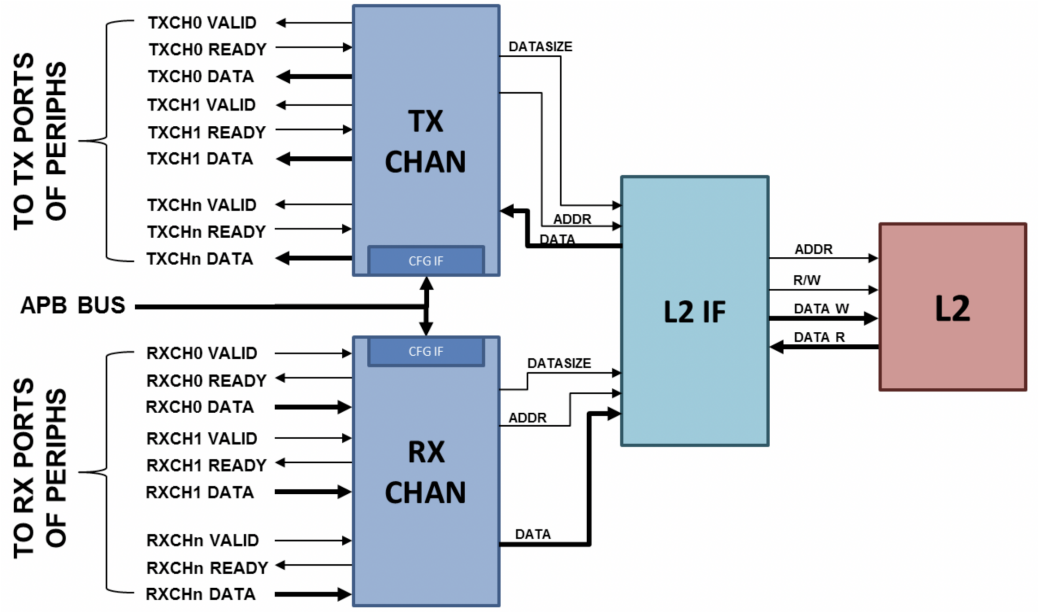

Figura 2.8: uDMA top level

In figura 2.8 si può vedere il diagramma a blocchi di alto livello del uDMA. I componenti principali sono i canali tx, i canali rx, l'interfaccia per la memoria L2 e il blocco di configurazione. Ogni periferica può avere uno o più canali tx o rx a seconda delle necessità. Ad esempio, SPI richiederà un canale tx e un canale rx. L'interfaccia tra le periferiche e il uDMA è stata implementata con un semplice protocollo valid/ready. Le periferiche impostano il segnale

valid dei canali rx quando hanno dati disponibili e generano il segnale ready di un canale tx quando richiedono dati dalla memoria. Questi canali sono ora l'unico modo per controllare il flusso di dati tra le periferiche e uDMA. Tutti gli altri controlli delle periferiche vengono effettuati accedendo direttamente all'interfaccia di configurazione di ciascuna periferica.
Queste connessioni dirette con la memoria L2 limitano il uDMA ad accedere solo alla memoria di sistema e non permettono lo scambio diretto tra CPU, cluster o altre periferiche mappate sul bus APB.
Ciascun canale uDMA, sia tx che rx, ha delle risorse dedicate nell'interfaccia di configurazione. Le risorse hardware associate a ciascun canale sono:

1. Registro degli indirizzi a 32 bit, che memorizza l'indirizzo del trasferimento successivo;

2. Registro a 16 bit, che memorizza il numero di byte rimasti nel trasferimento corrente;

3. Registro della dimensione dei dati a 2 bit, che mantiene la quantità di byte da trasferire ad ogni trasferimento uDMA. Possibili valori sono 1,2 e 4.

Dopo la prima configurazione del trasferimento, il software può avviare il primo trasferimento con un indirizzo di destinazione e subito dopo accodare un altro trasferimento con un secondo indirizzo di destinazione. Non appena il primo trasferimento è terminato, il software può accodare nuovamente una transazione con il primo indirizzo di destinazione ed elaborare il primo blocco di dati mentre il secondo blocco viene trasferito al secondo indirizzo di destinazione in background.

2.3.1 RX Channel

Lo schema a blocchi della logica che gestisce l'handshake con le periferiche ed esegue i trasferimenti in memoria è mostrato in Fig. **??**, mentre l'handshake del canale è mostrato in Fig. 2.10.

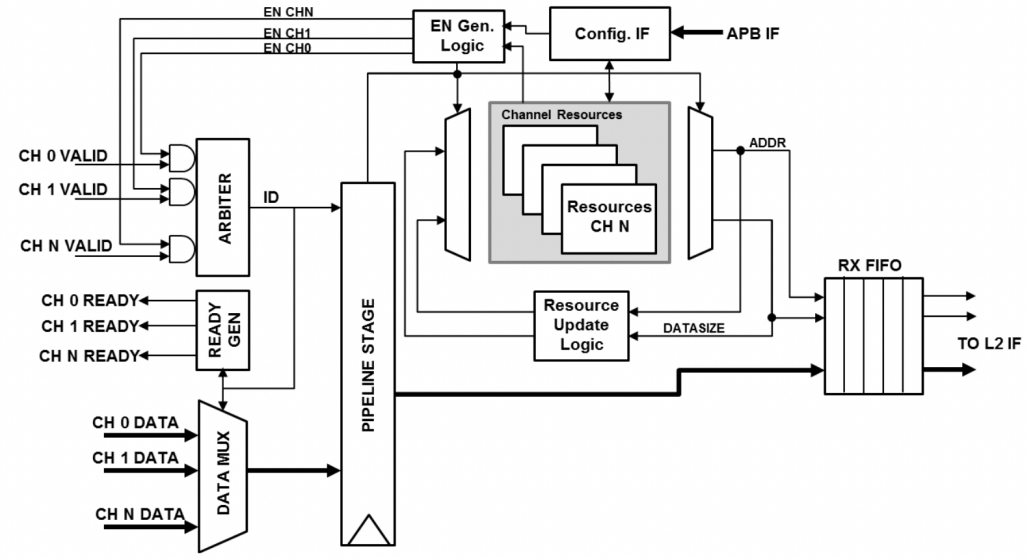

Figura 2.9: UDMA RX Channels architecture

Quando una periferica ha un dato disponibile, pone a 1 il segnale VALID e notifica al uDMA che ha un dato pronto. Nel blocco RX Channel, tutti i segnali VALID, se i canali sono attivi, vengono gestiti con un arbitraggio di tipo Round Robin. In base al ID del canale vincente, vengono presi i dati dalla periferica da inviare alla memoria. ID del canale vincente e i dati, relativi a questo, vengono campionati e inviati alla fase successiva della pipeline. Nella fase successiva della pipeline, ID memorizzato viene utilizzato per selezionare il puntatore corrente associato al canale, la dimensione dei dati del canale e il numero di byte rimasti per il trasferimento. Il puntatore, la dimensione dei dati e i dati stessi vengono inseriti in un FIFO e inviati per il trasferimento. Parallelamente nello stesso ciclo vengono aggiornati il puntatore successivo e il numero di byte rimasti per i trasferimenti. Per ridurre la complessità della logica di retropropagazione degli stalli alle periferiche, ogni canale, una volta vinto un arbitrato, viene disabilitato fino a quando i suoi dati non vengono inseriti nella FIFO RX. Guardando l'effetto sui singoli canali, questo può sembrare un grosso limite poiché dimezza la larghezza di banda. Nei casi di utilizzo reale questo non è mai un problema poiché tutte le periferiche

comportano un qualche tipo di conversione da seriale a parallela e nessuna delle periferiche disponibili è in grado di produrre dati a piena larghezza di banda in grado di saturare la larghezza di banda della memoria, nemmeno durante i picchi.

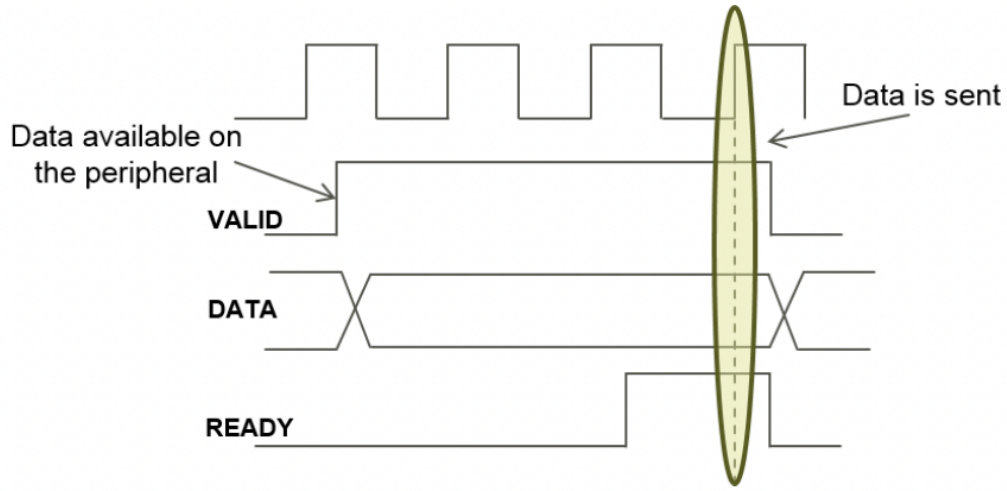

Figura 2.10: UDMA RX protocol

2.3.2 TX Channel

Lo schema a blocchi della logica che gestisce l'handshake con le periferiche ed esegue i trasferimenti in memoria è mostrato in Fig. 2.11, mentre l'handshake di canale è mostrato in Fig. 2.12.

Il blocco TX Channels gestisce il trasferimento dei dati nella direzione opposta, cioè dalla memoria alle periferiche. Le periferiche con un canale TX, quando hanno la pipe pronta per la trasmissione, sollevano il segnale di data READY, informando il uDMA che ora è pronto per accettare nuovi dati. Il uDMA, analogamente a quanto fatto nei canali RX, arbitra le richieste di dati provenienti dai canali attivi e blocca il risultato dell'arbitrato (ID canale) da utilizzare nello stage. Durante la fase successiva della pipeline, in base al ID del canale della periferica vincitrice del arbitraggio, si vanno a selezionare

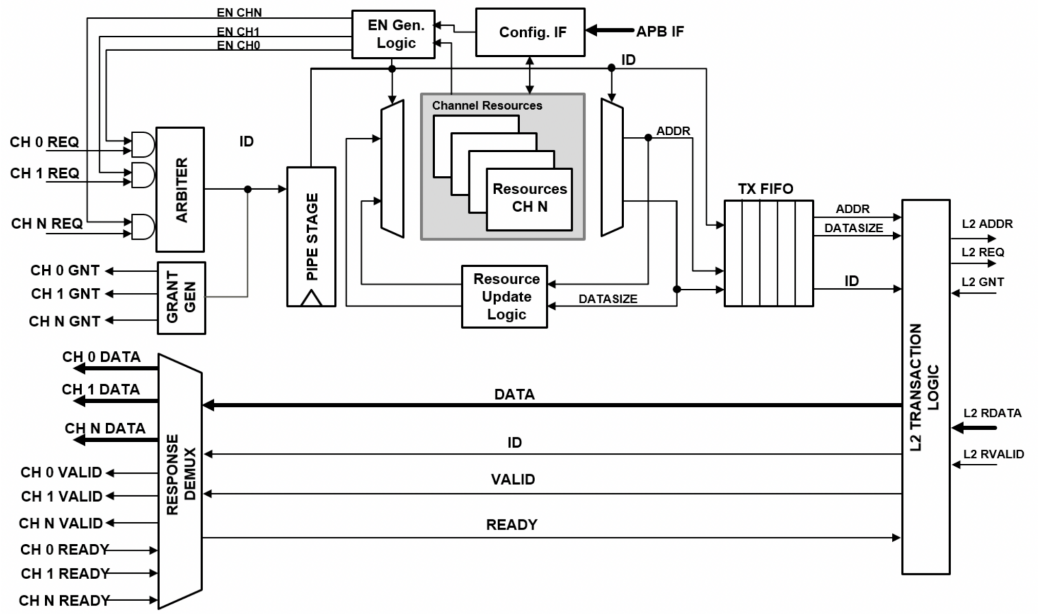

Figura 2.11: UDMA TX Channels architecture

le risorse corrispondenti. Successivamente il puntatore, la dimensione dei dati e l'ID del canale vengono inviati alla FIFO TX. Parallelamente viene aggiornato il nuovo indirizzo e il numero di byte rimasti. Nei canali TX anche l'ID viene inserito nella FIFO TX, perché l'informazione è necessaria per demuxare i dati al canale corretto sul percorso di risposta. A valle della FIFO TX è posta una logica di transazione della memoria che estrae più informazioni dalla FIFO TX ed esegue la lettura del dato da inviare dalla memoria L2. Al termine della lettura, i dati ricevuti vengono poi inviati al canale appropriato. A causa della natura bloccante del semplice protocollo valid/ready, ogni canale TX viene bloccato fino a quando i dati non vengono recuperati dalla memoria. Ciò implica che senza contese possiamo recuperare un dato ogni 4 cicli. Questo imposta la larghezza di banda massima di un canale TX a 1/4 della larghezza di banda L2. Ancora una volta questo non è un limite nell'attuale sottosistema IO poiché la conversione da seriale a parallelo assorbe la latenza.

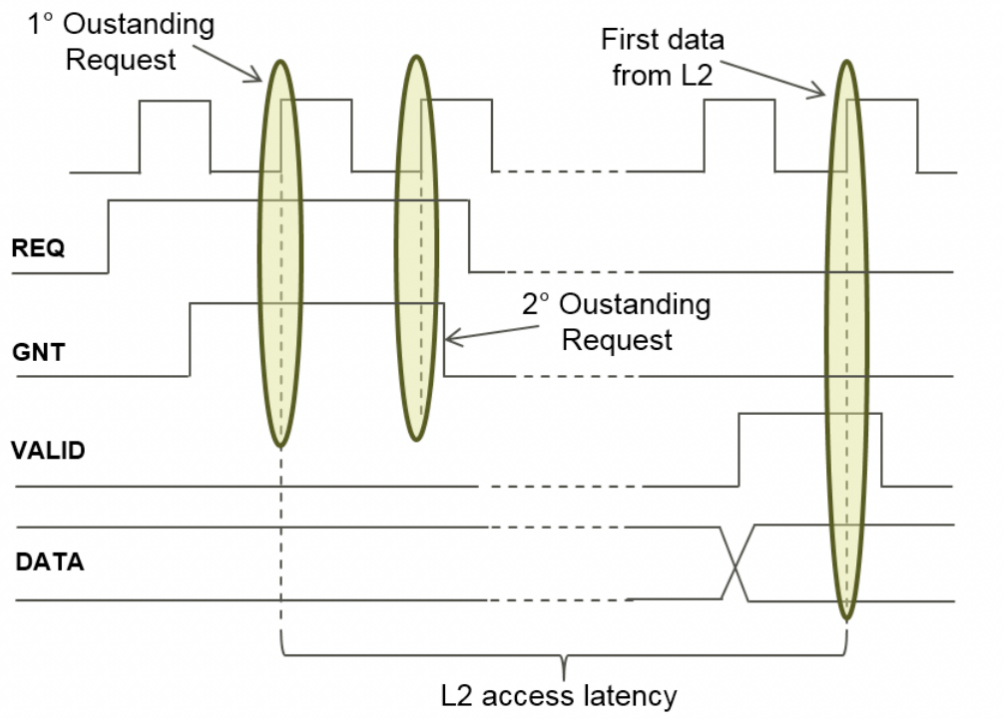

Figura 2.12: UDMA TX protocol

2.4 Periferica SPI in PULP

2.4.1 Protocollo SPI

Il protocollo di comunicazione SPI, Serial Peripheral Interface, è stato ideato da Motorola a supporto dei microcontrollori e microprocessori. Rispetto al protocollo di comunicazione I^2C, non è mai stato standardizzato, e questo ha portato ad avere modi diversi per interfacciarsi con questo protocollo. In confronto al I^2C, il protocollo di comunicazione SPI è più veloce però viene usato per brevi distanze, a causa del driver Push-Pull. Il driver Push-Pull è composto da un transistor PMOS e un NMOS. Il transistor PMOS serve per portare il segnale verso VCC, mentre il transistor NMOS serve per portare il segnale verso ground. Nell'interfaccia I2C si ha che invece del PMOS si ha una resistenza, però come vantaggio si ha quello di poter

collegare più dispositivi

Come si vede dalla Fig. 2.13, l'interfaccia SPI ha quattro linee di collega-

Figura 2.13: Interfaccia SPI e Slave

mento, di cui due di dato, cioè MISO (master-in, slave-out data) e MOSI (master-out, slave-in data) e due linee di controllo, cioè SCK (clock) e CSN (chip select, one per slave – usually active low). La presenza del segnale di clock permette all'interfaccia SPI di andare a sincronizzare più velocemente il master con lo slave.
Data che il protocollo di comunicazione SPI non è stato standardizzato, è possibile trovare diverse definizioni per i segnali descritti prima. Ad esempio, la linea MOSI può prendere il nome di SDO (Serial Data Out), DO (Data Out), DOUT e SO (Serial Out). La linea MISO può prendere il nome di SDI (Serial Data In), DI (Data In), DIN e SI (Serial In). La linea di clock può prendere il nome di CLK, SCK (Serial Clock). La linea di Enable può prende il nome di CS (Chip Select), CE (Chip Enable) o SS (Slave Select).

Di seguito vengono riportate le caratteristiche principali del protocollo SPI.

- La periferica è di tipo seriale, quindi i bit sono trasferiti lungo un canale di comunicazione uno di seguito all'altro e giungono sequenzialmente al ricevente nello stesso ordine in cui li ha trasmessi il mittente.

- Il master può essere connesso a più slave, e può scegliere con chi comunicare attraverso il segnale di Slave Select.

- La comunicazione è di tipo Full-duplex. Di conseguenza si può avere che il master trasmetta i dati allo slave e in parallelo lo slave trasmetta

i dati al master. La lunghezza delle word trasmesse può essere di 8, 16 o 32 bit.

- La polarità e la fase del clock dipendono dalle specifiche applicazioni. Il protocollo SPI definisce come bisogna collegare il master agli slave e lascia molte libertà al programmatore su come gestire il campionamento. Allora il campionamento può essere fatto sul fronte di salita o discesa del clock. In base allo slave bisognerà andare a vedere con quale configurazione funziona la periferica SPI. Questo è uno svantaggio dal punto di vista del programmatore.

Ci possono essere quattro modalità di funzionamento per la polarità, CPOL, e la fase, CPHA. Questo determina il modo in cui il master e lo slave campionano il segnale, cioè sul fronte positivo o negativo del clock. Le varie modalità sono:

1. CPOL = 0 e CPHA = 0. In questo caso lo slave campiona sul fronte positivo del clock mentre il master sul fronte negativo del clock.

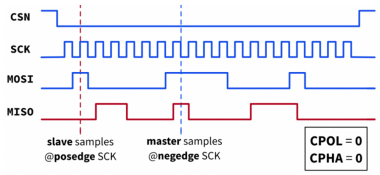

Figura 2.14: SPI CPOL = 0, CPHA = 0

2. CPOL = 0 e CPHA = 1. In questo caso lo slave campiona sul fronte negativo del clock mentre il master sul fronte positivo del clock.

3. CPOL = 1 e CPHA = 0. In questo caso lo slave campiona sul fronte negativo del clock mentre il master sul fronte positivo del clock.

4. CPOL = 1 e CPHA = 1. In questo caso lo slave campiona sul fronte positivo del clock mentre il master sul fronte negativo del clock.

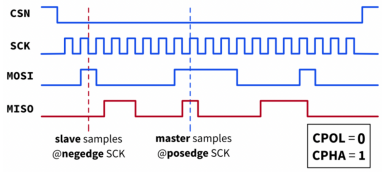

Figura 2.15: SPI CPOL = 0, CPHA = 1

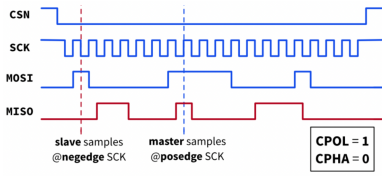

Figura 2.16: SPI CPOL = 1, CPHA = 0

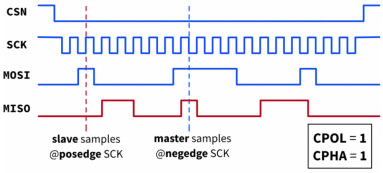

Figura 2.17: SPI CPOL = 1, CPHA = 1

I modi in cui è possibile collegare le periferiche al SPI, sono la modalità Daisy-Chain e la modalità indipendente. La differenza tra le due modalità sono nella velocità con cui si comunica con la periferica e il numero di fili. In entrambi i casi il clock viene sempre pilotato dal master e viene poi inviato agli slave.

Nella modalità Daisy-chain si ha un unico slave select, SS, che in questo caso è negato quindi quando è attivo è basso, e viene pilotato su tutti i dispositivi. Successivamente nella modalità Daisy-chain si ha una catena di collegamenti, data dal fatto che il MOSI del master va collegato al MOSI dello slave numero 1, poi il MISO dello slave numero 1 va collegato nel MOSI dello slave numero 2 e il MISO dello slave numero 2 collegato nel MOSI dello slave numero 3, infine il MISO dello slave numero 3 va nel MISO del master. In

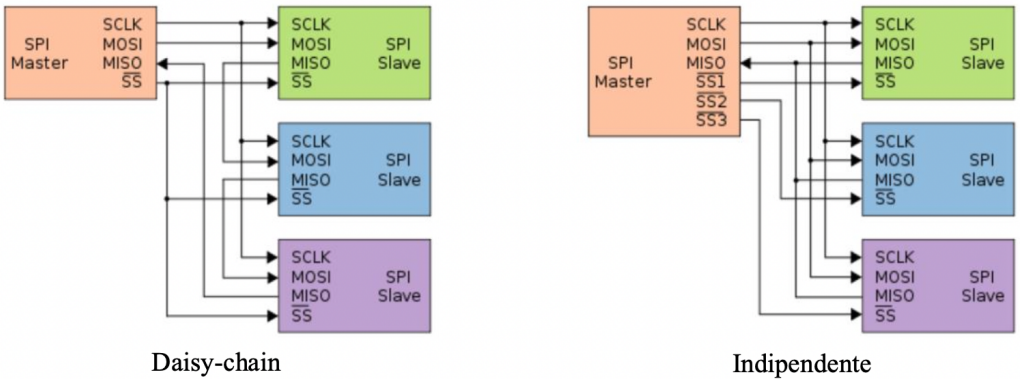

Figura 2.18: Modalità Daisy-Chain e Indipendente SPI

questo modo il master può accedere a tutti i registri interni di ogni slave. Questo permette di realizzare PCB compatte e poco costose, però introduce una elevata latenza.
La modalità indipendente è più efficiente dal punto di vista della comunicazione. In questo caso si ha uno slave select per ogni slave, però questo porta ad avere più pin sulla periferica di comunicazione SPI. In questo caso l'errore potrebbe essere quello di pilotare più slave contemporaneamente, dato che se si fa un errore nel firmware, ad esempio abilito due slave contemporaneamente, e abilito l'operazione di lettura attraverso il MOSI, allora sia lo slave 1 che lo slave 2 inviano dei dati e si possono creare dei conflitti di tipo elettrici. Il conflitto porta a non poter leggere nessun contenuto e può causare un consumo di potenza molto elevato.

Con riferimento alla Fig. 2.19, tipicamente la programmazione di un dispositivo che utilizza la periferica di comunicazione SPI avviene andando prima a inviare i segnali, che identificano la scrittura o lettura, poi l'indirizzo e infine sul segnale MOSI si invia il dato che si vuole scrivere, oppure sul segnale MISO il segnale da leggere, oppure entrambi dato che il dispositivo può funzionare parallelamente. Nel caso della periferica di comunicazione SPI del uDMA l'utilizzo del Full-Duplex non è disponibile. Il protocollo SPI

non è univoco ma dipende dalla periferica che si sta andando a utilizzare.

Figura 2.19: Sequenza comando di lettura da una memoria flash

2.4.2 Registri per SPI

Di seguito vengono riportati i registri per la periferica di comunicazione SPI Master.

- RX_SADDR, è il registro di configurazione dell'indirizzo di base del buffer RX SPIM nel uDMA.

- RX_SIZE, è il registro di configurazione della dimensione del buffer RX SPIM nel uDMA.

- RX_CFG, è il registro di configurazione dello stream del uDMA per RX SPIM.

- TX_SADDR, è il registro di configurazione dell'indirizzo di base del buffer TX SPIM nel uDMA.

- TX_SIZE, è il registro di configurazione della dimensione del buffer TX SPIM nel uDMA.

- TX_CFG, è il registro di configurazione dello stream del uDMA per TX SPIM.

2.4.3 Comandi del uDMA per SPI

Di seguito vengono riportati i comandi per poter programmare la periferica di comunicazione sul uDMA.

- SPI_CMD_CFG. Si utilizza per impostare la configurazione per SPI master.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è CFG.
  - Bit 9 - CPOL. Definisce il campo di bit del clock polarity per SPI.
  - Bit 8 - CPHA. Definisce il campo di bit del clock phase per SPI.
  - Bit 7:0 - CLKDIV. Definisce il campo di bit per il divisore di clock per SPI.
- SPI_CMD_SOT. Si utilizza per impostare il CS.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è SOT.
  - Bit 1:0 - CS. Definisce il campo di bit per la selezione del Chip Select per SPI da mettere a 0.
- SPI_CMD_SEND_CMD. Si utilizza per trasmettere un comando di dimensione configurabile.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è SEND_CMD.
  - Bit 27 - SPI_CMD. Definisce il campo di bit per l'utilizzo del QPI.
  - Bits 20:16 - CMD_SIZE. Definisce il campo di bit per la dimensione del comando da inviare. Il valore è (num bit – 1).
  - Bits 15:0 - CMD_VALUE. Definisce il campo di bit per il comando da inviare.

- SPI_CMD_SEND_ADDR. Si utilizza per trasmettere un indirizzo di dimensione configurabile.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è SEND_ADDR.
  - Bit 27 - SPI_CMD. Definisce il campo di bit per l'utilizzo del QPI.
  - Bits 20:16 - CMD_SIZE. Definisce il campo di bit per la dimensione del indirizzo da inviare. Il valore è (num bit – 1).
- SPI_CMD_DUMMY. Riceve un numero di bit dummy (non inviati all'interfaccia rx).
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è DUMMY.
  - Bits 20:16 - DUMMY_CYCLE. Definisce il campo di bit per il numero di cicli di dummy.
- SPI_CMD_WAIT. Aspetta un trigger esterno per andare alla prossima istruzione.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è WAIT.
  - Bits 1:0 - EVENT_ID. Definisce il campo di bit per l'evento esterno del uDMA.
- SPI_CMD_TX_DATA. Invio dei dati, massimo 64kbits.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è TX_DATA.
  - Bit 27 - SPI_CMD. Definisce il campo di bit per l'utilizzo del QPI.
  - Bits 26 - BYTE_ALIGN. Definisce il campo di bit per l'allineamento dei byte.

  - Bits 16:0 - DATA_SIZE. Definisce il campo di bit per la dimensione in bit dei dati da inviare. Il valore è (num bit – 1).

- SPI_CMD_RX_DATA. Ricezione dei dati, massimo 64 kbits.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è RX_DATA.
  - Bit 27 - SPI_CMD. Definisce il campo di bit per l'utilizzo del QPI.
  - Bits 26 - BYTE_ALIGN. Definisce il campo di bit per l'allineamento dei byte.
  - Bits 15:0 - DATA_SIZE. Definisce il campo di bit per la dimensione in bit dei dati da ricevere. Il valore è (num bit – 1).
- SPI_CMD_RPT. Ripete l'istruzione N volte.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è RPT.
  - Bits 15:0 - RPT_CNT. Definisce il campo di bit per il valore del contatore di ripetizione del trasferimento.
- SPI_CMD_EOT. Pulisce il CS.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è EOT.
  - Bits 1:0 - EVENT_GEN. Definisce il campo di bit per generare l'evento di EOT del uDMA.
- SPI_CMD_RPT_END. Conclude la ripetizione dei comandi.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è RPT_END.
- SPI_CMD_RX_CHECK. Controlla fino a 16 bit di dati rispetto a un valore previsto

  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è RX_CHECK.
  - Bit 27 - SPI_CMD. Definisce il campo di bit per l'utilizzo del QPI.
  - Bits 26 - BYTE_ALIGN. Definisce il campo di bit per l'allineamento dei byte.
  - Bits 25:24 - CHECK_TYPE. Definisce il campo di bit per il check mode. Seleziona il controllo da eseguire
    * 00: Confronta un bit a bit.
    * 01: Confronta solo gli 1.
    * 10: Confronta solo gli 0.
  - Bits 19:16 - STATUS_SIZE. Definisce il campo di bit per la dimensione dei dati da leggere, in bit. Il valore è (num bit – 1).
  - Bits 15:0 - COMP_DATA. Definisce il campo di bit per il valore di confronto.

- SPI_CMD_FULL_DUPLEX. Fa un trasferimento Full-Duplex.
  - Bits 31:28 - SPI_CMD. Definisce il comando SPIM da elaborare, che in questo caso è FULL_DUPLEX.
  - Bits 26 - BYTE_ALIGN. Definisce il campo di bit per l'allineamento dei byte.
  - Bits 15:0 - COMP_DATA. Definisce il campo di bit per la dimensione dei bit da inviare. Il valore è (num bit – 1).

Di seguito è mostrato un esempio sul utilizzo di questi comandi. Si suppone di voler utilizzare l'interfaccia SPIM0 per andare a scrivere su una memoria FLASH. La memoria Flash utilizzata in questo caso è la S25FS256S della Cypress.

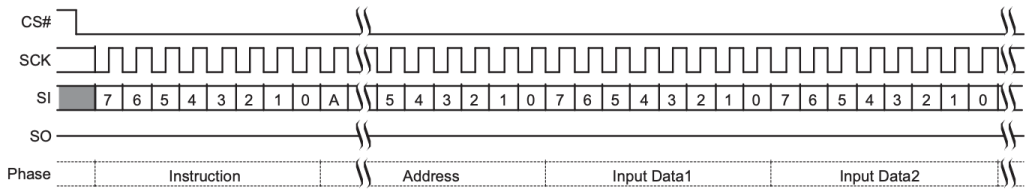

Figura 2.20: Page Program (PP 02h or 4PP 12h) Command Sequence

```
1
2  int tx_buffer_cmd_program[BUFFER_SIZE] =
3      {SPI_CMD_CFG(1, 0, 0),
4       SPI_CMD_SOT(0),
5       SPI_CMD_SEND_CMD(0x06, 8, 0),
6       SPI_CMD_SOT(0),
7       SPI_CMD_SEND_CMD(0x12, 8, 0),
8       SPI_CMD_TX_DATA(4, 0, 8, 0, 0),
9       SPI_CMD_TX_DATA(TEST_PAGE_SIZE, 0, 8, 0, 0),
10      SPI_CMD_EOT(0, 0)};
```

Figura 2.21: Comandi uDMA per SPI per scrivere una Flash

Da protocollo questa memoria vuole un flusso di dati come quello mostrato in Fig. 2.20. Il buffer di comandi che viene inviato, tramite il uDMA, all'interfaccia SPIM0 è mostrato in Fig. 2.21.
Il comando SPI_CMD_CFG(1,0,0), imposta nella prima posizione il valore del divisore di clock, mentre alla seconda e terza posizione si impostano il CPHA e il CPOL entrambi a 0. Di conseguenza lo slave campiona il segnale sul fronte positivo, mentre il master campiona il segnale sul fronte negativo del clock.
Il comando SPI_CMD_SOT(0) selezione il l'interfaccia SPI da utilizzare, in questo caso la 0.
Il comando SPI_CMD_SEND_CMD, serve invece per inviare un comando attraverso all'interfaccia SPIM0 alla memoria Flash. Questo si presenta due

volte, una volta compare il comando 0x06, che è il comando per dire alla Flash che si vuole fare una operazione di scrittura su di essa, e successivamente si ha il comando 0x12, che invece serve per impostare la pagina della Flash che si vuole scrivere. Il secondo termine invece, 8, indica la dimensione del comando in bit.
Il comando SPI_CMD_TX_DATA(4,0,8,0,0), serve invece per inviare i dati. Il numero 4 indica il numero di parole da inviare, lo 0 successivo indica che si vuole trasmettere una word per trasferimento , mentre 8 successivo è la dimensione di ogni word da inviare. Questo comando serve per inviare alla memoria Flash l'indirizzo della partizione di memoria che si vuole scrivere. Per l'invio del buffer di dati si utilizza invece il seguente comando SPI_CMD_TX_DATA(TEST_PAGE_SIZE,0,8,0,0).
Il comando SPI_CMD_EOT(0,0) serve per alzare il segnale del CS dell' interfaccia usata precedentemente.

2.5 Periferica I^2C in PULP

2.5.1 Protocollo I^2C

Il protocollo di comunicazione I^2C è stato introdotto da Philips nel 1982, e serve per comunicare con le periferiche a bassa banda, come ad esempio le memorie EEPROM, sensori di temperatura, oppure serve per controllare dispositivi esterni, come telecamere, microfoni, LCD, potenziometri digitali e convertitori A/D. Una delle caratteristiche principali del protocollo di comunicazione I^2C è quello di permettere di collegare sullo stesso bus un numero elevato di periferiche, ognuna individuata da un proprio indirizzo, e allo stesso tempo si possono avere più master. Di conseguenza si possono avere due microcontrollori connessi allo stesso sensore, dove però la lettura del sensore viene fatta uno alla volta, oppure avere un microcontrollore connesso a più sensori. Dal punto di vista del funzionamento ci sono tre modalità di funzionamento cioè:

- Slow, che permette di trasmettere dati fino a 100 Kbit/s;
- Fast, che permette di trasmettere dati fino a 400 Kbit/s;
- High-Speed, che permette di trasmettere dati fino a 3,4 Mbit/s;

La periferica di comunicazione I²C è composta da un bus seriale che necessita di sole due linee, cioè SDA (Serial Data) e SCL (Serial Clock) più la linea di massa. Tutte e due le linee sono bidirezionali. La prima è utilizzata per il transito dei dati, a 8 bit, mentre la seconda è utilizzata per trasmettere il segnale di clock necessario per la sincronizzazione della trasmissione.
La massima distanza definita dallo standard per la periferica di comunicazione I²C è di 3 metri. La velocità di funzionamento dell'interfaccia dipende dalla lunghezza dei "wire" collegata a questa interfaccia, perciò più è grande la distanza tra il master e lo slave collegati a questa interfaccia, e più piccola è la banda e la velocità con cui è possibile raggiungerli. Di solito però nei sistemi da noi considerati la distanza è molto bassa perché le periferiche vengono montate sulla stessa PCB.
Rispetto alla periferica di comunicazione SPI, nel I²C la comunicazione è di tipo Half-Duplex. Allora i dati possono transitare sia dal master allo slave che dallo slave al master, ma non possono transitare in parallelo, e quindi contemporaneamente.
L'interfaccia I²C, rispetto alle altre, utilizza dei driver di tipo Open-Drain. Quando si va a pilotare una linea con i transistori CMOS, si ha un transistore PMOS che serve per portare la linea a 1 e poi un transistore NMOS che serve per portare la linea a 0. I master e gli slave sono configurati in Pull-Down. La comunicazione avviene nel seguente modo. Se nessun transistori di Pull-Down collegato alla linea è acceso allora la linea viene portata dal segnale logico 1 a VDD attraverso la resistenza di Pull-Up. Se almeno uno dei transistori di Pull-Down è acceso allora la linea va a 0. Dalla Fig. 2.22 si vede che sia il master che lo slave possono portare a 0 o a 1 le linee e che sia il master che gli slave sono in grado di leggere i valori dalle linee mediante i due buffer. Allora man mano che aumentano il numero di transistor, buffer

e lunghezza dei fili, il carico capacitivo legato alle due linee aumenta e di conseguenza la banda diminuisce.

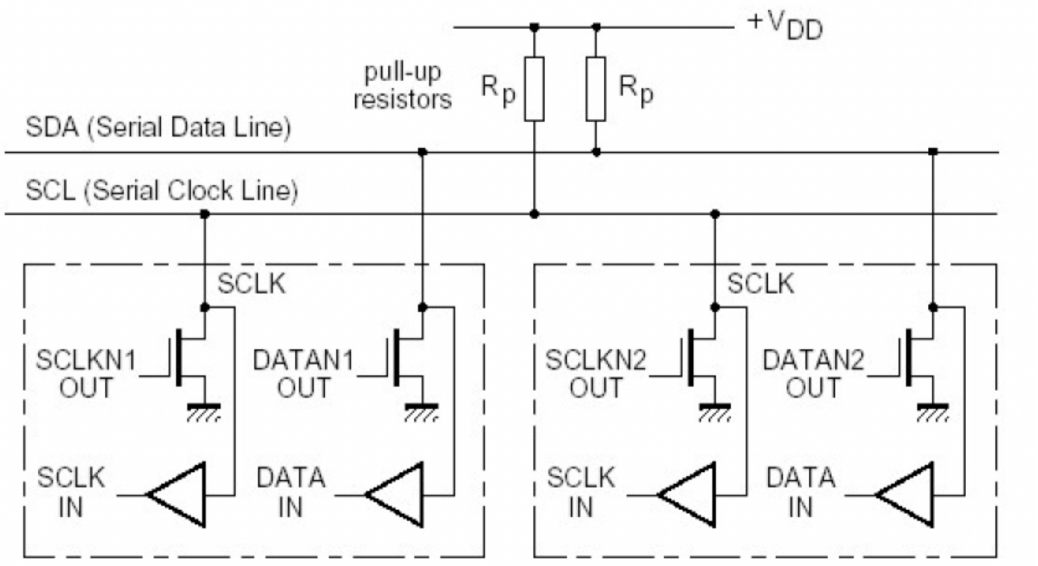

Figura 2.22: Driver Open-Drain

Di seguito viene mostrato come funziona il protocollo del I^2C. Con rife-

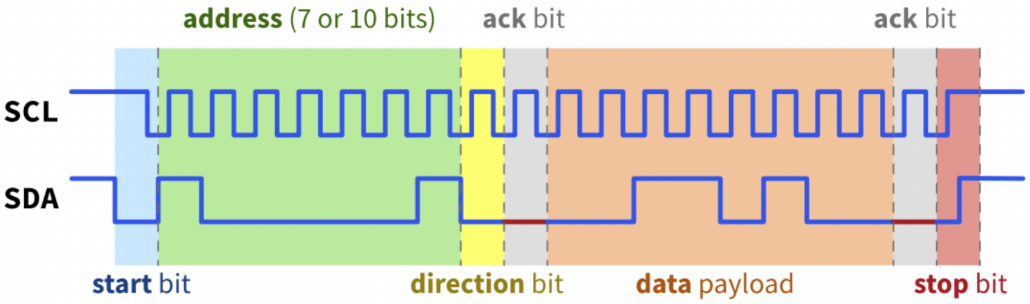

Figura 2.23: Protocollo I2C

rimento alla Fig. 2.23, si ha che inizialmente sul bus non è presente nessuna trasmissione, quindi le linee SDA e SCL sono a 1. Per far partire una trasmissione bisogna che il segnale di SDA vada a 0, mentre il segnale SCL

continui a essere a 1. A questo punto tutti gli slave presenti sul bus si mettono in ascolto. Per collegarsi a uno slave preciso bisogna che il master invii l'indirizzo dello slave a ci cui si vuole collegare attraverso il bus. L'indirizzo può andare da 7 a 10 bit. Come si vede dalla Fig. 2.23, l'indirizzo che si invia è b1000001. Dopo aver inviato l'indirizzo dello slave a cui si vuole collegare, il master invia un direction bit, con il quale dice se vuole effettuare un operazione di scrittura o lettura. Nel caso in cui SDA è 0 allora si vuole fare una operazione di scrittura, mentre quando SDA è 1 allora si vuole fare un operazione di lettura. Nel caso in esempio l'operazione è di scrittura. Dopo che il master ha inviato l'indirizzo e il direction bit, lo slave invia un segnale di acknowledges. Nel caso in cui non ci sia nessuno slave collegato al bus con l'indirizzo inviato dal master, allora il segnale SDA rimane a 1, e di conseguenza il segnale di acknowledges è 0. In questo caso non si è riusciti a collegarsi allo slave. Nel caso in cui, invece, ci sia uno slave collegato al bus che ha lo stesso indirizzo inviato dal master, allora il segnale SDA va a 0, e di conseguenza il segnale di acknowledges è 1. In questo caso si è riusciti a collegarsi allo slave. Una volta verificato che il bit di acknowledges è 1, allora si passa all'invio del dato allo slave. Come si vede dalla Fig. 2.23 il dato inviato in bit è b00110100. Una volta finito il trasferimento, lo slave dichiara che il trasferimento è avvenuto con un bit di acknowledges, che viene portato a 0. Nel caso in cui invece di scrivere si fosse andato a leggere dallo slave, allora sarebbe stato il master a inviare il bit di acknowledges. Alla fine del trasferimento si ha uno stop bit. Allora SDA viene portato a 0 e SCL viene rilasciato a 1 e successivamente SDA viene portato a 1.
Una caratteristica della periferica di comunicazione I²C è il Clock Stretching. Questo si utilizza quando ci si connette con una periferica vecchia. Allora se il master sta fornendo allo slave un clock troppo veloce, lo slave può portare a 0 il clock per un periodo ad esso necessario. In questo modo lo slave ha il tempo necessario per rispondere al master.

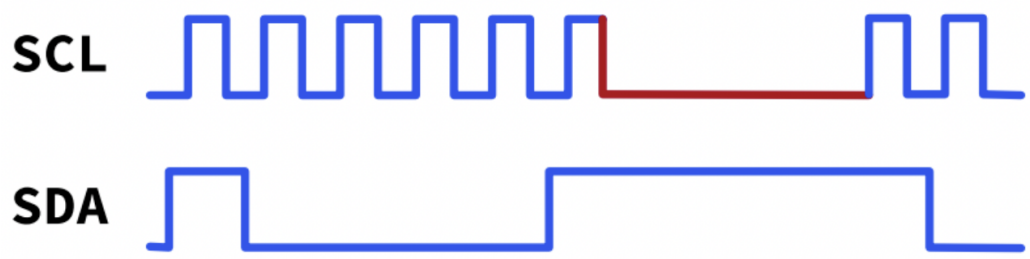

Figura 2.24: Clock Stretching

2.5.2 Registri per I²C

Di seguito vengono riportati i registri per la periferica di comunicazione I²C.

- RX_SADDR, è il registro di configurazione dell'indirizzo di base del buffer RX I²C nel uDMA.
- RX_SIZE, è il registro di configurazione della dimensione del buffer RX I²C nel uDMA.
- RX_CFG, è il registro di configurazione dello stream del uDMA per RX I²C.
- TX_SADDR, è il registro di configurazione dell'indirizzo di base del buffer TX I²C nel uDMA.
- TX_SIZE, è il registro di configurazione della dimensione del buffer TX I²C nel uDMA.
- TX_CFG, è il registro di configurazione dello stream del uDMA per TX I²C.
- STATUS, è il registro di stato per I²C nel uDMA.
- SETUP, è il registro di configurazione per I²C nel uDMA.

2.5.3 Comandi del uDMA per I²C

Di seguito vengono riportati i comandi per poter programmare la periferica di comunicazione sul uDMA.

- I2C_CMD_START. Comando di inizio trasferimento.
- I2C_CMD_STOP. Comando di fine trasferimento.
- I2C_CMD_RD_ACK. Comando per ricevere i dati e inviare acknowledge.
- I2C_CMD_RD_NACK. Comando per ricevere i dati e non inviare acknowledge.
- I2C_CMD_WR. Comando per inviare i dati e aspettare il comando acknowledge.
- I2C_CMD_WRB. Comando per inviare un buffer di dati e aspettare il comando acknowledge.
- I2C_CMD_WAIT. Comando per aspettare un certo numero di cicli.
- I2C_CMD_RPT. Comando per ripetere il comando successivo.
- I2C_CMD_CFG. Comando di configurazione.
- I2C_CMD_WAIT_EV. Comando per aspettare un evento esterno del uDMA.
- I2C_CMD_EOT. Comando per ricevere il segnale di end of transfer.

Di seguito è mostrato un esempio sul utilizzo di questi comandi. Si suppone di voler utilizzare l'interfaccia I2C0 per andare a scrivere su una memoria EEPROM.

La memoria EEPROM utilizzata in questo caso è la 24FC1025 della Microchip. Da protocollo questa memoria vuole un flusso di dati come quello mostrato in Fig. 2.25. Il buffer di comandi che viene inviato, tramite il

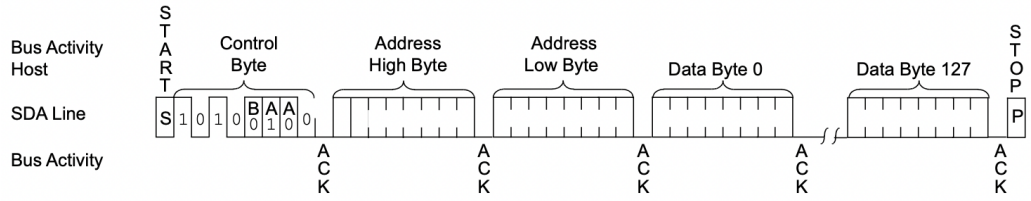

Figura 2.25: Page write eeprom

```
volatile uint32_t cmd_buffer_wr[BUFFER_SIZE] = {
  (((uint32_t)I2C_CMD_CFG) << 24) | 0x40,
  (((uint32_t)I2C_CMD_START)<<24),
  (((uint32_t)I2C_CMD_WRB)<<24) | 0xa0, //Control->write
  (((uint32_t)I2C_CMD_WRB)<<24), //Addr MSB
  (((uint32_t)I2C_CMD_WRB)<<24), //ADDR LSB
  (((uint32_t)I2C_CMD_WRB)<<24) | expected_rx_buffer[0], //DATA0
  (((uint32_t)I2C_CMD_WRB)<<24) | expected_rx_buffer[1], //DATA1
  (((uint32_t)I2C_CMD_WRB)<<24) | expected_rx_buffer[2], //DATA2
  (((uint32_t)I2C_CMD_WRB)<<24) | expected_rx_buffer[3], //DATA3
  (((uint32_t)I2C_CMD_STOP)<<24)
};
```

Figura 2.26: Comandi uDMA per I2C per scrivere una EEPROM

uDMA, all'interfaccia I2C0, per scrivere su questa memoria EEPROM, è mostrato nel codice di Fig. 2.26.
L'interfaccia del uDMA accetta buffer di comandi a 32 bit, mentre i comandi del uDMA per I^2C sono a 8 bit. Allora per inviare correttamente i comandi al uDMA per poter utilizzare I^2C si è utilizzato il metodo mostrato in Fig. 2.26. Quello che si è fatto è prendere i comandi a 8 bit e fare uno shift di 24 posizioni a sinistra, in questo modo anche se si invia un buffer di 32 bit solo le prime 8 posizioni di questo hanno l'informazione sul comando. Come si può vedere, il buffer di comandi inizialemnte invia i comandi di configurazione e di start. Successivamente si invia l'indirizzo della EEPROM, 0x50, e si ag-

giunge il direction bit, dato che l'operazione è di scrittura questo deve essere 0, quindi il comando da inviare sarà 0xa0. Successivamente invio l'indirizzo della partizione di memoria dove si vuole scrivere i dati. Infine si inviano i dati e lo stop bit.

2.6 Interrupt

Ci sono due modi in cui una periferica può comunicare il proprio stato ad un processore.
Il primo modo prende il nome di metodo a polling. Nel metodo a polling il processore continua a interrogare la periferica, tipicamente fino a quando questa con cambia il suo stato. Il vantaggio del metodo a polling è che non richiede logica aggiuntiva, lo svantaggio invece è che il processore utilizza cicli di clock per interrogare la periferiche anche se questa non richiede nessun servizio da parte sua.
Il secondo metodo invece prevede l'utilizzo degli interrupt. In questo caso quando la periferica vuole comunicare con il processore, questa invia un segnale hardware che prende il nome di interrupt request. Il vantaggio in questo caso è che non si sprecano più cicli di clock per interrogare la periferica, ma questa volta sarà la periferica a inviare un segnale al processore quando avrà bisogno di un suo servizio. Lo svantaggio è però l'utilizzo di logica aggiuntiva.

Affinchè il processore possa gestire le richieste di interrupt da parte delle periferiche, bisogna che questo identifichi il tipo di interrupt e determini quale interrupt gestire prima. Allora quello che succede quando la periferica invia un interrupt request è che il processore prima completa l'istruzione corrente, poi segnala alla periferiche di aver ricevuto la sua richiesta di interrupt, questo grazie a un segnale di acknowledge, ACK, e successivamente esegue Interrupt Service Routine, ISR, relativa a quel interrupt request.

Interrupt Service Routine è una sequenza di codice che viene eseguita dal

processore, quando questo deve gestire un interrupt, e si divide in tre parti. Per prima cosa salva i registri del MCU nella memoria stack, poi esegue il codice per gestire l'interrupt e successivamente ripristina il valore dei registri salvati nello stack e passa all'istruzione successiva.

Tra la periferica che invia interrupt request e il processore, che invece deve gestire interrupt, si ha Interrupt Controller. Questo serve per dare priorità e indirizzare gli interrupt del SoC.

Nel caso in cui ci siano richieste multiple di interrupt, allora il sistema deve identificare quale periferica ha generato interrupt e risolvere le richieste simultanee attravero una logica di priorità. In genere possono essere utilizzate delle maschere per abilitare alcuni interrupt e disabilitarne altri. Uno schema comunemente usato, per gestire più richieste di interrupt, consiste nell'allocare permanentemente un'area nella memoria per contenere gli indirizzi delle Interrupt Service Routine. Questi indirizzi sono generalmente indicati come interrupt vectors e si dice che costituiscano la interrupt-vector table. Quando arriva una richiesta di interrupt da una periferica, le informazioni fornite dalla periferica vengono utilizzate come puntatore nella interrupt-vector table e l'indirizzo viene automaticamente caricato nel contatore del programma.

PULP supporta fino a 32 linee di interrupt. Ad esempio la linea di interrupt numero 26, cioè quella legata al SoC, consente di generare interrupt per le periferiche di comunicazione SPI e I2C. L'interrupt controller contiene una FIFO dove vengono inseriti gli eventi provenienti dalle periferiche o eventi SW. Quando un interrupt è pronto ed è abilitato, l'event unit lo codifica con un ID a 5 bit. Se il core accetta l'interrupt, risponde con l'ID dell'interrupt preso e il segnale di acknowledge. La comunicazione tra l'interrupt controller ed il core è completamente asincrona. Si noti che interrupt controller può modificare l'ID del interrupt in qualsiasi momento, ma deve fare affidamento sul ID inviato dal core per sapere quale interrupt è stato preso. Questa è

una caratteristica importante che copre la situazione in cui una richiesta di interrupt con priorità più alta ne impedisce un'altra che è già stata inviata al core.
Gli eventi provenienti dalle periferiche o da altre fonti, possono essere inoltrati al fabric controller, al cluster o a determinate periferiche. Il SoC Event Generator deve poi controllare quali eventi devono essere inoltrati e dove.

Capitolo 3

Sviluppo dei sistemi RTOS sulla piattaforma PULP

3.1 Introduzione ai sistemi RTOS

In un Sistema Embedded Real-Time, una componente critica per le prestazioni del sistema è il tempo necessario per rispondere a un input. Questo perchè se il sistema produce una risposta corretta, in corrispondenza di un certo input, ma questa non viene generata entro un certo limite di tempo, si possono avere delle anomalie più o meno gravi nel sistema. Allora la risposta di un sistema embedded Real-Time deve essere sia corretta che tempestiva. Nei sistemi embedded non Real-Time quello che si fa è realizzare del firmware, senza però preoccuparsi dei limiti di tempo neccessari per la sua esecuzione ma bensì concentrandosi sulla correttezza dei risultati che questo produce. Le applicazioni in real-time possono essere suddivise in due categorie: soft real-time e hard real-time. Nei sistemi soft real-time è altamente desiderato ma non assolutamente necessario soddisfare tutti i requisiti di temporizzazione del sistema. In questo caso oltrepassare questo vincolo non produce danni irreparabili al sistema. Nei sistemi hard real-time bisogna in ogni circostanza, soddisfare rigorosamente tutti i requisiti di temporizzazione del sistema. In questo caso oltrepassare questo vincolo produce danni irreparabili al sistema.

La maggior parte dei sistemi embedded si basa sull'uso di interrupt per notificare al processore quando un'operazione è richiesta da una certa periferica. In un'applicazione real-time, la gestione degli interrupt può diventare un fattore critico per garantire il corretto funzionamento del sistema. Ogni volta che avviene un interrupt, si ha che il codice principale viene sospeso per gestire l'interrupt. Ciò significa che quando il codice sospeso riprende l'esecuzione, avrà meno tempo per il completamento. Come regola generale, è meglio ridurre al minimo la quantità di tempo dedicata alla gestione degli interrupt.
Alcuni dispositivi, come ad esempio una memoria flash, potrebbero richiedere un periodo di tempo considerevole per completare un'operazione di lettura o scrittura. Quando si lavora con questi dispositivi, non è accettabile che il processore si fermi e attenda semplicemente il completamento dell'operazione, dato che potrebbe non rispettare i limiti temporali. Allora sono necessarie tecniche più sofisticate quando si lavora con questi tipi di dispositivi. In genere un sistema embedded deve eseguire più task contemporaneamente.

3.1.1 Elementi chiave di un RTOS

Processo

Di seguito viene mostrato quali sono le fasi, Fig. 3.1 per eseguire un file in C.

- Editing. Questa è la fase in cui viene scritto il programma.
- Precompilazione. In questa fase avviene inclusione di file all'interno del file che si vuole eseguire. Ad esempio si include il contenuto di un determinato file, solitamente chiamato header file, con suffisso ".h". Successivamente si rimuovono i commenti e si sostituiscono le macro, #define, con le loro definizioni.
- Compilatore. Il compilatore C traduce il codice sorgente ricevuto dal preprocessore in codice assembly, cioè linguaggio macchina.

- L' assembler. Questo crea il codice oggetto, in UNIX i file con il suffisso .o sono i file in codice oggetto.
- Linker. In genere i programmi in C contengono funzioni definite in librerie standard oppure in file privati del programmatore. Il file oggetto creato dal compilatore in C contiene dei "buchi" dovuti a queste funzioni mancanti. Allora il linker collega il codice oggetto con quello delle funzioni mancanti.
- Loader. Il programma viene caricato, prima di essere eseguito, nella memoria centrale.
- Esecuzione. Il programma viene eseguito un'istruzione alla volta.

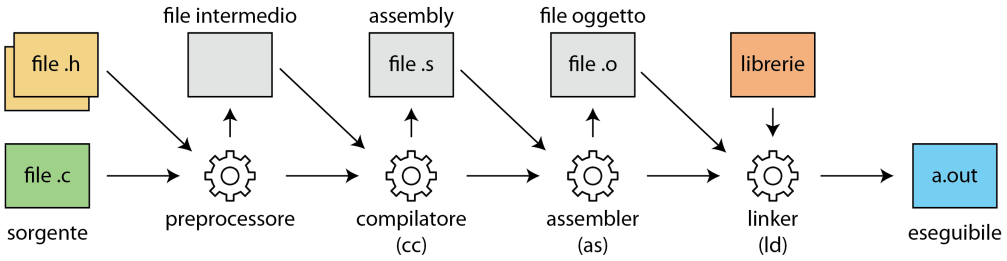

Figura 3.1: Fasi per eseguire un file in C

In generale un processo, o task, è un programma in esecuzione. Un task contiene, oltre le istruzioni da eseguire, anche l'insieme dei dati associati all'esecuzione.
In un computer a single core, è possibile eseguire solo un task alla volta. Quando avviene un scheduling event si va a selezionare il successivo task da eseguire oppure avviare o riprendere l'esecuzione di un task. Gli scheduling event si hanno quando si verificano eventi prodotti dai timer, quando si verificano system-call prodotte dal codice dell'applicazione e dalle Interrupt Service Routines.
I task nei sistemi embedded si trovano solitamente in uno dei tre stati seguenti:

- Ready. Il task è pronto per l'esecuzione ed aspetta di essere assegnato a un processore.
- Running. Il task è stato assegnato a un processore ed è in esecuzione.
- Block. Il task aspetta che un evento venga preso.

Ad ogni task viene assegnata una priorità dallo sviluppatore del sistema. Ogni volta che si verifica un scheduling event, il sistema identifica il task con la priorità più alta che si trova nello stato di Ready o Running. Di conseguenza se il task è nello stato di Ready allora questo passa allo stato di Running e quindi viene eseguito. Se invece il task è nello stato di Running , questo continua la sua esecuzione. Il passaggio da un task ad un altro implica la memorizzazione delle informazioni di contesto, principalmente il contenuto del registro del processore, associate al task. Queste informazioni sono presenti nel Task Control Block (TCB). Ogni context-switch richiede una piccola quantità di tempo, che viene sottratta dal tempo disponibile per l'esecuzione del task.

Di seguito vengono riportati alcuni meccanismi per la gestione dell'accesso a una risorsa condivisa tra i vari task.

Sincronizzazione dei processi

In un sistema embedded real-time è frequente che si utilizzino dei processi cooperanti. Un processo cooperante è un processo che può influenzare un processo in esecuzione e subirne l'influenza. Allora i processi cooperanti utilizzano uno stesso spazio di indirizzi o di dati, quindi delle risorse condivise. A causa di questo nascono le Race Condition. La Race Condition è la situazione che si crea quando più task vogliono andare a modificare una risorsa condivisa. Questo può causare situazioni di incoerenza degli stessi dati. Ecco perchè devono essere usati meccanismi come i Mutex o i Semafori per assicurare la corretta esecuzione dei task.

Un mutex, mutual exclusion, è un meccanismo per gestire l'accesso a una risorsa condivisa tra i task. Un mutex è concettualmente identico a una variabile globale che può essere letta e scritta da tutti i task. La variabile ha il valore 1 quando la risorsa condivisa è libera mentre e 0 quando è utilizzata da un task. Quando un task ha bisogno di accedere ad una risorsa condivisa, legge la variabile del mutex, se questa ha il valore 1, la imposta su 0 per indicare che la risorsa è stata occupata da questo task. Il task sarà quindi libero di interagire con la risorsa. Quando l'interazione è completa, il task imposta la varibile del mutex su 1, rilasciando così la proprietà.
Se un task, ad esempio il task_1, tenta di assumere la proprietà del mutex, mentre il mutex è detenuto da un'altro task, ad esempio task_2, allora il task_1 verrà bloccato fino a quando la proprietà non verrà rilasciata dal task_2. Ciò rimane vero anche se il task che contiene il mutex ha una priorità inferiore rispetto al task che lo richiede.

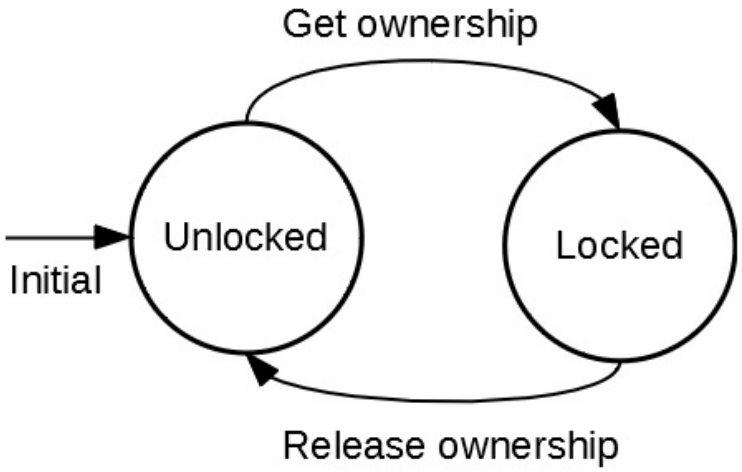

Figura 3.2: Mutex states

Il principale svantaggio del mutex è che richiede una attesa attiva (busy waiting). Mentre un task sta utilizzando la risorsa condivisa, ogni altro task che cerca di accedere alla risorsa condivisa deve continuare a leggere la va-

riabile del mutex. Questo tipo di mutex è anche chiamato spinlock, perché il processo continua a "girare" (spin), in attesa che il mutex diventi disponibile. Questo continuo girare è chiaramente un problema in un sistema multi task, dove una singola CPU è condivisa tra diversi task. L'attesa attiva spreca cicli di CPU che qualche altro task potrebbe utilizzare in modo produttivo. Tuttavia, gli spinlock hanno il vantaggio di non rendere necessario alcun cambio di contesto, operazione che può richiedere molto tempo.

Il semaforo è una generalizzazione del mutex, e serve per accedere alle risorse condivise. Il semaforo può essere di due tipi, cioè un semaforo binario o un semaforo contatore.
Un semaforo binario funziona in modo simile a un mutex. Se un task, esempio task_1, tenta di acquisire un semaforo mentre è utilizzato da un'altro task, esempio task_2, allora il task_1 invece di applicare l'attesa infinita, che era il principale problema dei mutex, si bloccherà fino a quando il task_2, in possesso del semaforo, non lo rilascerà. L'operazione di bloccaggio pone il task in una coda d'attesa associata al semaforo e lo stato del task cambia in waiting. Quando il task_2 rilascerà il semaforo binario, allora il task bloccato, task_1, cambierà il suo stato da waiting a ready ed entrerà nella coda dei trasferimenti pronti. Ognivolta che il processore passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella ready queue. In particolare sarà lo scheduler del processore che, tra i processi in memoria nello stato di ready, sceglierà quello da assegnare al processore.
Un semaforo a contatore contiene un contatore inizializzato a un limite superiore, che rappresenta il numero delle risorse disponibili. Ogni volta che il task prende un semaforo a contatore, il contatore diminuisce di uno. Quando il contatore raggiunge lo zero, i tentativi di prendere il semaforo si bloccheranno finché almeno un detentore del semaforo non lo rilascerà, il che incrementerà il contatore.
La lista dei processi che attendono a un semaforo si può facilmente realizzare

con una coda FIFO, della quale il semaforo contiene i puntatori al primo e all'ultimo elemento, oppure con una coda LIFO. In generale si può usare qualsiasi criterio d'accodamento, il corretto uso dei semafori non dipende dal particolare criterio adottato. Tuttavià la scelta di una coda FIFO o LIFO può portare al problema dell'attesa indefinita. Con questo termine si definisce una situazione d'attesa indefinita nella coda di un semaforo, che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio LIFO.

Allocazione dinamica della memoria

I sistemi RTOS generalmente forniscono meccanismi per allocare e rilasciare blocchi di memoria. In genere il sistema utilizza dati in ingresso provenienti da sensori, flussi di dati audio o video oppure strutture dati, dove mantiene le proprietà di una periferica in esecuzione. Per alcune applicazioni real-time, ha senso salvare questi dati utilizzando l'allocazione dinamica della memoria. Vi sono, tuttavia, alcuni problemi ben noti che possono sorgere nelle applicazioni che utilizzano l'allocazione dinamica della memoria. Il linguaggio C è ampiamente utilizzato nello sviluppo di sistemi embedded. Questo linguaggio di programmazione non fornisce l'allocazione e la deallocazione automatica della memoria, poiché gli oggetti e le strutture dati vengono creati e distrutti attraverso istruzioni specifiche nel firmware. Spetta allo sviluppatore del sistema garantire che l'allocazione e la deallocazione della memoria avvenga in modo corretto, efficiente e affidabile. Gli altri due problemi principali nell'utilizzo dell'allocazione dinamica della memoria sono overflow dell'heap e heap fragmentation.
Vediamo in cosa consiste overflow dell'heap. L'area di memoria utilizzata per l'allocazione dinamica è denominata heap. Se la memoria allocata non viene rilasciata dopo il suo uso, lo spazio di memoria nel heap si esaurirà presto. Se l'operazione di liberazione di ciascun blocco di memoria dopo l'uso viene erroneamente esclusa dal codice o ignorata per qualche motivo, o se i

blocchi di memoria vengono conservati per un tempo così lungo da ridurre la memoria disponibile a zero, si verificherà un overflow dell'heap. In questa situazione, ulteriori tentativi di allocare memoria non andranno a buon fine. Il problema del overflow dell'heap in un sistema embedded può avere risvolti molto importanti. A seconda della particolare architettura hardware del processore, è possibile leggere e scrivere vari indirizzi di memoria, tra cui importanti registri del processore, e a causa della scrittura di dati arbitrari su questi registri è probabile che il sistema smetta di funzionare correttamente. Poiché questo tipo di errore si verifica solo dopo che il sistema è stato in esecuzione per un tempo prolungato, potrebbe essere molto difficile identificare ed eseguire il debug dell'origine del problema.
Vediamo in cosa consiste heap fragmentation. Se si verificano frequenti allocazioni di memoria, anche se non si verifica un overflow dell'heap, è possibile, e anche probabile, che l'area di memoria gestita venga frammentata in blocchi di varie dimensioni. Quando ciò accade, l'allocazione di memoria per un blocco di grandi dimensioni, potrebbe non essere immediatamente possibile anche se è disponibile molta memoria libera. In uno scenario in cui la memoria è altamente frammentata, si può avere il mancato rispetto delle scadenze temporali del sistema. Bug come questo (ed è un bug, anche se il sistema alla fine funziona in modo funzionalmente corretto) potrebbero verificarsi raramente, con gravi effetti sul comportamento del sistema e sono spesso difficili da replicare in un ambiente di debug.

3.2 Gestione dei Task sulla piattaforma PULP

In PULP vengono utilizzati due tipo di RTOS, cioè FreeRTOS e PULP-OS, dove la loro descrizione è presente nei capitoli successivi. Di seguito si vuole esporre quali sono le funzioni che permettono la gestione dei task nei Firmware scritti per PULP. L'implementazione di queste API cambia a seconda che si utilizzi FreeRTOS o PULP-OS.

Con la funzione pi_task_block si va a inizializzare un evento di notifica

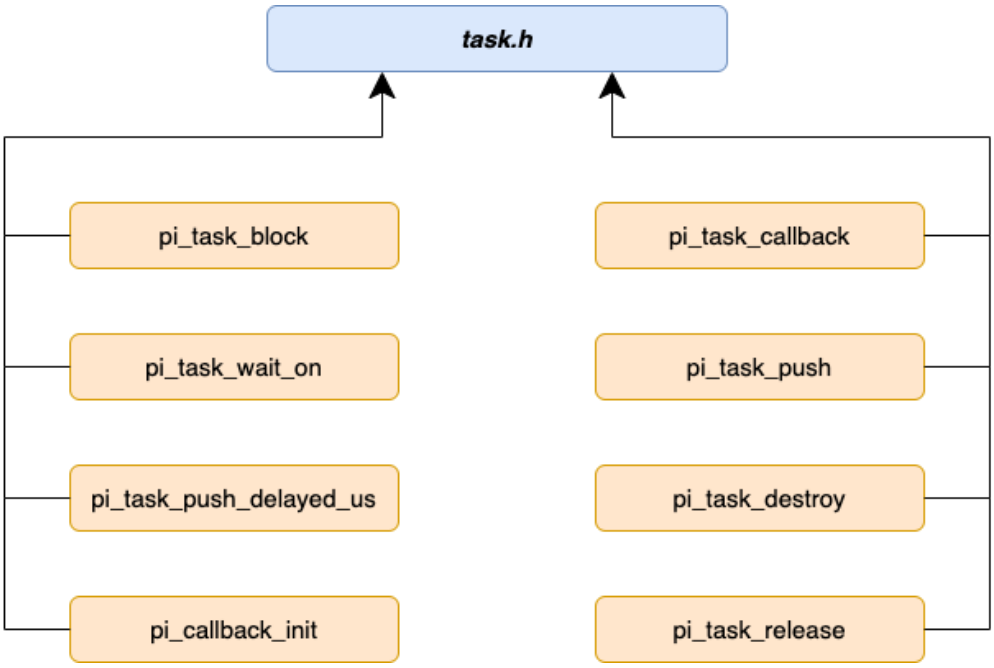

Figura 3.3: PMSIS API per gestire i task

in modo che sia pronto per essere attivato. Un evento di notifica può essere utilizzato per bloccare il chiamante, cioè chi chiama la funzione, usando pi_task_wait_on, fino a quando non si verifica una determinata azione, ad esempio la fine del trasferimento.
Con la funzione pi_task_callback si va a inizializzare una notifica di callback in modo che sia pronta per essere attivata. Un notifica di callback può essere utilizzata per attivare l'esecuzione di una funzione quando si verifica una determinata azione, ad esempio la fine di un trasferimento.
La funzione pi_task_wait_on si utilizza per bloccare il chiamante, cioè chi chiama la funzione, fino a quando l'evento di notifica, creato con pi_task_block, non è stato attivato.
La funzione pi_task_push si utilizza per attivare la notifica specificata. Se la notifica è una callback, verrà fatto lo scheduling per l'esecuzione della callback. Se la notifica è un evento, questo attiverà l'evento.
La funzione pi_task_push_delayed_us si utilizza per attivare una notifica

dopo un certo ritardo, espresso in microsecondi. Se la notifica è una callback, verrà fatto lo scheduling per l'esecuzione della callback. Se la notifica è un evento, questo attiverà l'evento.
La funzione pi_task_destroy si utilizza per cancellare un task, che è stato eseguito.
La funzione pi_task_release serve per fare il release di un task.

La struttura che descrive i task in PULP è mostrata in Fig. 3.4.

```
typedef struct pi_task {
        uintptr_t arg[4];
        int32_t id;
        uint32_t data[PI_TASK_IMPLEM_NB_DATA];
        pi_sem_t wait_on;
        struct pi_task *next;
        volatile int8_t done;
        int8_t core_id;
        int8_t cluster_id;
        PI_TASK_IMPLEM;
} pi_task_t;
```

Figura 3.4: Struttura per i task

3.3 Applicazione di FreeRTOS sulla piattaforma PULP

Introduzione

FreeRTOS è un RTOS a microkernel gratuito, sviluppato da Real Time Engineers Ltd. Un microkernel contiene una quantità minima di codice che implementa le funzionalità di base di un RTOS, inclusa la gestione dei task e la comunicazione tra i vari task. FreeRTOS offre diverse opzioni per la

gestione dinamica della memoria, supporta 35 diverse piattaforme di microcontroller ed è scritto nel linguaggio C con alcune funzioni del linguaggio assembly. FreeRTOS viene utilizzato per applicazioni real-time che utilizzano microcontrollori o piccoli microprocessori. Questo tipo di applicazione normalmente includono un mix di requisiti in real-time sia hard che soft. Il microkernel di FreeRTOS sviluppa, tra le varie cose, anche i mutex, semafori a contatore e semafori binari. FreeRTOS è stato usato per lo sviluppo di driver per le applicazioni in tempo reale su sistemi basati su PULP.

Gestione dei Task in FreeRTOS

Di seguito viene descritto come vengono implementare, Fig. 3.5, le dichiarazioni della Fig. 3.3.
Con la funzione __pi_task_block si va a inizializzare un'istanza di event task. Verrà inizializzato un semaforo e i vari parametri della struttura che contiene le informazioni del task.
Con la funzione __pi_task_callback si va a inizializzare un'istanza di event task. Questa event task esegue la callback fornita in argomento.
La funzione __pi_task_wait_on permette di aspettare un event task fino a quando il task non viene rilasciato. Per fare questo viene associato ad un task un semaforo a contatore.
La funzione __pi_task_push è usata per mettere un event task in un event kernel.
La funzione __pi_task_destroy va a rimuovere un event task. Questa funzione andrà a cancellare un semaforo, se questo è stato allocato.
La funzione pi_task_release va a rilasciare il semaforo a contatore precedentemente occupato dal task. Successivamente va a rilasciare il task.

Quando viene associato un task ad una funzione che gestisce il trasferimento di dati, in modo sincrono, Fig. 3.6, quello che succede è che inizialmente viene chiamata la funzione pi_task_block, che va a inizializzare la struttura che descrive il task e il semaforo a contatore. Dopodiché si ha il trasferimento dei dati, questo attraverso la periferica di comunicazio-

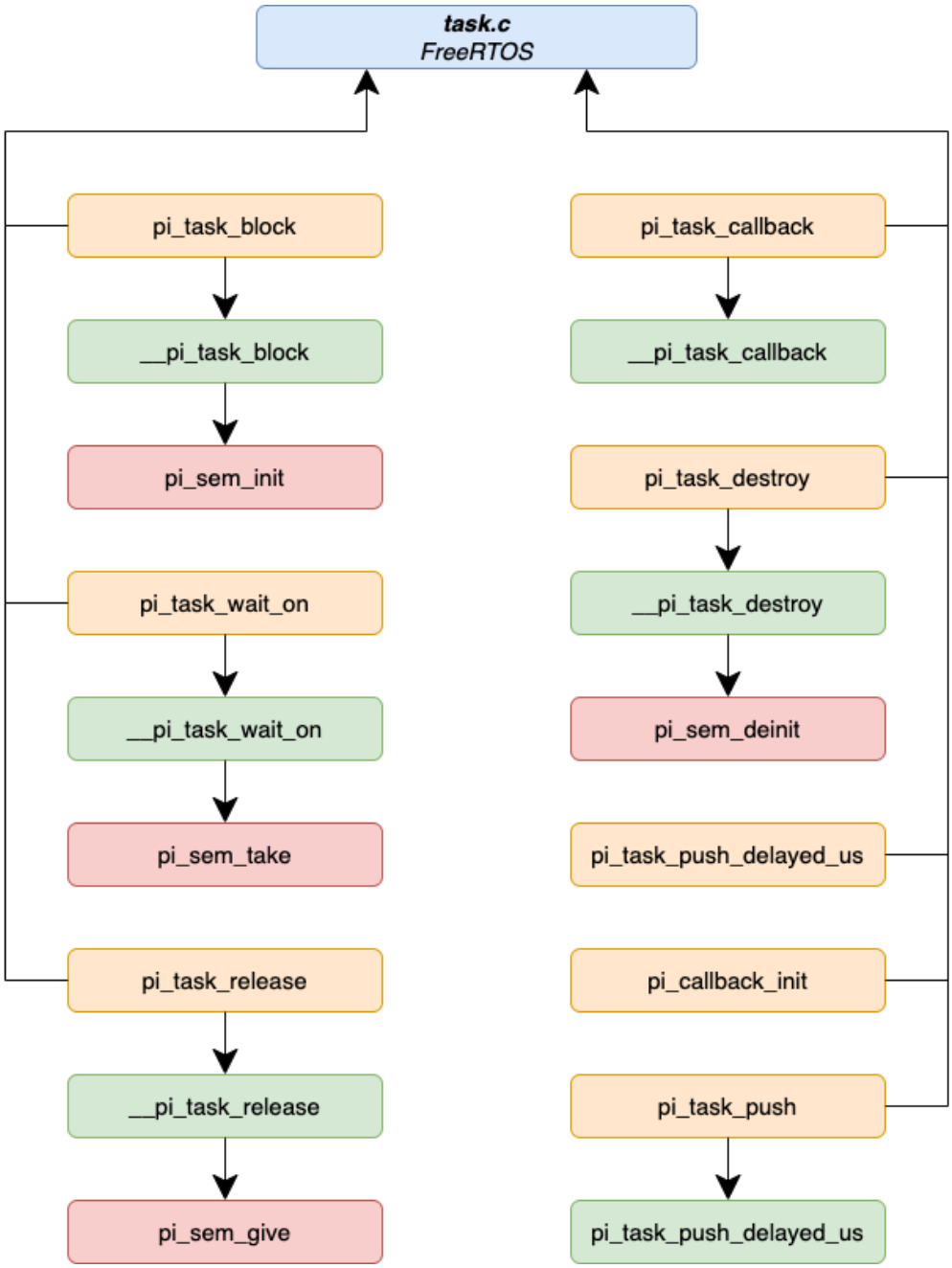

Figura 3.5: Gestione Task per trasferimento sincrono in FreeRTOS

ne SPI o I^2C. Una volta chiamata questa funzione, il Firmware si porta in pi_task_wait_on. In questa proceduta viene preso un semaforo a contatore,

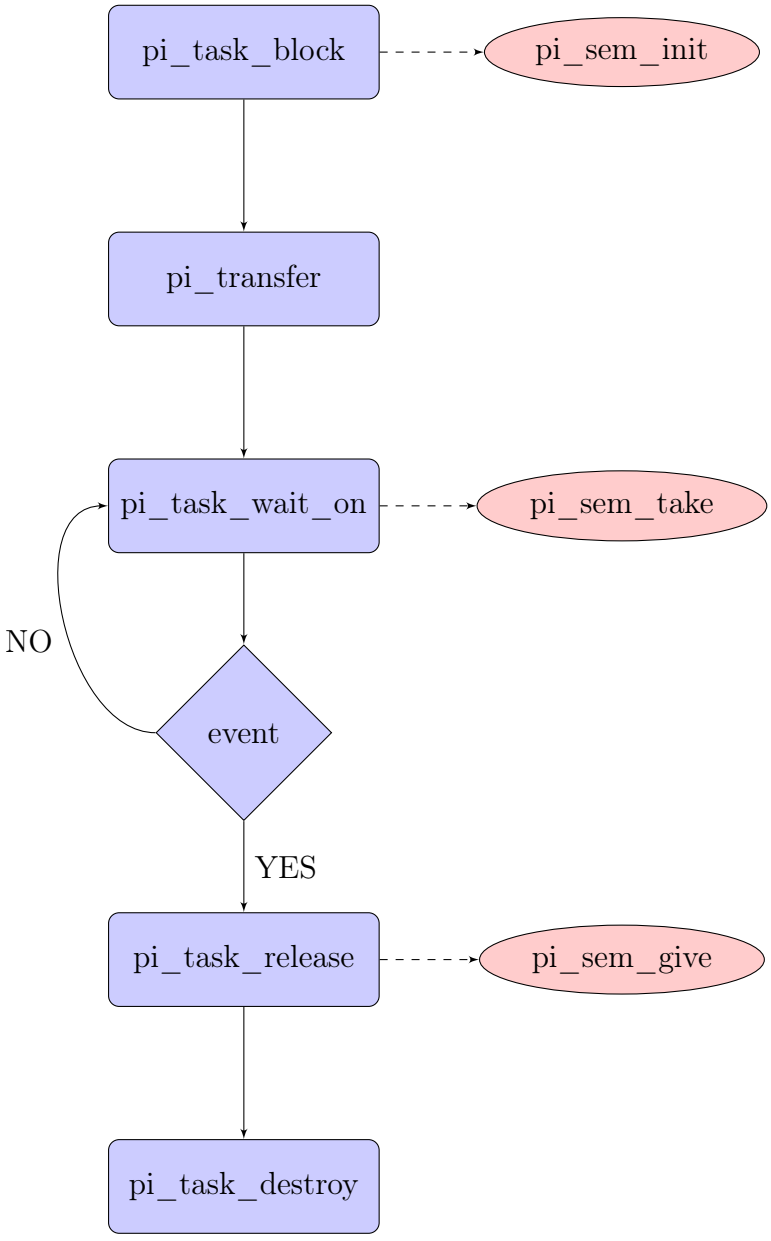

Figura 3.6: Diagramma di flusso per la gestione dei task per trasferimento sincrono in FreeRTOS

quindi una risorsa condivisa, finchè non avviene l'evento di fine trasferimento, quindi un interrupt. Solo allora verrà rilasciato il task e quindi il semaforo a contatore. Infine con la funzione pi_task_destroy verrà cancellato il task.
In caso di trasferimento di dati in modo asincrono, succede che avvengono più trasferimenti di fila. Dopo l'invio o la ricezione dei dati, attraverso la periferica di comunicazione SPI o I^2C, non si va ad aspettare un event task ma avvengono altri trasferimenti. Solo alla fine di tutti questi trasferimenti viene chiamata la funzione pi_task_wait_on che prende un semaforo a contatore per ogni task, fino a quando non arriva l'evento di fine trasferimento per ognuno.

3.4 Applicazione di PULP-OS sulla piattaforma PULP

Introduzione

PULP-OS è un RTOS, sviluppato da GreenWaves Technologies. PULP-OS contiene una quantità minima di codice che implementa le funzionalità di base di un RTOS, inclusa la gestione dei task e la comunicazione tra i vari task, ed è scritto in C. PULP-OS è stato usato per lo sviluppo di driver per le applicazioni in tempo reale su sistemi basati su PULP.

Gestione dei Task in PULP-OS

Di seguito viene descritto come vengono implementare, Fig. 3.7, le dichiarazioni della Fig. 3.3.
Con la funzione _pi_task_block si va a inizializzare un'istanza di event task. Verrà associato al array arg del task, Fig. 3.4, l'indirizzo della funzione pos_task_handle_blocking, il puntatore del task stesso e si imposta la variabile di stato, done, a 0. La funzione pos_task_handle_blocking viene utilizzata come callback per cambiare lo stato del task.

La funzione _pi_task_wait_on permette di aspettare un event task fino a quando lo stato del task non cambia. Allora quando avviene un evento, il task viene associato a uno scheduler. Successivamente la funzione pos_task_handle andrà a chiamare la callback, definita nella funzione _pi_task_block, per rilasciare il task. La variabile di stato del task, done, passerà a 1, e si uscirà da questa procedura.
La funzione _pi_task_push è usata per mettere un event task in un event kernel.
La funzione _pi_task_destroy va a rimuovere un event task.

Quando viene associato un task ad una funzione che gestisce il

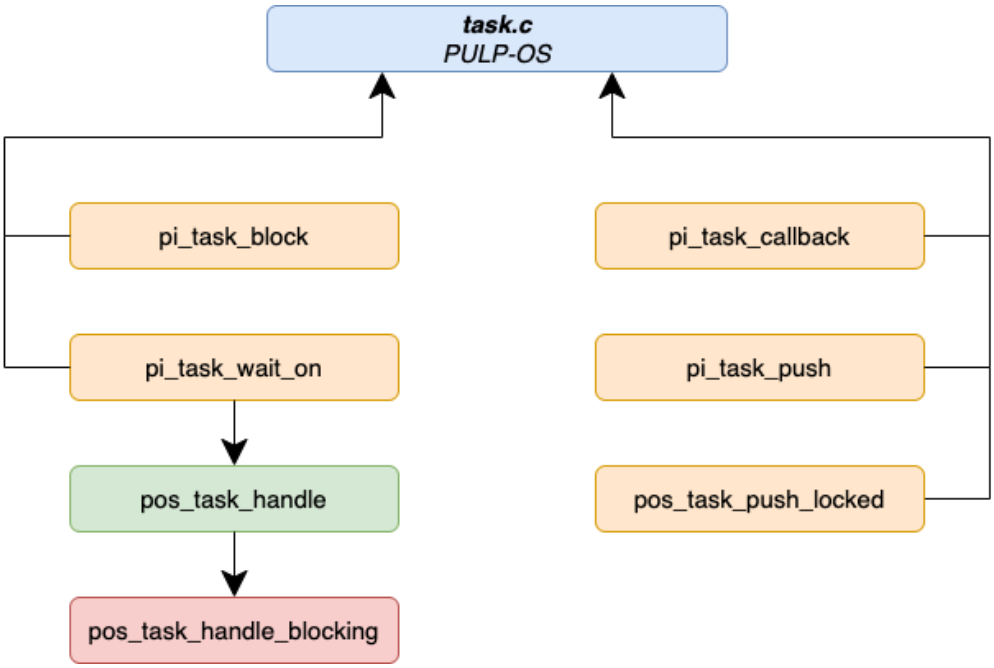

Figura 3.7: Gestione Task per trasferimento sincrono in PULP-OS

trasferimento di dati, in modo sincrono, Fig. 3.8, quello che succede è che inizialmente viene chiamata la funzione pi_task_block, che va a inizializzare array del task. Dopodiché si ha il trasferimento dei dati, questo attraverso la periferica di comunicazione SPI o I^2C. Una volta chiamata questa funzione, il Firmware si porta in pi_task_wait_on. In questa proceduta verrà rilasciato

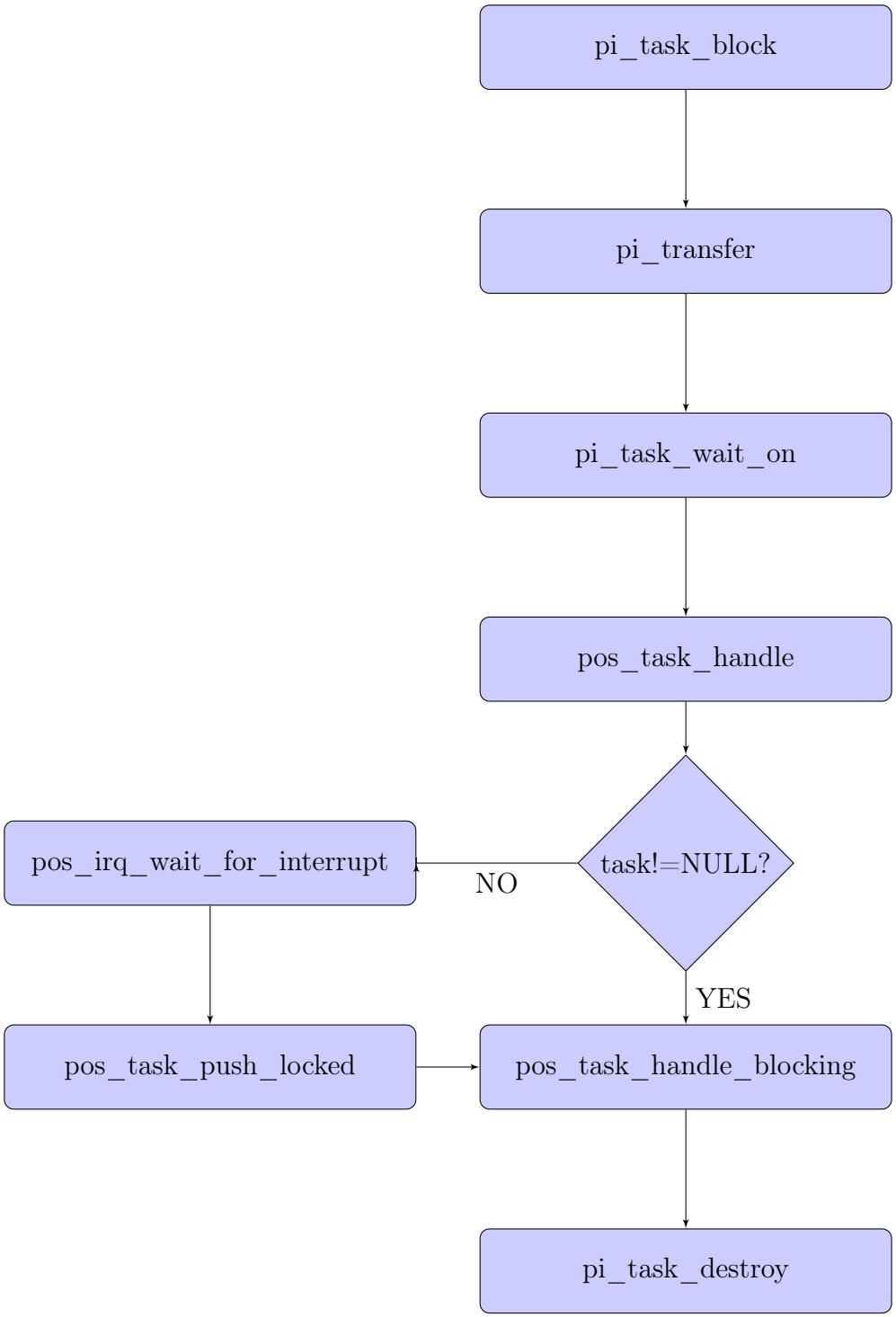

Figura 3.8: Diagramma di flusso per la gestione dei task per trasferimento sincrono in PULP-OS

un task, qualora sia arrivato un evento. In caso contrario si aspetta finché questo arrivi. Infine con la funzione pi_task_destroy verrà cancellato il task.

In caso di trasferimento di dati in modo asincrono, succede che avvengono più trasferimenti di fila. Dopo l'invio o la ricezione dei dati, attraverso la periferica di comunicazione SPI o I^2C, non si va ad aspettare un event task ma avvengono altri trasferimenti. Solo alla fine di tutti questi trasferimenti viene chiamata la funzione pi_task_wait_on. Quando arriva un evento, il task viene associato ad una lista. Questa lista verrà poi utilizzata nella funzione pos_task_handle, per fare il release dei task.

Capitolo 4

Sviluppo dei driver per la piattaforma PULP

4.1 Descrizione del problema

Questo capitolo mostra come sono stati sviluppati i driver per le periferiche di comunicazione SPI e I^2C per la piattaforma PULP. Come mostra la Fig. 2.7 e la Fig. 4.1, i driver per la periferica di comunicazione SPI e I^2C utilizzano quelle che sono le PMSIS. Dato che SDK sfrutta due RTOS differenti, cioè FreeRTOS e PULP-OS, i driver per le periferiche di comunicazione devono essere implementati in maniera diversa per poterli sfruttare. Come soluzione a questo problema si è realizzato un abstraction-layer, in modo tale da poter utilizzare lo stesso driver con due RTOS differenti. Di conseguenza si ha che i due driver avranno dei file comuni e file dipendenti dal RTOS scelto, che prendono il nome di File OS Dependent. I driver per la periferica di comunicazione SPI e per la periferica di comunicazione I^2C hanno una struttura come quella mostrata in Fig. 4.3 e Fig. 4.4. I driver per funzionare sfruttano quelle che sono le PMSIS.

Le PMSIS, PULP Microcontroller Software Interface Standard, Fig. 4.2, rappresentano un abstraction-layer per la piattaforma PULP, dove vengono definite le dichiarazioni per le funzioni che si interfacciano con il processore,

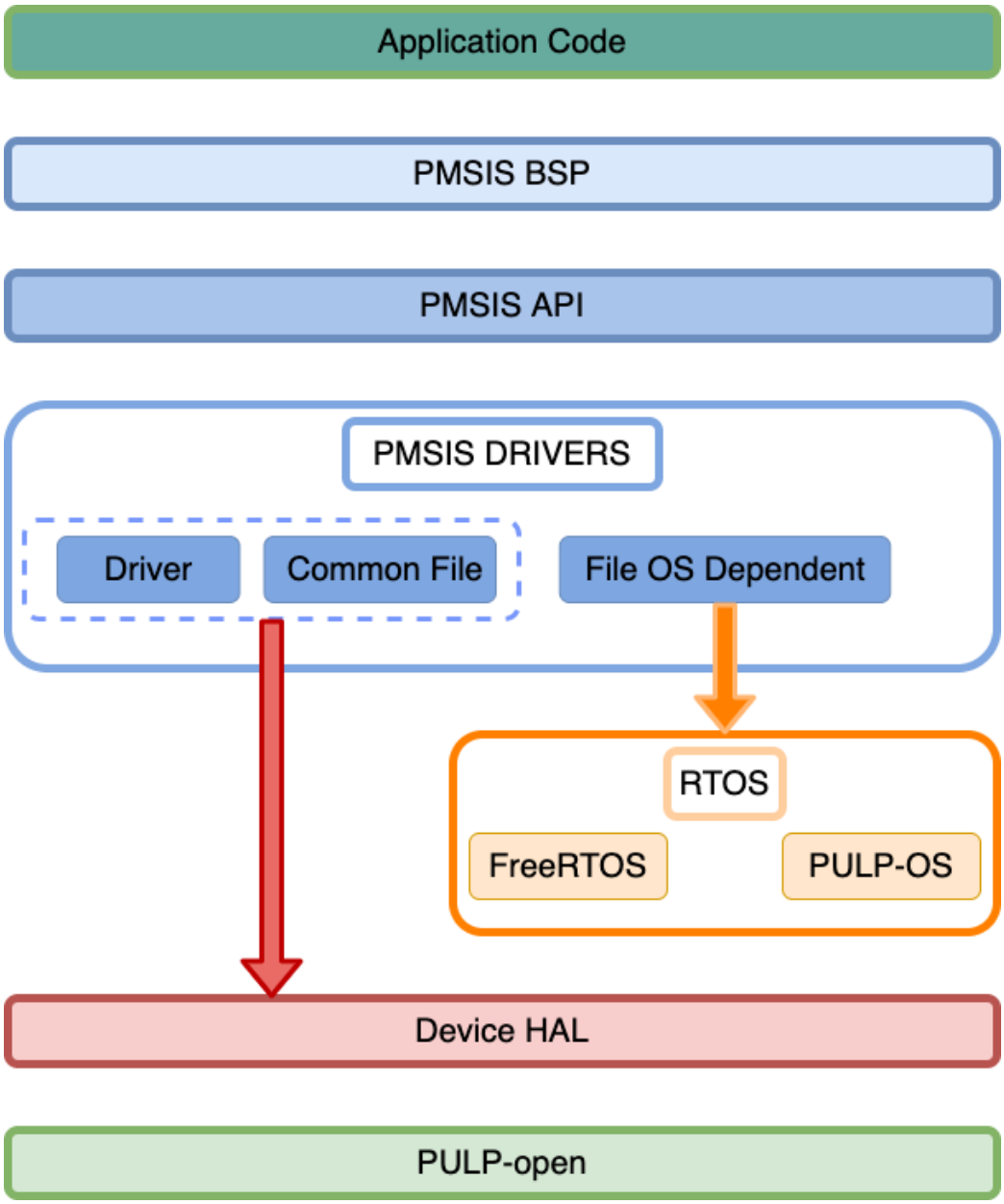

Figura 4.1: Software Stack Driver

periferiche, RTOS e componenti middleware. Grazie a queste interfacce si semplifica l'utilizzo del software e riducono la curva di apprendimento per gli sviluppatori.

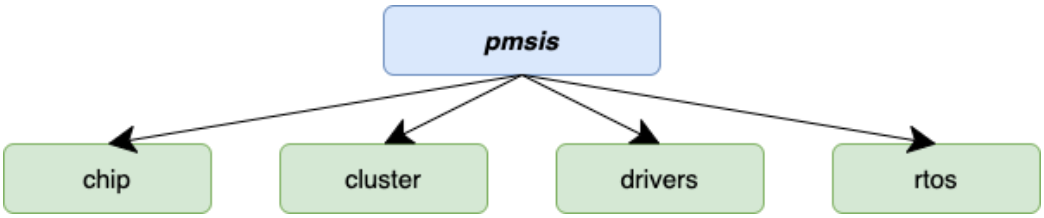

Figura 4.2: PMSIS in PULP

All'interno delle PMSIS ci sono gli header file, come spi.h e i2c.h, che contengono le dichiarazioni delle funzioni che devono essere implementate nel driver. Molte funzioni, come la pi_spi_open, hanno delle sotto-implementazioni, ad esempio __pi_spi_open. Quest'ultime saranno implementate in maniera diversa in base al RTOS che si sta utilizzando.
Il driver, per la periferica di comunicazione SPI o I^2C, si divide in tre parti. Si ha una parte comune, dove si inseriscono le parti comuni del driver rispetto ai due RTOS, ad esempio funzioni che manipolano le liste. Questa contiene due file che prendono il nome di common_spi.h, o common_i2c.h, in cui ci sono le dichiarazioni delle funzioni e le strutture dati, e common_spi.c, o common_i2c.c, dove ci sono le definizioni delle funzioni precedentemente dichiarate.
Una seconda parte, in cui si implementano le funzioni diverse dei driver per i due RTOS, ad esempio le funzioni di basso livello che vengono chiamate dalle definizioni delle PMSIS. Questa parte contiene due file che prendono il nome di abstraction_layer_spi.h, o abstraction_layer_i2c.h, in cui ci sono le dichiarazioni delle funzioni, e abstraction_layer_spi.c, o abstraction_layer_i2c.c, dove ci sono le definizioni delle funzioni precedentemente dichiarate.
Infine si ha una terza parte in cui si implementano le dichiarazioni delle PMSIS per i driver. Questa prende il nome di spim-v3.c o i2c-v2.c.

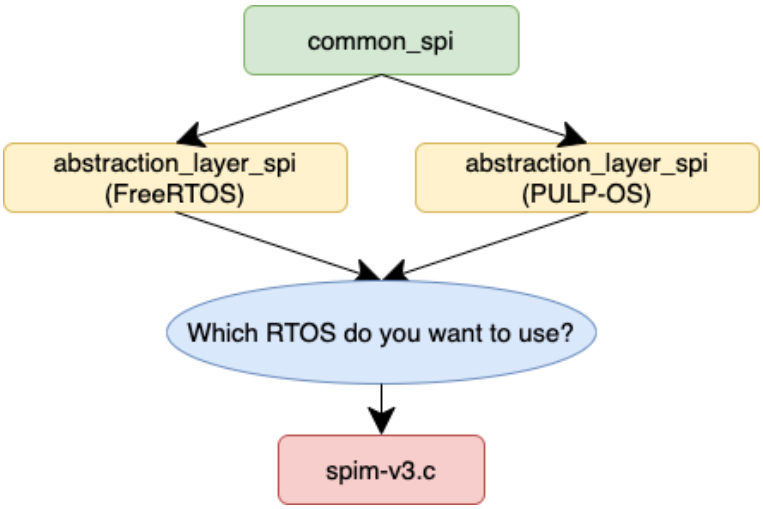

Figura 4.3: Disposizione file driver spi

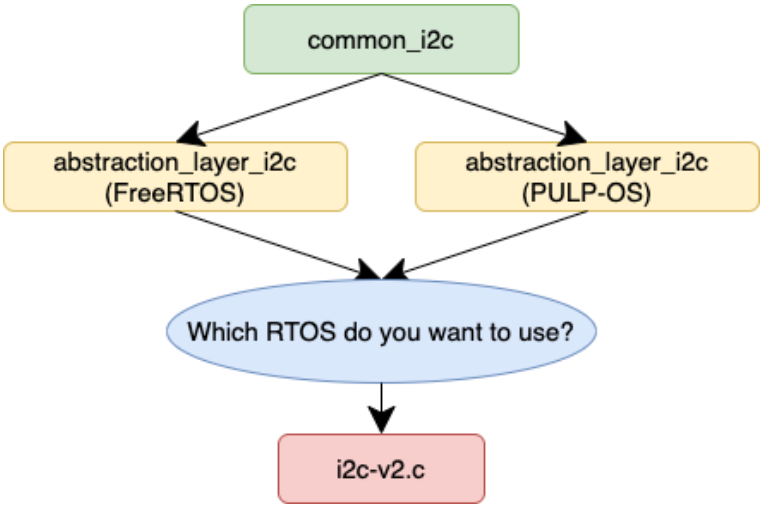

Figura 4.4: Disposizione file driver i2c

4.2 Sviluppo dei driver SPI per la piattaforma PULP

Il driver per la periferica di comunicazione SPI ha una struttura come quella mostrata in Fig. 4.3. Il driver per funzionare sfrutta quelle che sono le

PMSIS. All'interno delle PMSIS c'è header file, spi.h, Fig. 4.5, che contiene le dichiarazioni delle funzioni che devono essere implementate nel driver, Fig. 4.6. Come si può vedere dalla Fig. 4.6, molte funzioni, come la pi_spi_open, hanno delle sotto-implementazioni, ad esempio __pi_spi_open. Quest'ultime saranno implementate in modo diversa in base al RTOS che si sta utilizzando.

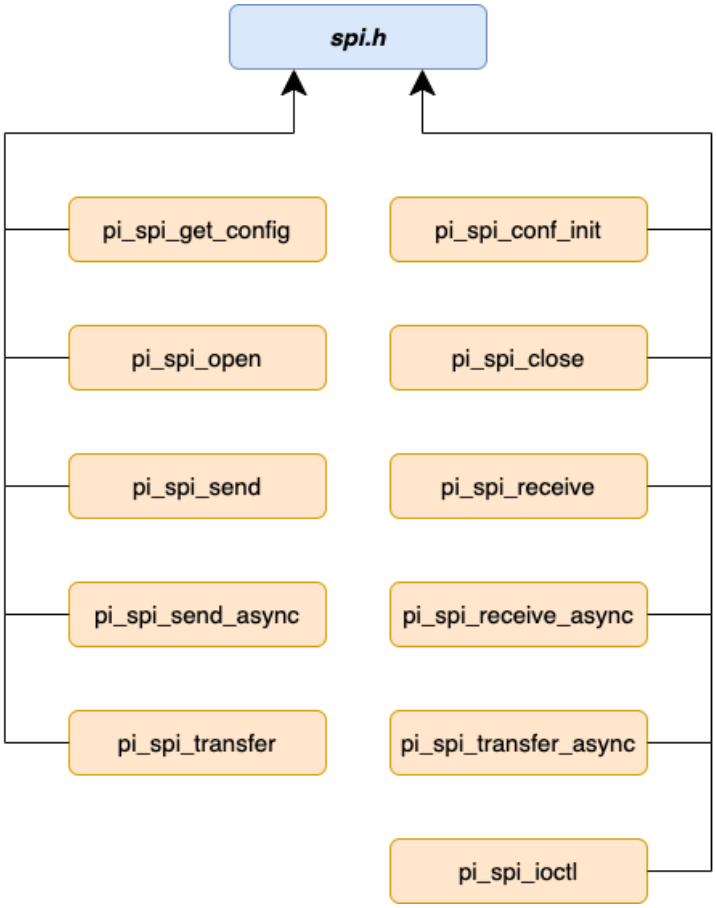

Figura 4.5: Dichiarazione delle API per SPI

Di seguito vengono descritte le funzioni di Fig. 4.5.

La funzione pi_spi_conf_init serve per impostare i valori predefiniti della struttura che descrive la periferica di comunicazione SPI.

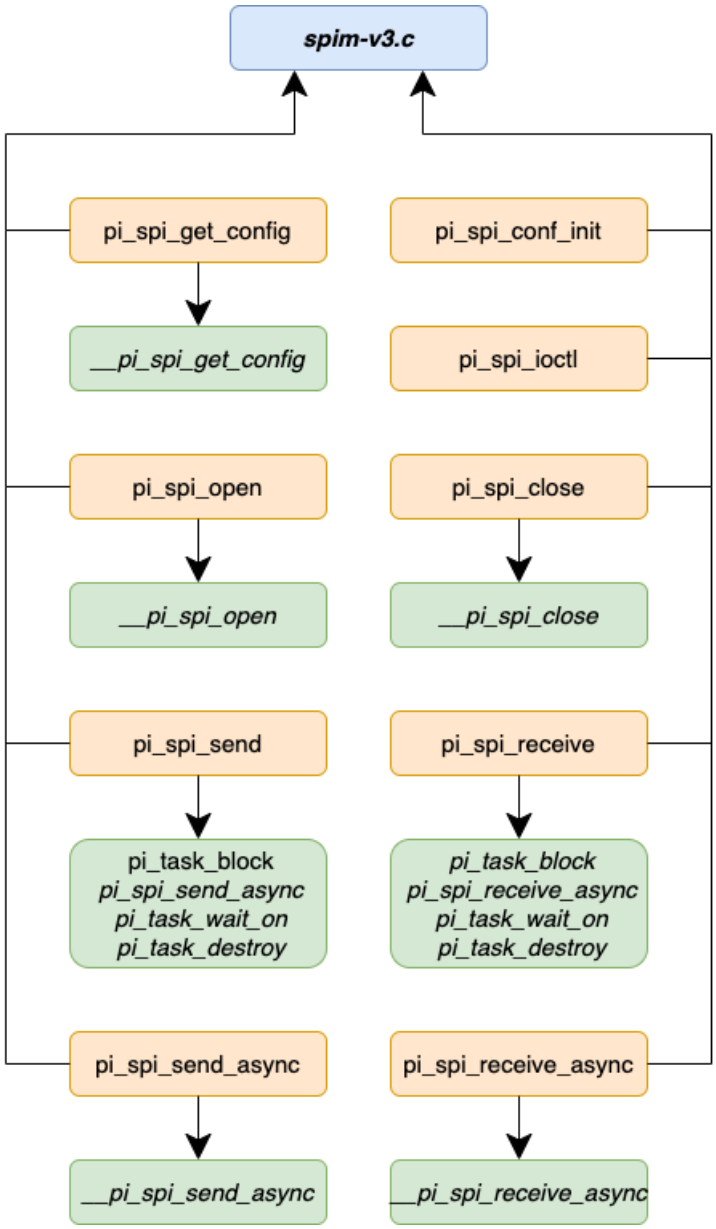

Figura 4.6: Definizione delle API per SPI

La funzione pi_spi_open deve essere richiamata prima di poter utilizzare un interfaccia SPI. Questa funzione farà tutta la configurazione necessaria per rendere l'interfaccia SPI utilizzabile. Il chiamante verrà bloccato fino al termine dell'operazione. L' implementazione di questa funzione sarà diversa

a seconda del RTOS utilizzato.

La funzione pi_spi_close si utilizza per chiudere un'interfaccia SPI precedentemente aperta, tramite la funzione pi_spi_open, in modo da liberare le risorse allocate. Una volta che questa funzione viene usata su un'interfaccia SPI, per poter riutilizzare l'interfaccia bisogna riaprirla nuovamente. L' implementazione di questa funzione sarà diversa a seconda del RTOS utilizzato.

La funzione pi_spi_ioctl si utilizza per cambiare parte della configurazione dell'interfaccia SPI una volta che questa è stata aperta.

La funzione pi_spi_send si utilizza per inviare dati alla periferica di comunicazione SPI. Questa procedura effettuerà un trasfeirmento sincrono tra l'interfaccia SPI e il chip di memoria. Questa utilizza le linee di MOSI-MISO, ma è possibile utilizzare anche il trasferimento QSPI impostando alcuni flags. In questo caso il chiamante è bloccato fino alla fine del trasferimento. Le sotto implementazioni di questa funzione cambiano a seconda del RTOS scelto, come mostrato in Fig. 4.7 e Fig. 4.8.

La funzione pi_spi_receive si utilizza per ricevere dati dalla periferica di comunicazione SPI. Questa procedura effettuerà un trasfeirmento sincrono tra l'interfaccia SPI e il chip di memoria. Questa utilizza le linee di MOSI-MISO. In questo caso il chiamante è bloccato fino alla fine del trasferimento. Le sotto implementazioni di questa funzione cambiano a seconda del RTOS scelto, come mostrato in Fig. 4.9 e Fig. 4.10.

La funzione pi_spi_transfer si utilizza per inviare e ricevere dati dalla periferica di comunicazione SPI sfruttando la modalità Full-Duplex. Questa procedura effettuerà un trasfeirmento sincrono tra l'interfaccia SPI e il chip di memoria. Questa utilizza le linee di MOSI-MISO, ma è possibile utilizzare anche il trasferimento QSPI impostando alcuni flags. In questo caso il chiamante è bloccato fino alla fine del trasferimento.

La funzione pi_spi_send_async si utilizza per inviare dati alla periferica di comunicazione SPI. Questa procedura effettuerà un trasfeirmento asincrono tra l'interfaccia SPI e il chip di memoria. Questa utilizza le linee di

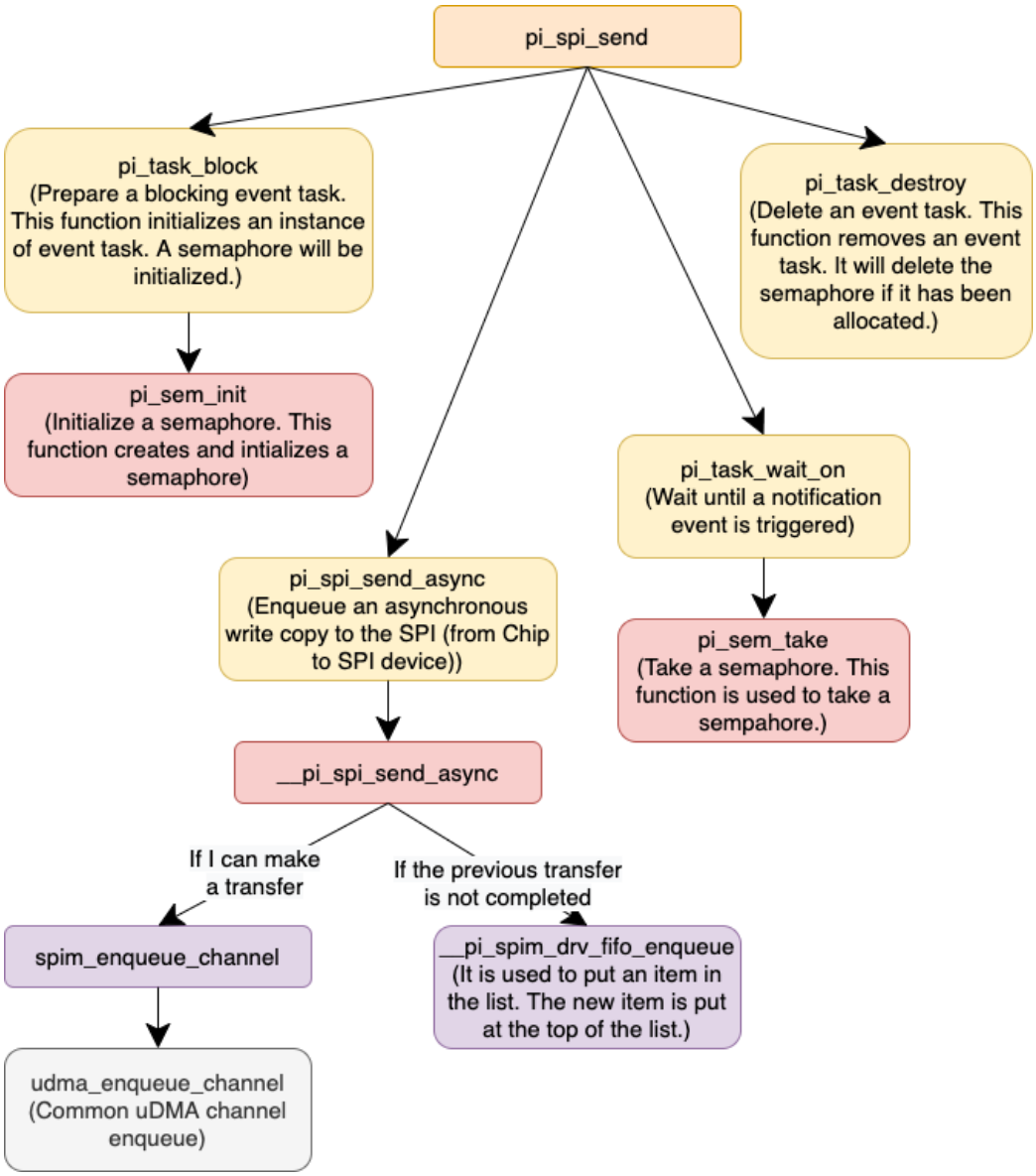

Figura 4.7: Schema pi_spi_send per SPI in FreeRTOS

MOSI-MISO, ma è possibile utilizzare anche il trasferimento QSPI impostando alcuni flags. In questo caso il chiamante è bloccato fino alla fine del trasferimento e un task deve essere utilizzato per notificare la fine di un trasferimento al chiamante. L' implementazione di questa funzione sarà diversa

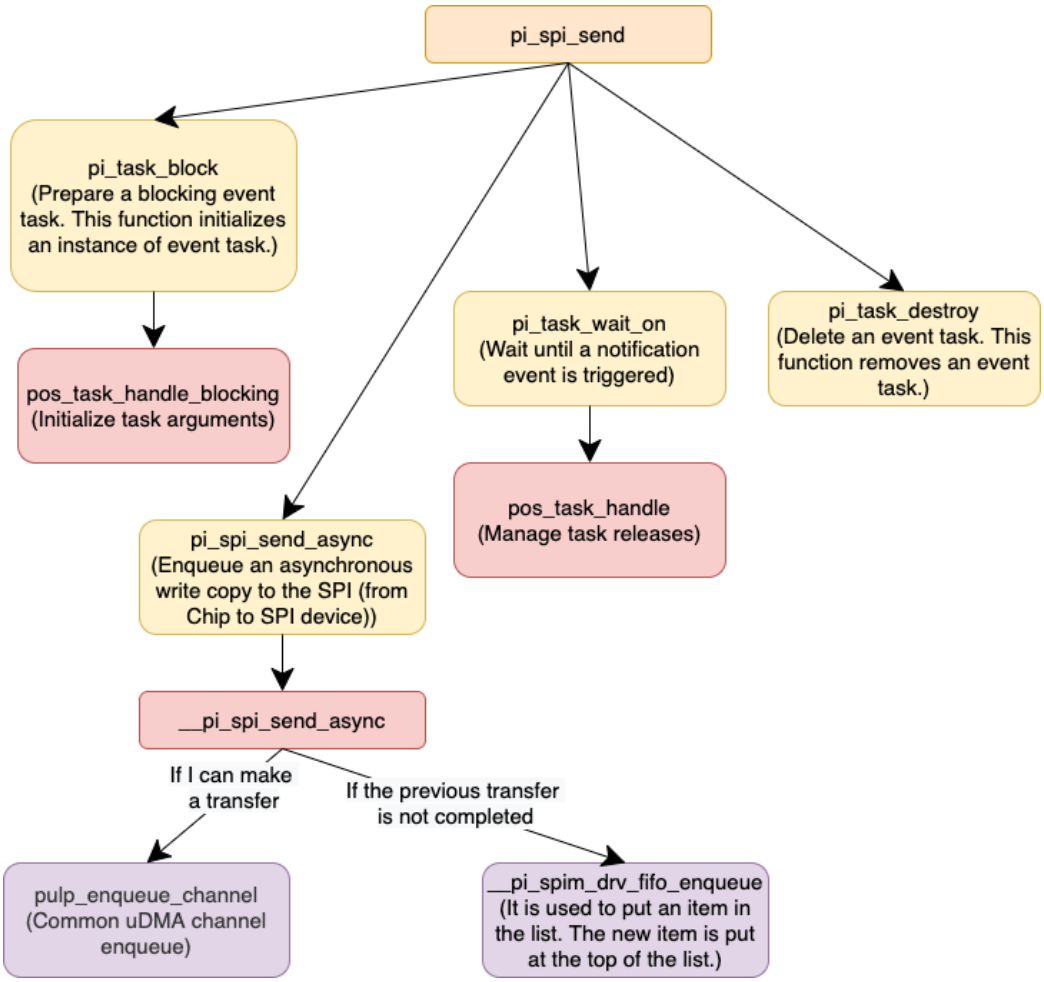

Figura 4.8: Schema pi_spi_send per SPI in PULP-OS

a seconda del RTOS utilizzato.

La funzione pi_spi_receive_async si utilizza per ricevere dati dalla periferica di comunicazione SPI. Questa procedura effettuerà un trasfeirmento asincrono tra l'interfaccia SPI e il chip di memoria. Questa utilizza le linee di MOSI-MISO, ma è possibile utilizzare anche il trasferimento QSPI impostando alcuni flags. In questo caso il chiamante è bloccato fino alla fine del trasferimento e un task deve essere utilizzato per notificare la fine di un trasferimento al chiamante. L' implementazione di questa funzione sarà diversa

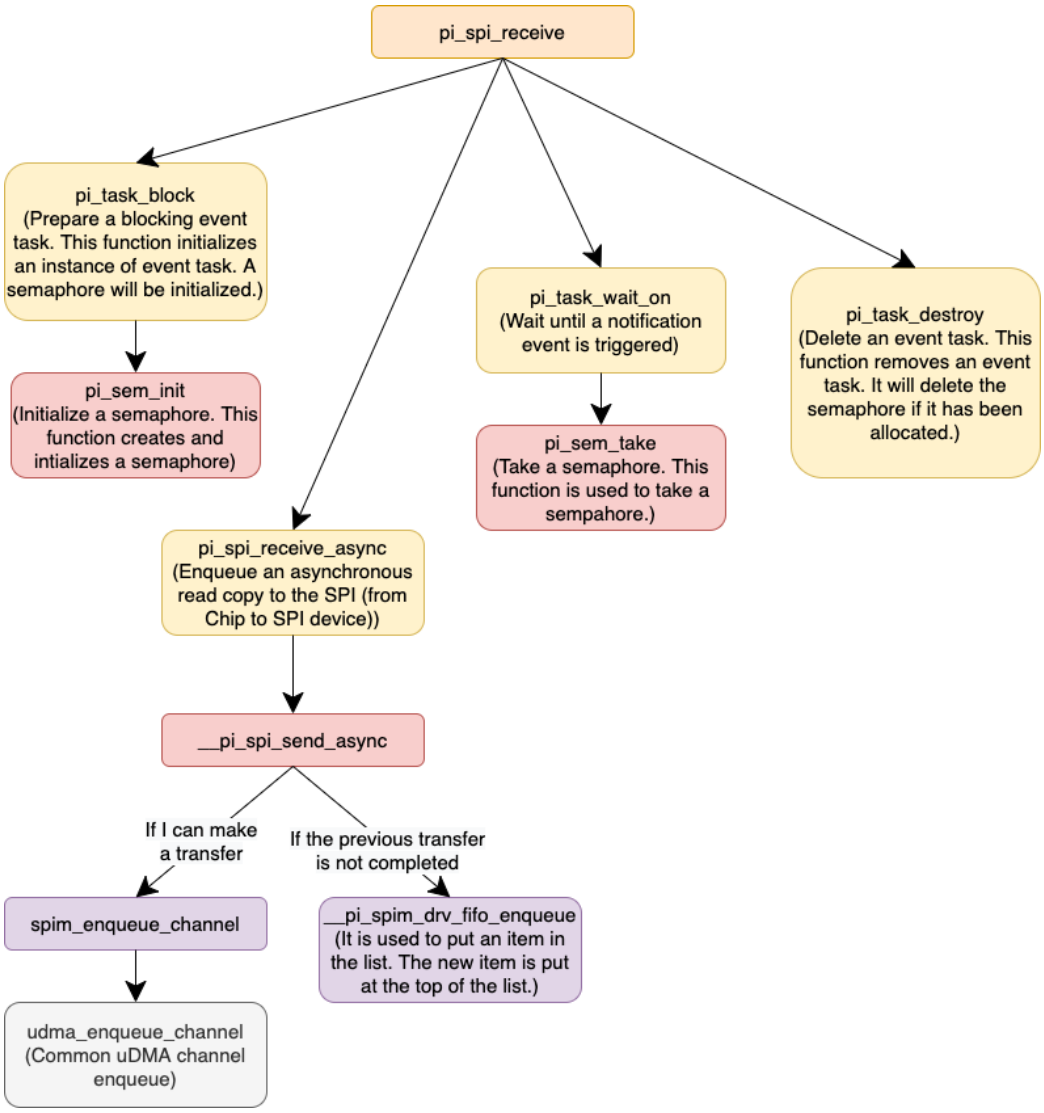

Figura 4.9: Schema pi_spi_receive per SPI in FreeRTOS

a seconda del RTOS utilizzato.

La funzione pi_spi_transfer_async si utilizza per inviare e ricevere dati dalla periferica di comunicazione SPI sfruttando la modalità Full-Duplex. Questa procedura effettuerà un trasfeirmento asincrono tra l'interfaccia SPI e il chip di memoria. Questa utilizza le linee di MOSI-MISO, ma è possibile utilizzare anche il trasferimento QSPI impostando alcuni flags. In questo

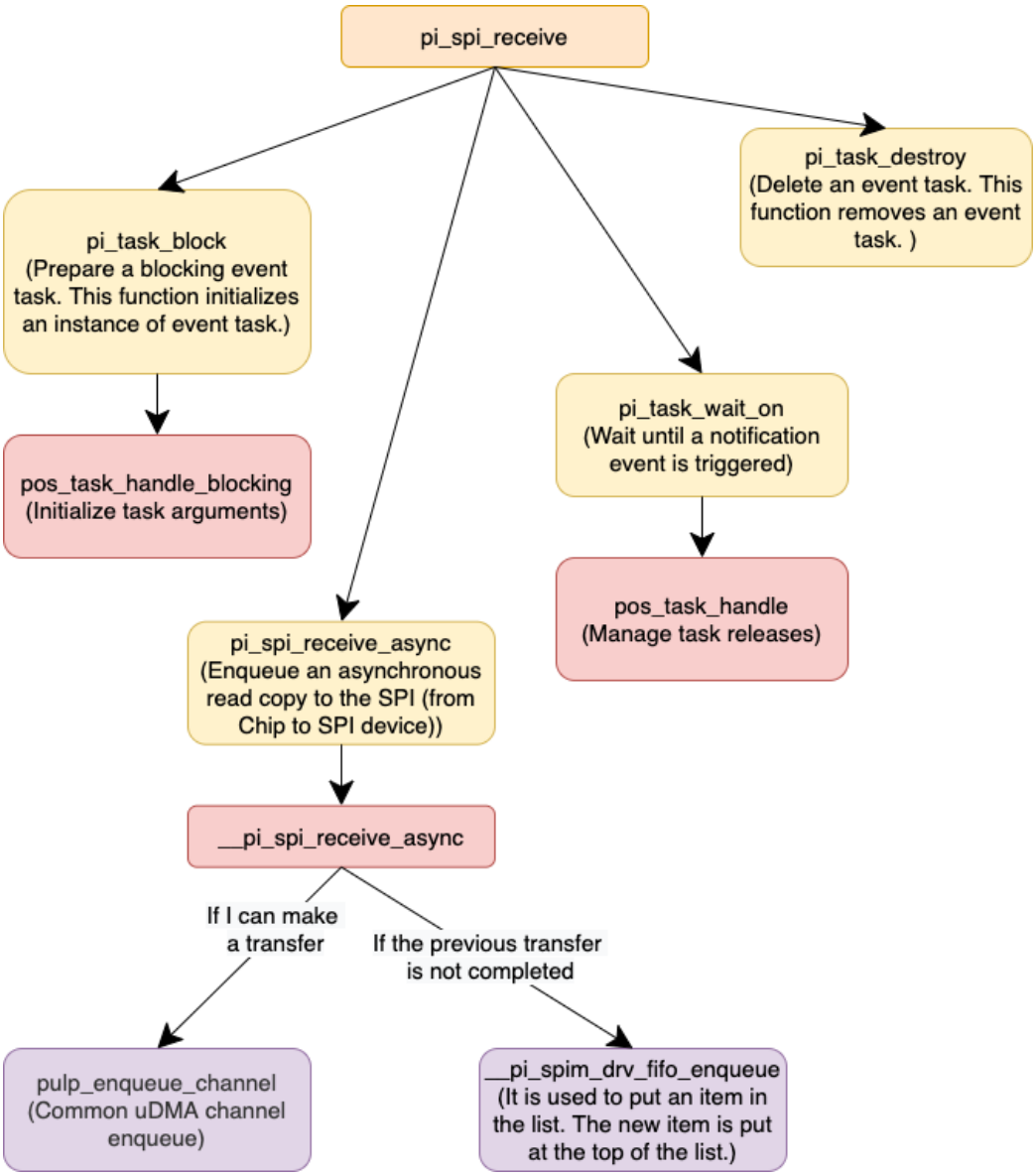

Figura 4.10: Schema pi_spi_receive per SPI in PULP-OS

caso il chiamante è bloccato fino alla fine del trasferimento e un task deve essere utilizzato per notificare la fine di un trasferimento al chiamante.

La funzione pi_spi_get_config si utilizza per ottenere la configurazione dell'interfaccia SPI.

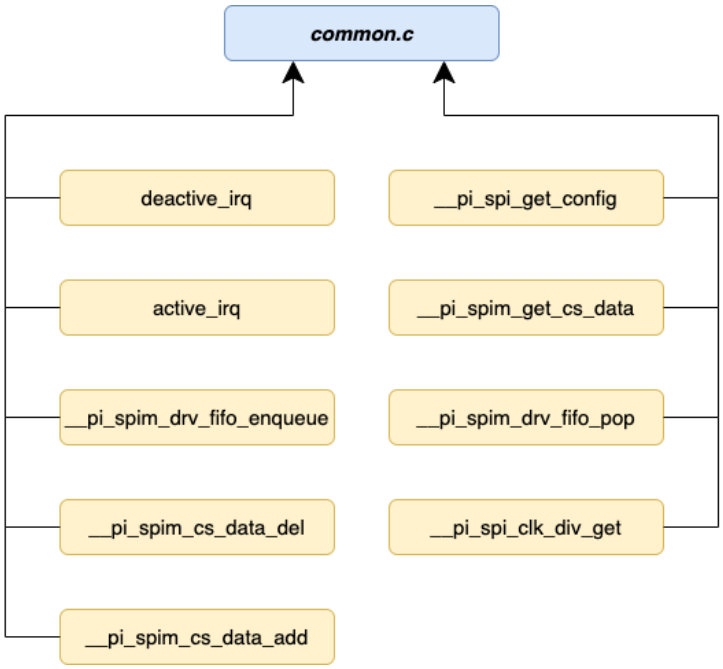

Figura 4.11: Schema del common file per SPI

Con riferimento alla Fig. 4.11, al interno dei file common_spi.h e common_spi.c ci sono rispettivamente le dichiarazioni e le definizioni di funzioni e strutture dati comuni ai driver scritti per i due RTOS.

La funzione deactive_irq si utilizza per disattivare gli interrupt, mentre la funzione active_irq si utilizza per attivare gli interrupt. Queste due funzioni vengono usate per eseguire delle procedure in modo atomico, come ad esempio quelle che operano sulle liste.

La funzione __pi_spim_drv_fifo_enqueue si utilizza per inserire un elemento in una memoria FIFO di tipo software.

La funzione __pi_spim_cs_data_del si utilizza per cancellare un elemento da una lista.

La funzione __pi_spim_cs_data_add si utilizza per inserire un elemento in una lista.

La funzione __pi_spi_get_config restituisce le informazioni dell'interfaccia SPI.

La funzione __pi_spim_get_cs_data va a trovare una certa interfaccia SPI in una lista.

La funzione __pi_spim_drv_fifo_pop serve per portare un elemento in cima ad una FIFO.

La funzione __pi_spi_clk_div_get si utilizza per impostare il divisore di clock per l'interfaccia SPI.

La struttura dati spim_cs_data, Fig. 4.12, contiene le informazioni del dispositivo connesso al CS dell'interfaccia SPI.

```
struct spim_cs_data {
        struct spim_cs_data *next;
        struct spim_driver_data *drv_data;
        uint32_t cfg;
        uint32_t udma_cmd[8];
        uint32_t max_baudrate;
        uint32_t polarity;
        uint32_t phase;
        uint8_t cs;
        uint8_t wordsize;
        uint8_t big_endian;
};
```

Figura 4.12: Struttura spim_cs_data

La struttura dati spim_driver_data si definisce in modo diverso nei casi in cui si utilizzi FreeRTOS, Fig. 4.13, o PULP-OS, Fig. 4.14, in entrambi i casi contiene le informazioni dell'interfaccia SPI.

La struttura dati spim_transfer, Fig. 4.15, contiene le informazioni sul trasferimento in corso.

```
struct spim_driver_data {
        struct spim_drv_fifo *drv_fifo;
        struct spim_cs_data *cs_list;
        pi_task_t *repeat_transfer;
        pi_task_t *end_of_transfer;
        uint32_t nb_open;
        uint8_t device_id;
};
```

Figura 4.13: Struttura spim_driver_data per FreeRTOS

```
struct spim_driver_data
{
        struct spim_drv_fifo *drv_fifo;
        struct spim_cs_data *cs_list;
        pi_task_t *repeat_transfer;
        pi_task_t *end_of_transfer;
        uint32_t nb_open;
        uint8_t device_id;
        pos_udma_channel_t *rx_channel;
        pos_udma_channel_t *tx_channel;
};
```

Figura 4.14: Struttura spim_driver_data per PULP-OS

4.2.1 Abstraction Layer SPI in FreeRTOS

L' abstraction layer per il driver SPI in FreeRTOS utilizza le funzioni di Fig. 4.16. Di seguito verranno mostrate le implementazioni per queste funzioni.

La funzione __pi_spi_open, Fig. 4.17, serve per poter andare ad aprire l'interfaccia SPI. Questa funzione va inizialmente ad abilitare il clock per l'interfaccia SPI scelta. Successivamente va ad abilitare la linea di interrupt associata agli eventi prodotti dal SoC e poi attiva l'evento associa-

```
struct spim_transfer {
        pi_spi_flags_e flags;
        void *data;
        uint32_t len;
        uint32_t cfg_cmd;
        uint32_t byte_align;
        uint32_t is_send;
};
```

Figura 4.15: Struttura spim_transfer

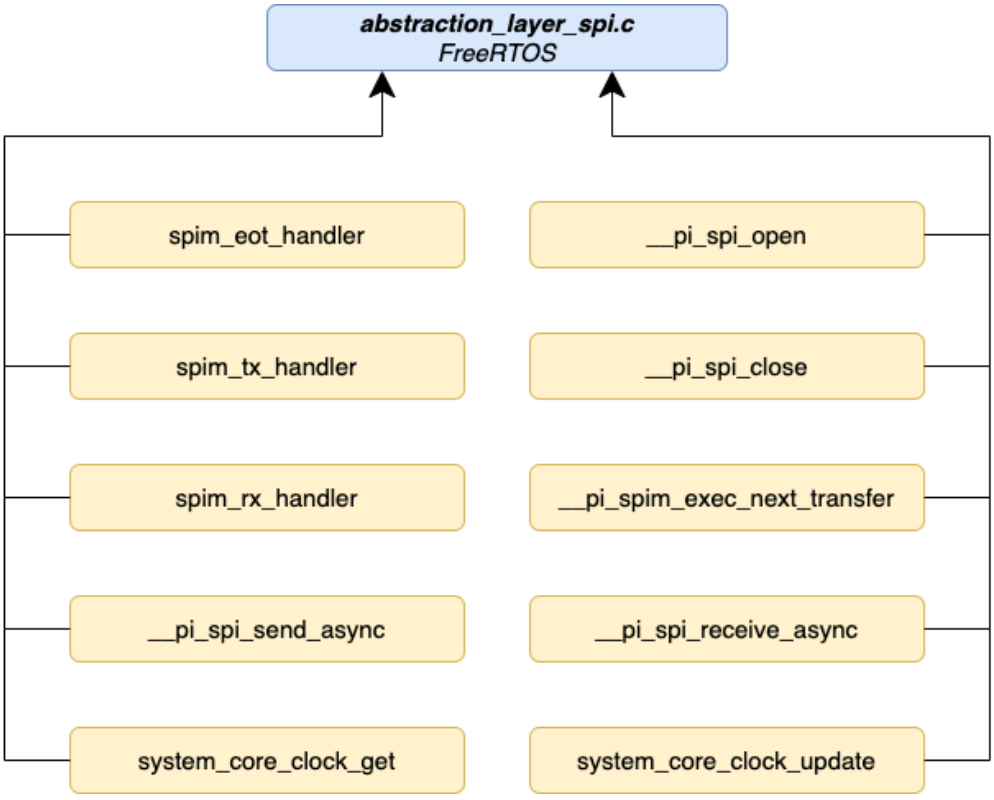

Figura 4.16: Schema del abstraction layer per SPI in FreeRTOS

to al segnale di EOT della periferica SPI. Poi viene impostato un array, pi_fc_event_handler_set, contenente l'indirizzo della ISR da eseguire in

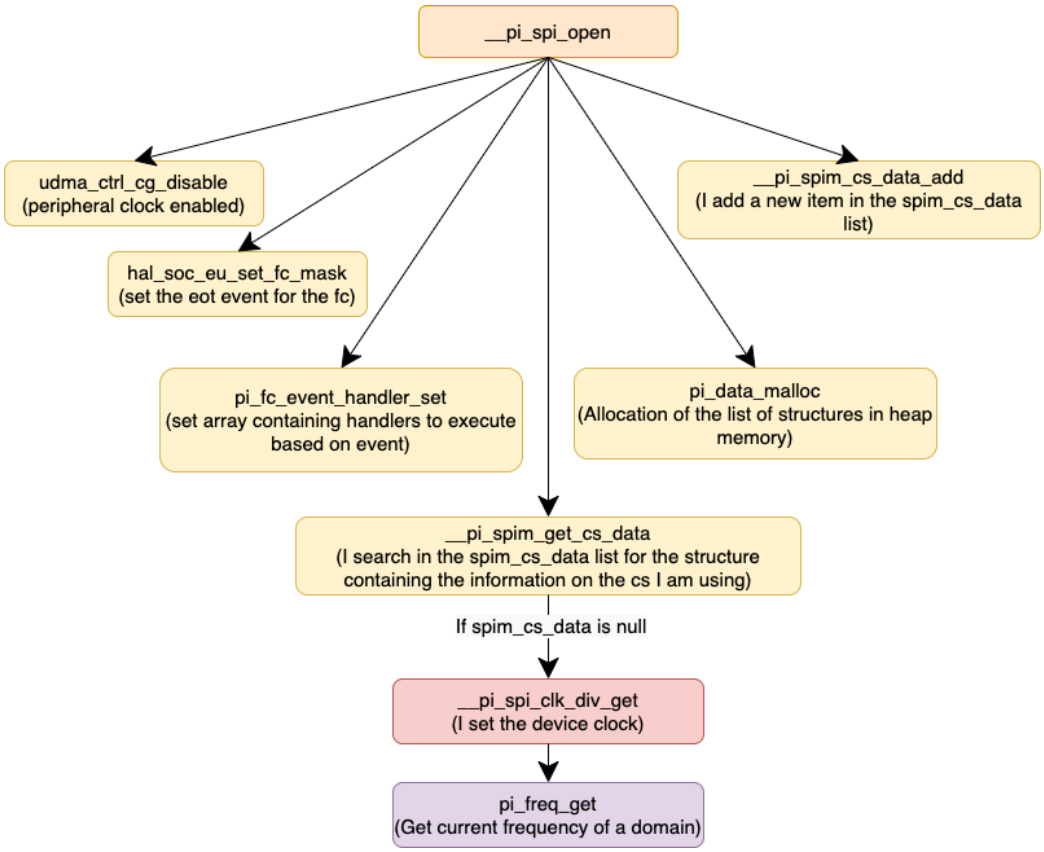

Figura 4.17: Schema Freertos per SPI di __pi_spi_open

base al evento settato. Dopodiché si vanno ad allocare le strutture dati contenenti le informazioni per ogni interfaccia SPI e le infromazioni del dispositivo connesso al CS. Per allocare queste strutture, si utilizza la funzione pi_data_malloc, con la quale si va ad allocare memoria nel heap e successivamente con la funzione __pi_spim_cs_data_add si aggiunge la struttura che contiene le informazioni del dispositivo connesso all'interfaccia SPI in una lista, mentre la struttra che descrive l'interfaccia SPI che si sta utilizzando viene salvata nel array __g_spim_drv_data.

La funzione __pi_spi_close, Fig. 4.18, va ad eliminare dalla lista la struttura dati contenente le informazioni sul dispositivo connesso al CS del interfaccia SPI che si sta utilizzando. Nel caso in cui non ci siano dispositivi

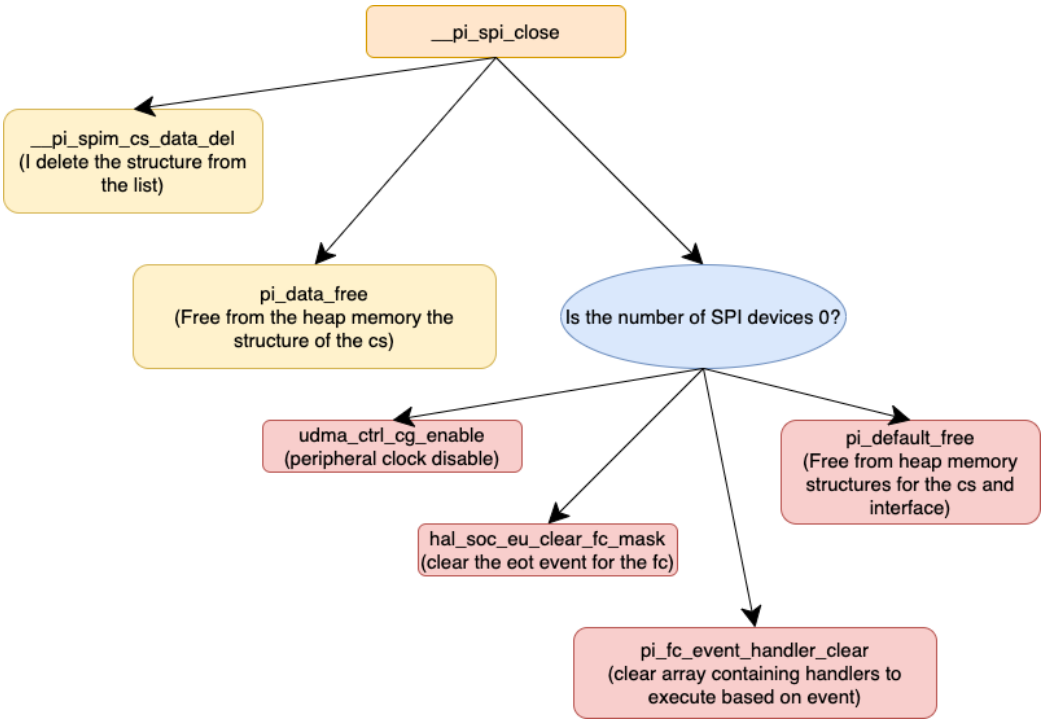

Figura 4.18: Schema Freertos per SPI di __pi_spi_close

connessi all'interfaccia SPI, allora viene disabilitato il clock della periferica di comunicazione SPI e ripulita la linea di interrupt. Infine vengo deallocate le risorse con la funzione pi_default_free.

La funzione __pi_spi_send_async, Fig. 4.19, va a controllare se è possibile fare l'invio dei dati oppure no. Nel caso in cui non sia possibile, ad esempio perchè il uDMA è impegnato con un altro trasferimento, allora il trasferimento viene messo in coda in una FIFO software, questo con la funzione __pi_spim_drv_fifo_enqueue. Quando il trasferimento dei dati è possibile, viene riempito il buffer di comandi cs_data->udma_cmd[ ], dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.4.3, e successivamente si utilizza la funzione spim_enqueue_channel per inviare sia il buffer di comandi che l'indirizzo della partizione di memoria in L2 dove si vogliono andare a prelevare i dati da inviare.

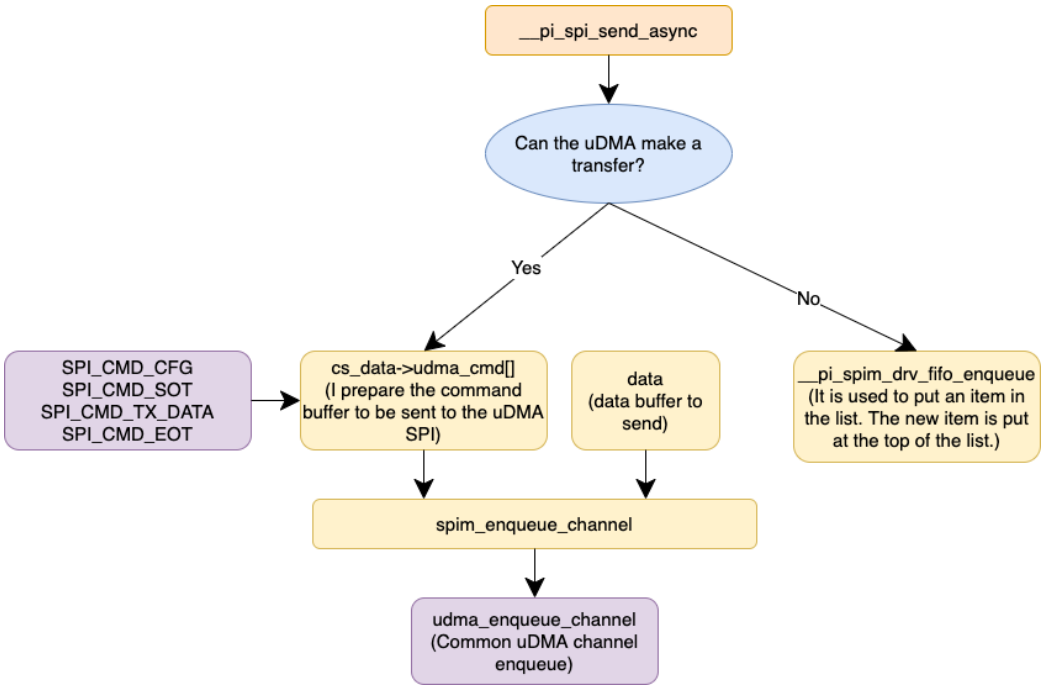

Figura 4.19: Schema Freertos per SPI di __pi_spi_send_async

La funzione __pi_spi_receive_async, Fig. 4.20, va a controllare se è possibile fare la ricezione dei dati oppure no. Nel caso in cui non sia possibile, ad esempio perchè il uDMA è impegnato con un altro trasferimento, allora il trasferimento viene messo in coda in una FIFO software, questo con la funzione __pi_spim_drv_fifo_enqueue. Quando il trasferimento dei dati è possibile, viene riempito il buffer di comandi cs_data->udma_cmd[ ], dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.4.3, e successivamente si utilizza la funzione spim_enqueue_channel per inviare il buffer di comandi e l'indirizzo della partizione di memoria in L2 dove si vogliono andare a collocare i dati ricevuti.

La funzione spim_eot_handler, Fig. 4.21, rappresenta ISR per l'evento di EOT prodotto dalla periferica di comunicazione SPI dopo aver eseguito un trasferimento. Nel caso in cui si utilizzassero gli eventi di tipo TX o RX, le loro ISR, cioè spim_tx_handler e spim_rx_handler, andrebbero a

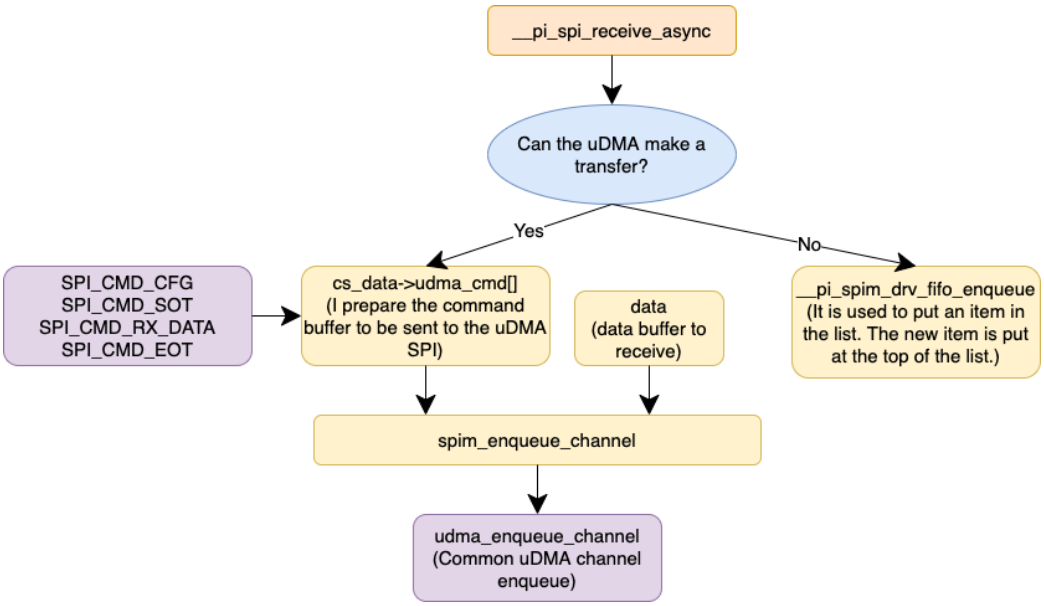

Figura 4.20: Schema Freertos per SPI di __pi_spi_receive_async

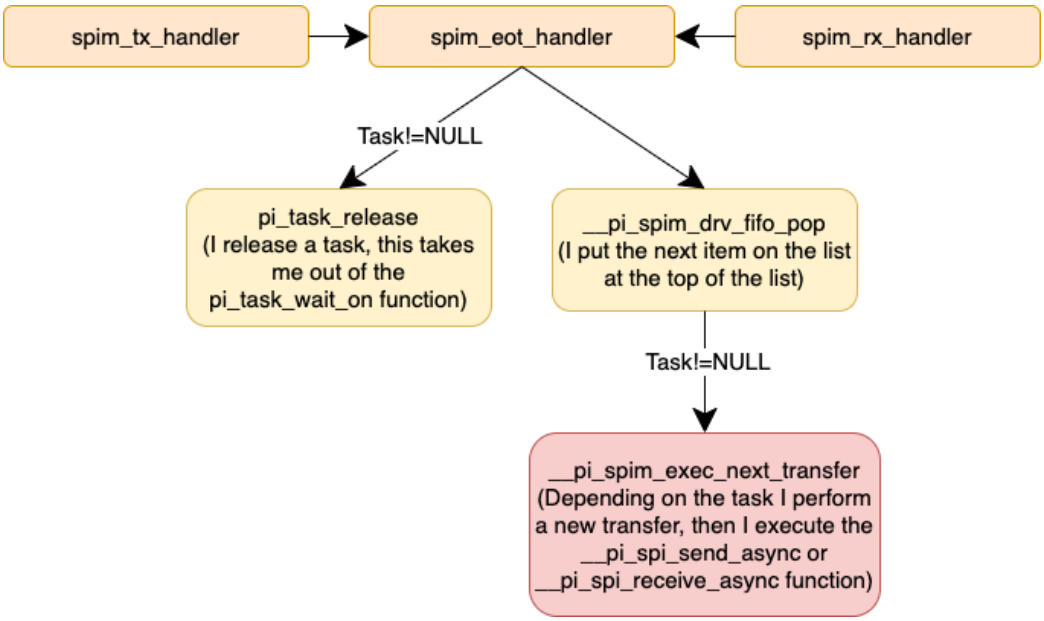

Figura 4.21: Schema Freertos per per Interrupt SPI

chiamare ISR legata al evento di EOT, cioè spim_eot_handler. Quello che fa la funzione spim_eot_handler è rilasciare il task legato al trasferimento che ha prodotto evento, attraverso la funzione pi_task_release e poi controllare se ha qualche trasferimento in coda nella FIFO software. Nel caso in cui ci sia qualche trasferimento in coda, allora viene chiamata la funzione __pi_spim_exec_next_transfer, la quale si occuperà di eseguire il nuovo trasferimento.

4.2.2 Abstraction Layer SPI in PULP-OS

L' abstraction layer per il driver SPI in PULP-OS utilizza le funzioni di Fig. 4.22. Di seguito verranno mostrate le implementazioni per queste funzioni.

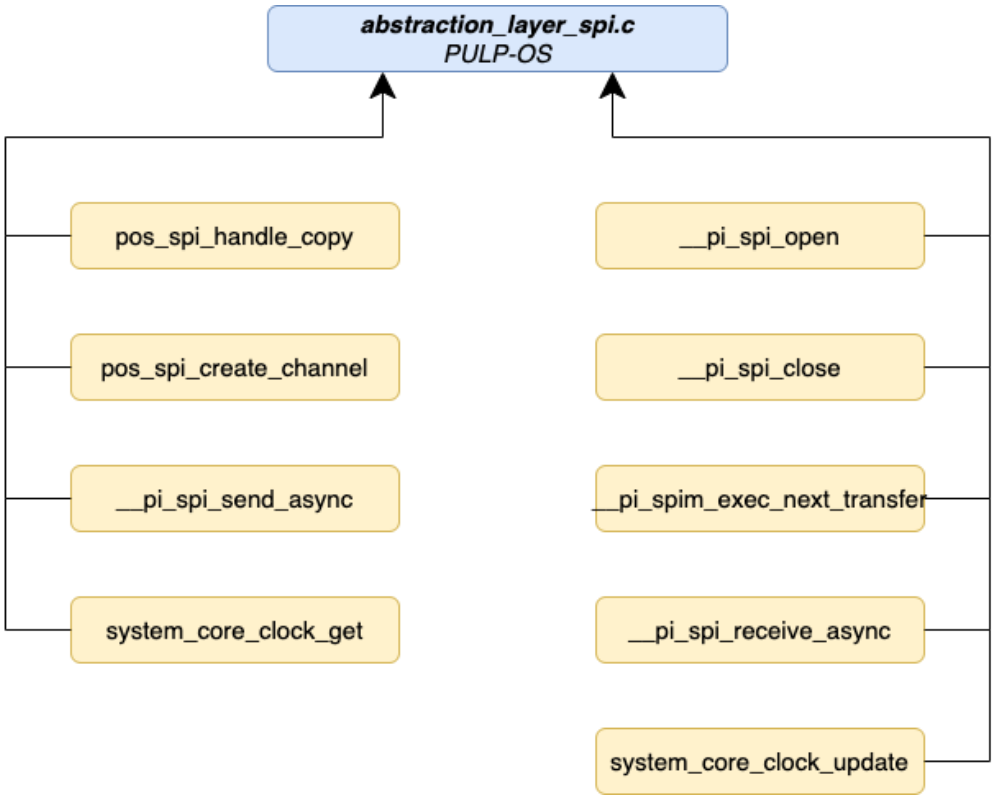

Figura 4.22: Schema del abstraction layer per SPI in PULP-OS

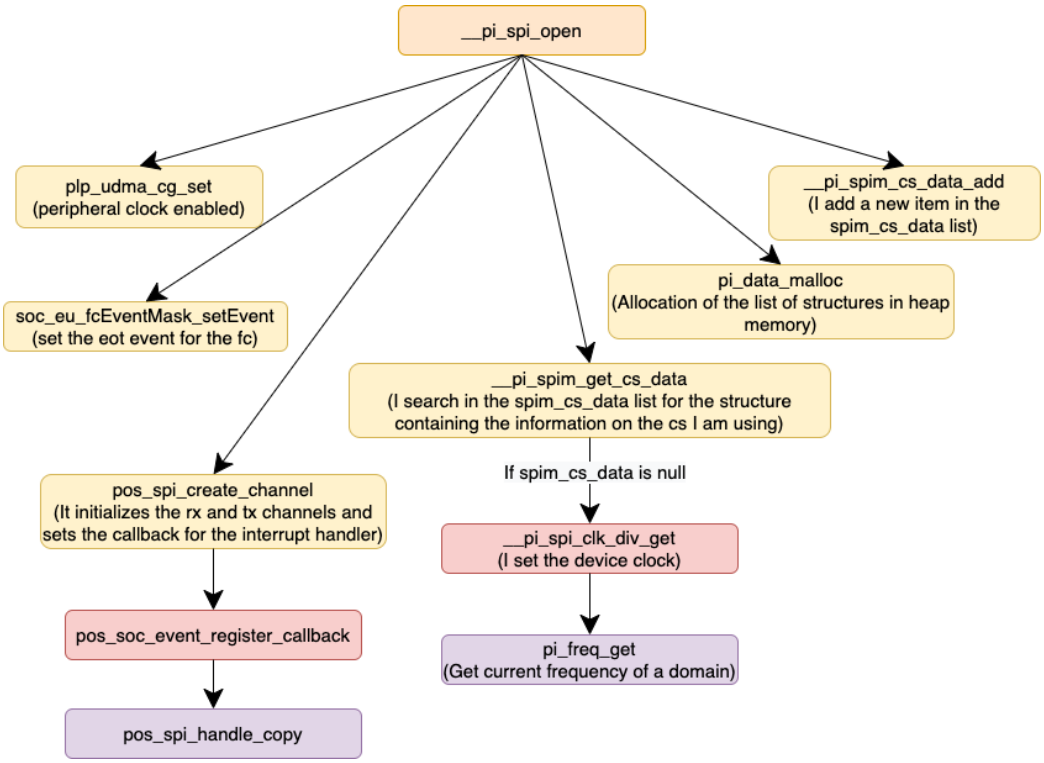

Figura 4.23: Schema PULP-OS per SPI di __pi_spi_open

La funzione __pi_spi_open, Fig. 4.23, serve per poter andare ad aprire l'interfaccia SPI. Questa funzione va inizialmente ad abilitare il clock per l'interfaccia SPI scelta. Successivamente va a creare un canale tx e uno rx, con la funzione pos_spi_create_channel. All'interno della funzione pos_spi_create_channel viene chiamata la procedura pos_soc_event_ register_callback. Quest'ultima in caso di evento, di tipo EOT, andra a fare la callback della funzione che gli si passa come parametro, cioè la pos_spi_handle _copy, la quale va ad eseguire ISR. Poi va ad abilitare la linea di interrupt associata agli eventi prodotti dal SoC e attiva l'evento associato al segnale di EOT della periferica SPI. Dopodiché si vanno ad allocare le strutture dati contenenti le informazioni per ogni interfaccia SPI e le infromazioni del dispositivo connesso al CS. Per allocare queste strutture, si utilizza la funzione pi_data_malloc, con la quale si va ad allocare memoria nel heap e

successivamente con la funzione __pi_spim_cs_data_add si aggiunge la struttura che contiene le informazioni del dispositivo connesso all'interfaccia SPI in una lista, mentre la struttra che descrive l'interfaccia SPI che si sta utilizzando viene salvata nel array __g_spim_drv_data.

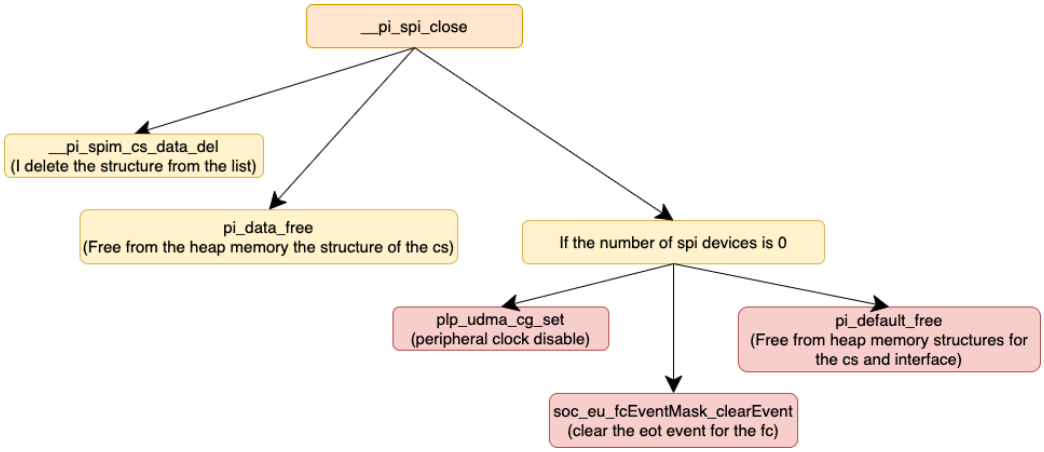

Figura 4.24: Schema PULP-OS per SPI di __pi_spi_close

La funzione __pi_spi_close, Fig. 4.24, va ad eliminare la struttura dati contenente le informazioni sul dispositivo connesso al CS del interfaccia SPI che si sta utilizzando. Nel caso in cui non ci siano dispositivi connessi all'interfaccia SPI, allora viene disabilitato il clock della periferica di comunicazione SPI e ripulita la linea di interrupt. Infine vengo deallocate le risorse con la funzione pi_default_free.

La funzione __pi_spi_send_async, Fig. 4.25, va prima a riempire il buffer di comandi cs_data->udma_cmd[ ], dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.4.3. Dopodiché va a controllare se è possibile fare l'invio dei dati oppure no. Il canale tx, definito precedentemente nella funzione __pi_spi_open, è composto da due pendings. Allora la funzione __pi_spi_send_async controlla quale pending del canale tx è libero, ed associa ad esso il trasferimento, quindi il task. Successivamente si utilizza la funzione plp_udma_enqueue per inviare sia il buffer di comandi che l'indirizzo della partizione di memoria in L2 dove si vogliono andare a

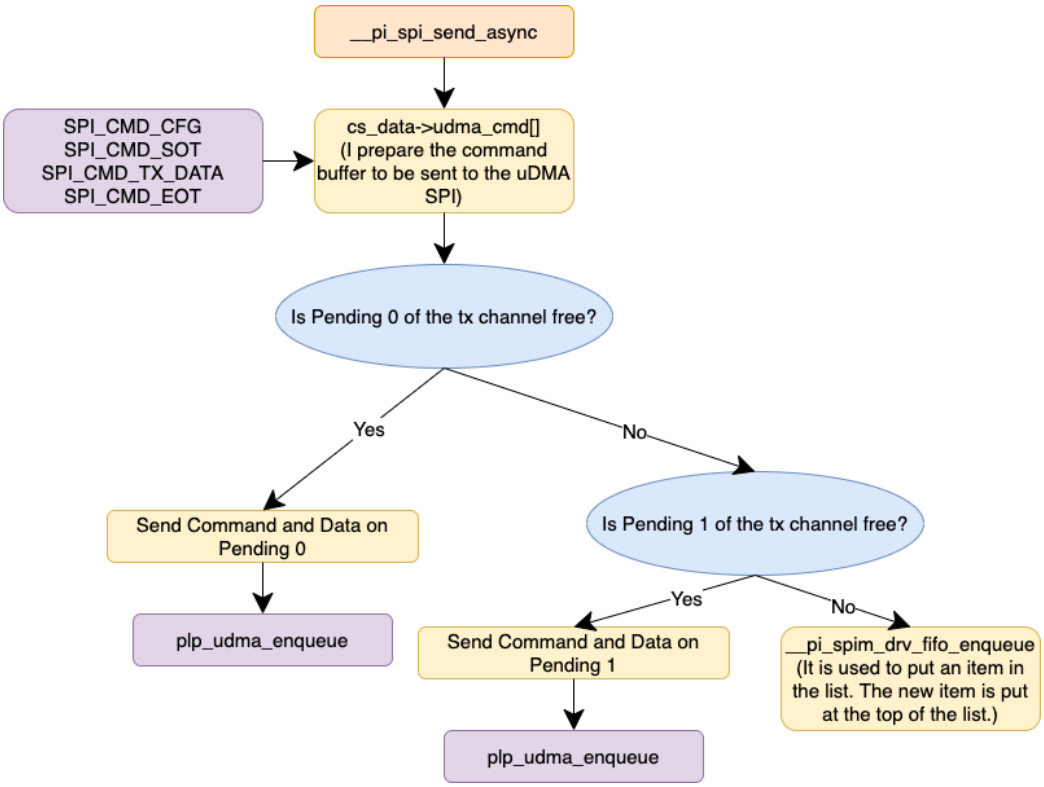

Figura 4.25: Schema PULP-OS per SPI di __pi_spi_send_async

prelevare i dati da inviare. Nel caso in cui entrambi i pendings del canale tx siano occupati, allora il trasferimento viene messo in coda in una FIFO software, questo con la funzione __pi_spim_drv_fifo_enqueue.

La funzione __pi_spi_receive_async, Fig. 4.26, va prima a riempire il buffer di comandi cs_data->udma_cmd[ ], dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.4.3. Dopodiché va a controllare se è possibile fare la ricezione dei dati oppure no. Il canale rx, definito precedentemente nella funzione __pi_spi_open, è composto da due pendings. Allora la funzione __pi_spi_receive_async controlla quale pending del canale rx è libero, ed associa ad esso il trasferimento, quindi il task. Successivamente si utilizza la funzione plp_udma_enqueue per inviare il buffer di comandi e l'indirizzo della partizione di memoria in L2 dove si vogliono andare a collo-

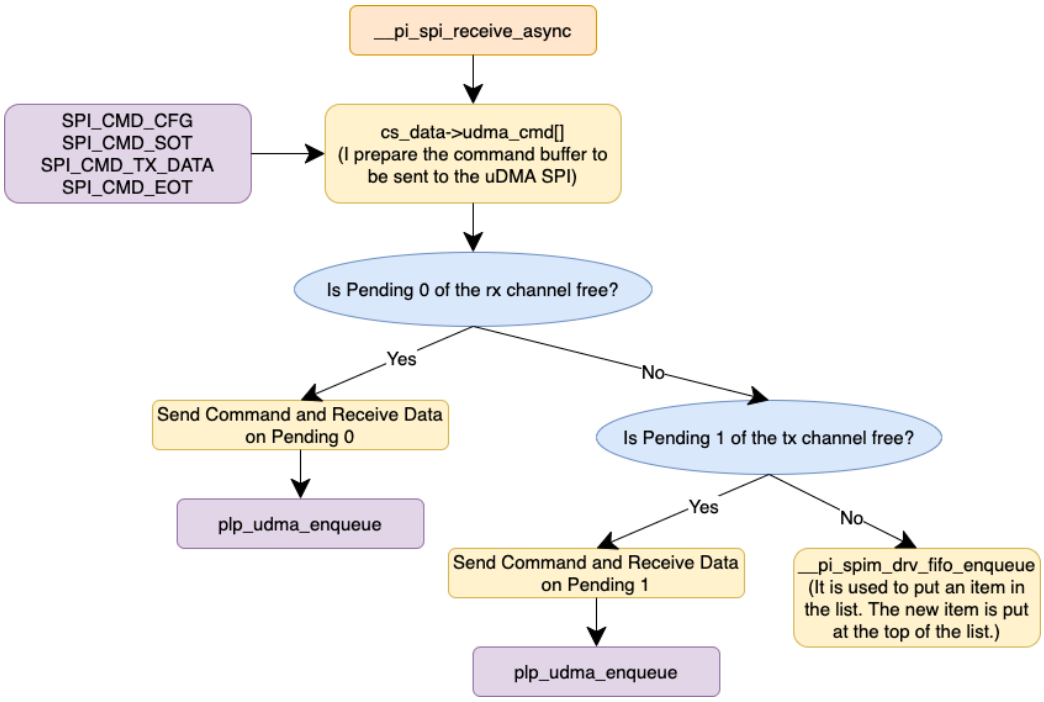

Figura 4.26: Schema PULP-OS per SPI di __pi_spi_receive_async

care i dati ricevuti. Nel caso in cui entrambi i pendings del canale rx siano occupati, allora il trasferimento viene messo in coda in una FIFO software, questo con la funzione __pi_spim_drv_fifo_enqueue.

La funzione pos_spi_handle_copy, Fig. 4.27, rappresenta ISR per l'evento di EOT prodotto dalla periferica di comunicazione SPI dopo aver eseguito un trasferimento. Quello che fa la funzione pos_spi_handle_copy è liberare il pendig uno del canale tx o rx. Poi controlla se ha qualche trasferimento in coda nella FIFO software. Nel caso in cui ci sia qualche trasferimento in coda, allora viene chiamata la funzione __pi_spim_exec_next_transfer, la quale si occuperà di eseguire il nuovo trasferimento. Infine chiama la funzione pos_task_push_locked, la quale servirà per poter associate il task alla funzione pi_task_wait_on, che si occuperà di rilasciare il task.

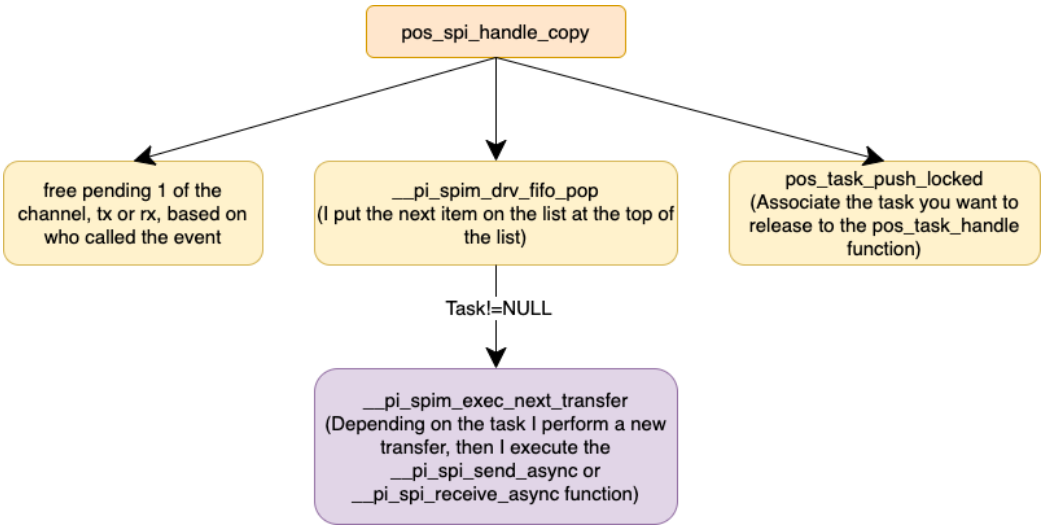

Figura 4.27: Schema PULP-OS per Interrupt SPI

4.2.3 Utilizzo del driver SPI su una memoria Flash

Di seguito vengono mostrati i test eseguiti per validare quelli che sono i driver per i due RTOS. I test realizzati sono di due tipi, si ha infatti il test spi_test_flash_sync, dove la sua caratteristica principale è quella di fare trasferimenti sincroni, e il test spi_test_flash_async, dove la sua caratteristica principale è quella di fare trasferimenti asincroni. Questi due test non cambiano nel caso si utilizzi FreeRTOS o PULP-OS. I test per funzionare utilizzeranno il file spim-v3.c, comune a entrambi gli RTOS. Si suppone di voler utilizzare l'interfaccia SPIM0 per andare a scrivere e leggere un buffer di dati su una memoria FLASH. La memoria Flash utilizzata in questo caso è la S25FS256S della Cypress. I comandi utilizzati per poter programmare questa memoria sono:

- CMD_WREN (0x06). Questo comando abilita la scrittura della memoria Flash.
- CMD_4PP (0x12). Questo comando serve per impostare la dimensione della pagina della memoria Flash che si vuole scrivere.

- CMD_RDSR1 (0x05). Questo comando va a leggere nel registro WIP, cioè Write-In-Progres. Allora se il WIP è 1, vuol dire che la Flash sta ancora eseguendo l'operazione di scrittura o formattazione, se invece il WIP è 0, allora la memoria Flash è nella modalità stand-by e può accettare comandi.

- CMD_4READ (0x13). Questo comando abilita la lettura della memoria Flash.

Sector Size (KB)	Sector Count	Sector Range	Address Range (Byte Address)	Notes
4	8	SA00	00000000h-00000FFFh	Sector Starting Address — Sector Ending Address
		:	:	
		SA07	00007000h-00007FFFh	
32	1	SA08	00008000h-0000FFFFh	
64	511	SA09	00010000h-0001FFFFh	
		:	:	
		SA519	01FF0000h-01FFFFFFh	

Figura 4.28: S25FS256S Sector Address Map

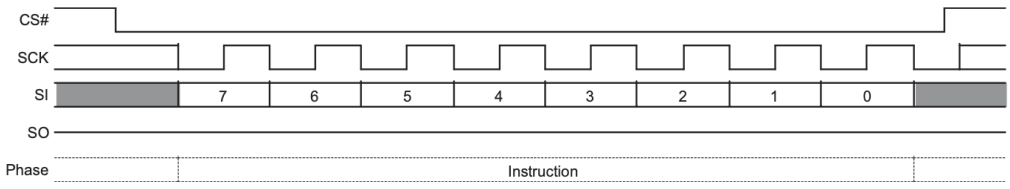

Figura 4.29: Write Enable (WREN) Command Sequence

La mappa dei vari indirizzi di memoria della Flash è mostrata in Fig. 4.28. Da protocollo, la memoria Flash per poter eseguire l'operazione di scrittura, vuole che gli si invii il comando CMD_WREN, Fig. 4.29, e successivamente il comando CMD_4PP seguito dall'indirizzo della partizione di memoria da scrivere e dal buffer di dati che si vuole scrivere, come mostrato in Fig. 2.20.

Da protocollo, la memoria Flash per poter eseguire il controllo del registro RDSR1, e quindi del valore del WIP, vuole che prima gli si invii il comando

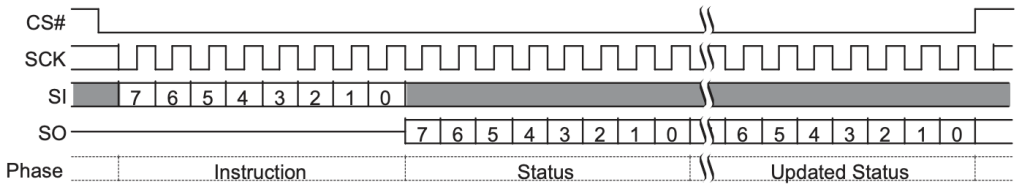

Figura 4.30: Read Status Register 1 (RDSR1) Command Sequence

CMD_RDSR1 e successivamente che si vada a leggere il valore del registro, come mostrato in Fig. 4.30.

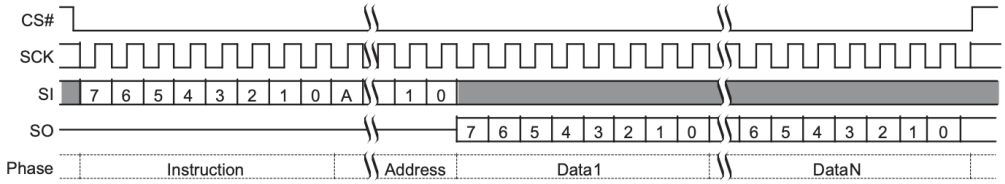

Figura 4.31: Read Command Sequence

Da protocollo, la memoria Flash per poter eseguire l'operazione di lettura, vuole che gli si invii il comando CMD_4READ seguito dall'indirizzo della partizione di memoria da leggere. Dopodiché la memoria invierà sulla linea MISO dell'interfaccia SPI i dati, come mostrato in Fig. 4.31.

Di seguito viene mostrato il flusso del codice nel caso del test spi_test_flash_sync, Fig. 4.32. Con la funzione pi_spi_conf_init si va ad associare, alla struttura dati che descrive l'interfaccia SPI, la configurazione iniziale dei suoi parametri. Con la funzione pi_open_from_conf si associa alla struttura device, la configurazione dell'interfaccia SPI. Con la funzione pi_spi_open si va ad inizializzare l'interfaccia SPI. Dopo questa prima parte di configurazioni, si ha il blocco di istruzioni con cui si vuole andare a scrivere sulla memoria Flash il buffer di dati. Per farlo si utilizzerà la funzione pi_spi_send, la cui sotto implementazione è diversa in base ai due RTOS, come mostrato in Fig. 4.7 e Fig. 4.8. Infatti le funzioni pi_task_block, pi_task_wait_on

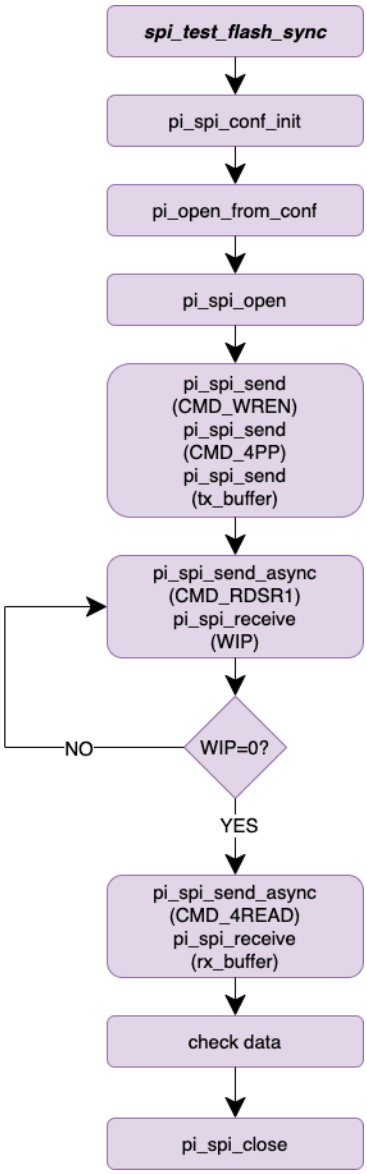

Figura 4.32: Diagramma di flusso test spi_test_flash_sync

e pi_spi_send_async avranno implementazioni diverse nei due RTOS. Una volta scritta la memoria bisogna aspettare che il registro WIP cambi il suo stato da 1, Flash occupata, a 0, Flash libera. Allora fino a quando il valore del WIP non è 0, bisogna aspettare che cambi il suo stato, questo si fa con un metodo a polling. Una volta completata l'operazione di scrittu-

ra sulla memoria Flash si può passare a leggerne il contenuto. Per farlo si utilizzerà la funzione pi_spi_receive, la cui sotto implementazione è diversa in base ai due RTOS, come mostrato in Fig. 4.9 e Fig. 4.10. Infatti le funzioni pi_task_block, pi_task_wait_on e pi_spi_receive_async avranno implementazioni diverse nei due RTOS. Dopodiché avverrà il check data, in cui si controlla che i valori ricevuti coincidano con quelli inviati. Infine si chiude l'interfaccia SPI con la funzione pi_spi_close. Queste istruzioni, cioè pi_spi_send e la pi_spi_receive, sono di tipo sincrono, quindi il processore aspetterà l'event task di ognuna di queste istruzioni prima di passare alla prossima, come descritto nel capitolo 3.3 e 3.4.

Di seguito viene mostrato il flusso del codice nel caso del test spi_test_flash_async, Fig. 4.33. Con la funzione pi_spi_conf_init si va ad associare, alla struttura dati che descrive l'interfaccia SPI, la configurazione iniziale dei suoi parametri. Con la funzione pi_open_from_conf si associa alla struttura device, la configurazione dell'interfaccia SPI. Con la funzione pi_spi_open si va ad inizializzare l'interfaccia SPI. Dopo questa prima parte di configurazioni, si ha il blocco di istruzioni nel quale si vanno a inizializzare i task, questo andando a utilizzare la struttura pi_task_t e la funzione pi_task_block. Il numero di task utilizzati saranno uguali al numero di funzioni di tipo pi_spi_send_async e pi_spi_receive_async utilizzate. Successivamente si vuole andare a scrivere sulla memoria Flash il buffer di dati. Per farlo si utilizzerà la funzione pi_spi_send_async, la cui sotto implementazione è diversa in base ai due RTOS, come mostrato in Fig. 4.19 e Fig. 4.25. Una volta scritta la memoria bisogna aspettare che il registro WIP cambi il suo stato da 1, Flash occupata, a 0, Flash libera. Allora fino a quando il valore del WIP non è 0, bisogna aspettare che cambi il suo stato, questo si fa con un metodo a polling. Una volta completata l'operazione di scrittura sulla memoria Flash si può passare a leggerne il contenuto. Per farlo si utilizzerà la funzione pi_spi_receive_async, la cui sotto implementazione è diversa in base ai due RTOS, come mostrato in Fig. 4.20 e Fig. 4.26. Alla fine con la funzione pi_task_wait_on, si aspetta di ricevere l'event task per ogni trasfe-

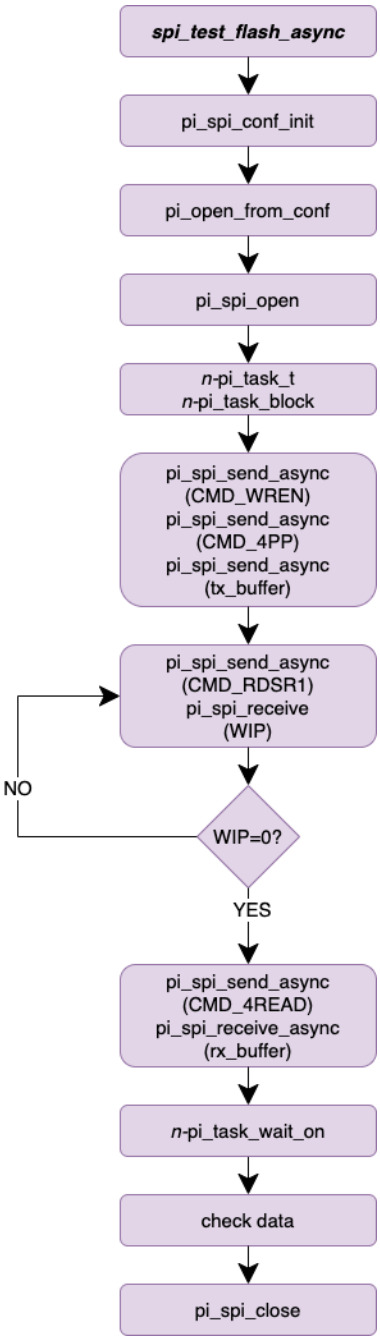

Figura 4.33: Diagramma di flusso test spi_test_flash_async

rimento asincrono fatto. Dopodiché avverrà il check data, in cui si controlla che i valori ricevuti coincidano con quelli inviati. Infine si chiude l'interfaccia SPI con la funzione pi_spi_close. Le istruzioni pi_spi_send_async e la pi_spi_receive_async, sono di tipo asincrono, quindi il processore non aspetta l' event task per ogni trasferimento ma solo alla fine verificherà, con la funzione pi_task_wait_on, di averli ricevuti tutti, come descritto nel capitolo 3.3 e 3.4.

4.3 Sviluppo dei driver I2C per la piattaforma PULP

Il driver per la periferica di comunicazione I^2C ha una struttura come quella mostrata in Fig. 4.4. Il driver per funzionare sfrutta quelle che sono le PMSIS. All'interno delle PMSIS c'è header file, i2c.h, Fig. 4.34, che contiene le dichiarazioni delle funzioni che devono essere implementate nel driver, Fig. 4.35. Come si può vedere dalla Fig. 4.35, molte funzioni, come la pi_i2c_open, hanno delle sotto-implementazioni, ad esempio __pi_i2c_open. Quest'ultime saranno implementate in modo diverso in base al RTOS che si sta utilizzando.

Di seguito vengono descritte le funzioni di Fig. 4.34.

La funzione pi_i2c_conf_init serve per impostare i valori predefiniti della struttura che descrive la periferica di comunicazione I^2C.

La funzione pi_i2c_open deve essere chiamata prima di poter utilizzare un interfaccia I^2C. Questa funzione farà tutta la configurazione necessaria per rendere l'interfaccia I^2C utilizzabile. Il chiamante verrà bloccato fino al termine dell'operazione. L' implementazione di questa funzione sarà diversa a seconda del RTOS utilizzato. Questa funzione restituisce 0 se l'operazione va a buon fine, -1 se si è verificato un errore.

La funzione pi_i2c_close si utilizza per chiudere un'interfaccia I^2C precedentemente aperta, tramite la funzione pi_i2c_open, in modo da liberare le risorse allocate. Una volta che questa funzione viene usata su un'inter-

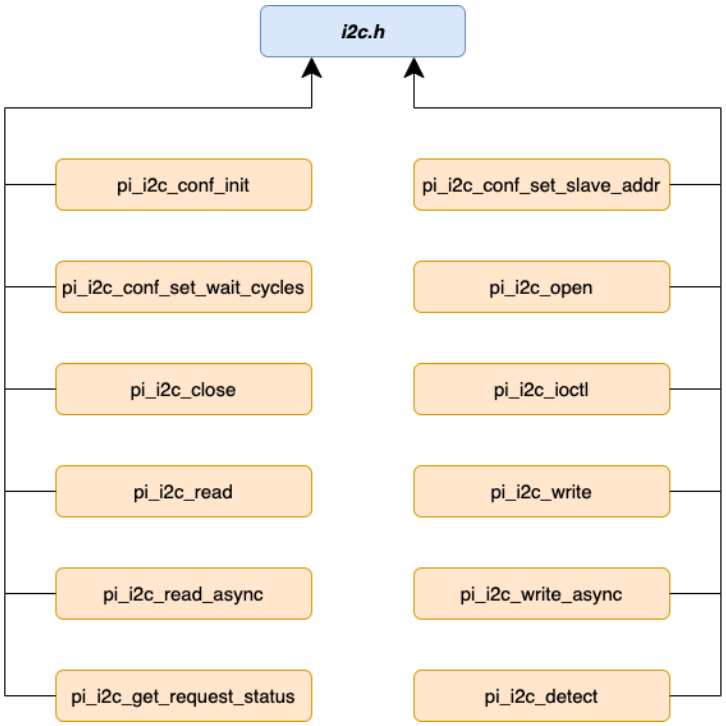

Figura 4.34: Dichiarazione delle API per I^2C

faccia I^2C, per poter riutilizzare l'interfaccia bisogna riaprirla nuovamente. L' implementazione di questa funzione sarà diversa a seconda del RTOS utilizzato.

La funzione pi_i2c_ioctl si utilizza per cambiare parte della configurazione dell'interfaccia I^2C una volta che questa è stata aperta.

La funzione pi_i2c_write si utilizza per inviare dati alla periferica di comunicazione I^2C. Questa procedura effettuerà un trasfeirmento sincrono tra l'interfaccia I^2C e il chip di memoria. In questo caso il chiamante è bloccato fino alla fine del trasferimento. Le sotto implementazioni di questa funzione cambiano a seconda del RTOS scelto, come mostrato in Fig. 4.36 e Fig. 4.37. Questa funzione restituisce PI_OK se la richiesta ha esito positivo o PI_ERR_I2C_NACK se è stato ricevuto un NACK durante la richiesta.

La funzione pi_i2c_read si utilizza per ricevere dati dalla periferica di comunicazione I^2C. Questa procedura effettuerà un trasfeirmento sincrono

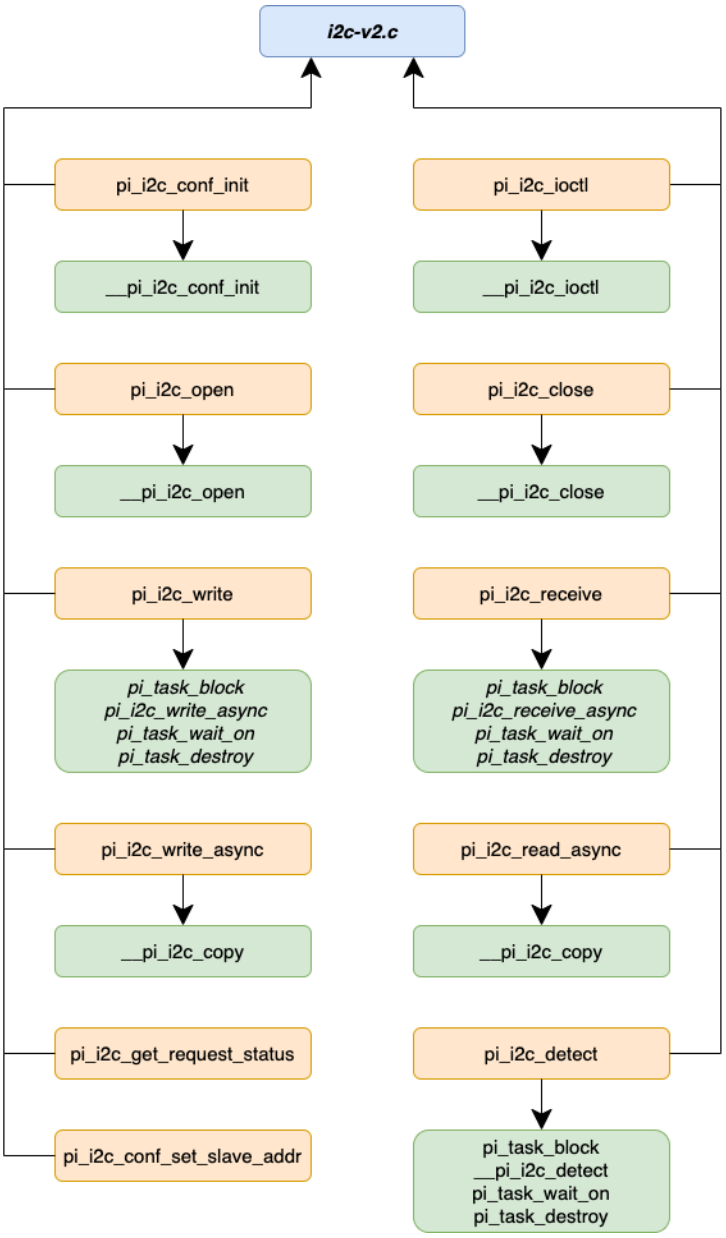

Figura 4.35: Definizione delle API per I^2C

tra l'interfaccia I^2C e il chip di memoria. In questo caso il chiamante è bloccato fino alla fine del trasferimento. Le sotto implementazioni di questa funzione cambiano a seconda del RTOS scelto, come mostrato in Fig. 4.38 e

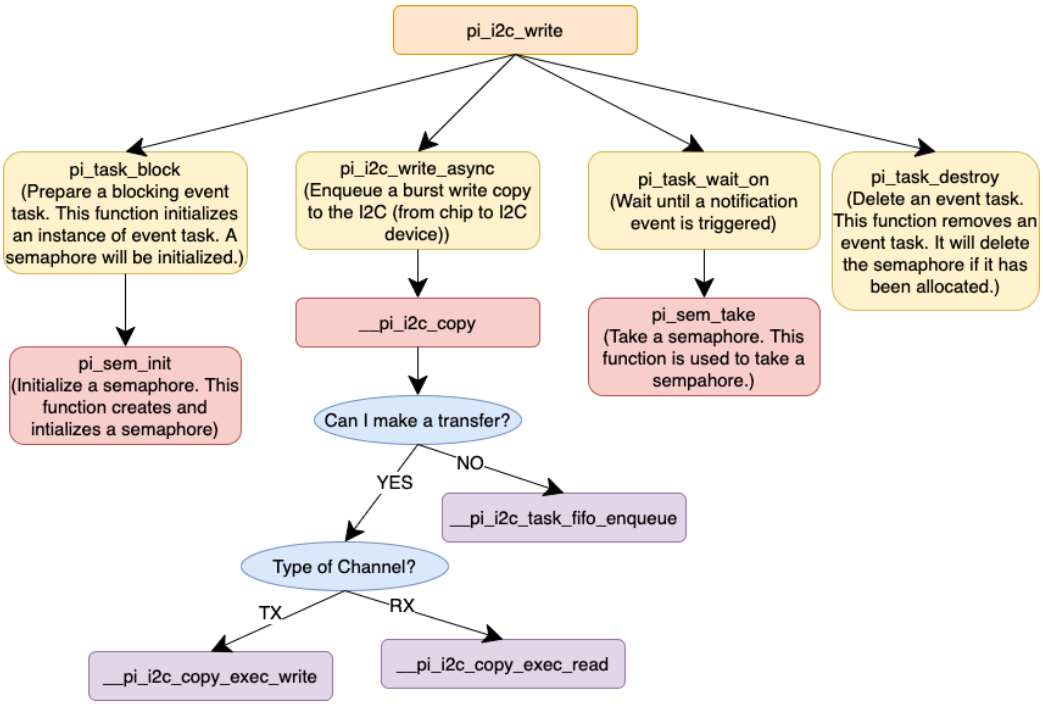

Figura 4.36: Schema pi_i2c_write per i2c in FreeRTOS

Fig. 4.39. Questa funzione restituisce PI_OK se la richiesta ha esito positivo o PI_ERR_I2C_NACK se è stato ricevuto un NACK durante la richiesta.

La funzione pi_i2c_write_async si utilizza per inviare dati alla periferica di comunicazione I^2C. Questa procedura effettuerà un trasfeirmento asincrono tra l'interfaccia I^2C e il chip di memoria. In questo caso il chiamante è bloccato fino alla fine del trasferimento e un task deve essere utilizzato per notificare la fine di un trasferimento al chiamante. L' implementazione di questa funzione sarà diversa a seconda del RTOS utilizzato.

La funzione pi_i2c_receive_async si utilizza per ricevere dati dalla periferica di comunicazione I^2C. Questa procedura effettuerà un trasfeirmento asincrono tra l'interfaccia I^2C e il chip di memoria. In questo caso il chiamante è bloccato fino alla fine del trasferimento e un task deve essere utilizzato per notificare la fine di un trasferimento al chiamante. L' implementazione di questa funzione sarà diversa a seconda del RTOS utilizzato.

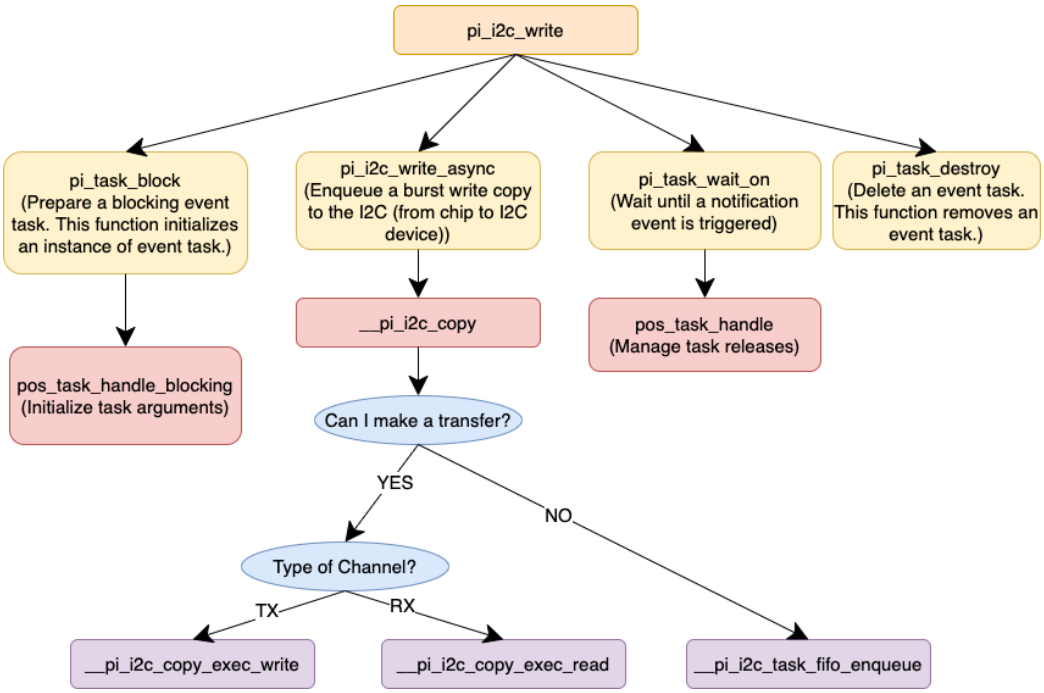

Figura 4.37: Schema pi_i2c_write per i2c in PULP-OS

La funzione pi_i2c_conf_set_slave_adr si utilizza per settare l'indirizzo del dispositivo connesso all'interfaccia I^2C.

La funzione pi_i2c_conf_set_wait_cycles si utilizza per impostare i wait cycles.

La funzione pi_i2c_get_request_status si utilizza per recuperare da un task lo stato di una richiesta.

La funzione pi_i2c_detect si utilizza per rilevare se un dispositivo è connessa al bus I^2C oppure no.

Con riferimento alla Fig. 4.40, al interno dei file common_i2c.h e common_i2c.c ci sono rispettivamente le dichiarazioni e le definizioni di funzioni e strutture dati comuni ai driver scritti per i due RTOS.

La funzione deactive_irq_i2c si utilizza per disattivare gli interrupt, mentre la funzione active_irq_i2c si utilizza per attivare gli interrupt. Queste due funzioni vengono usate per eseguire delle procedure in modo atomico,

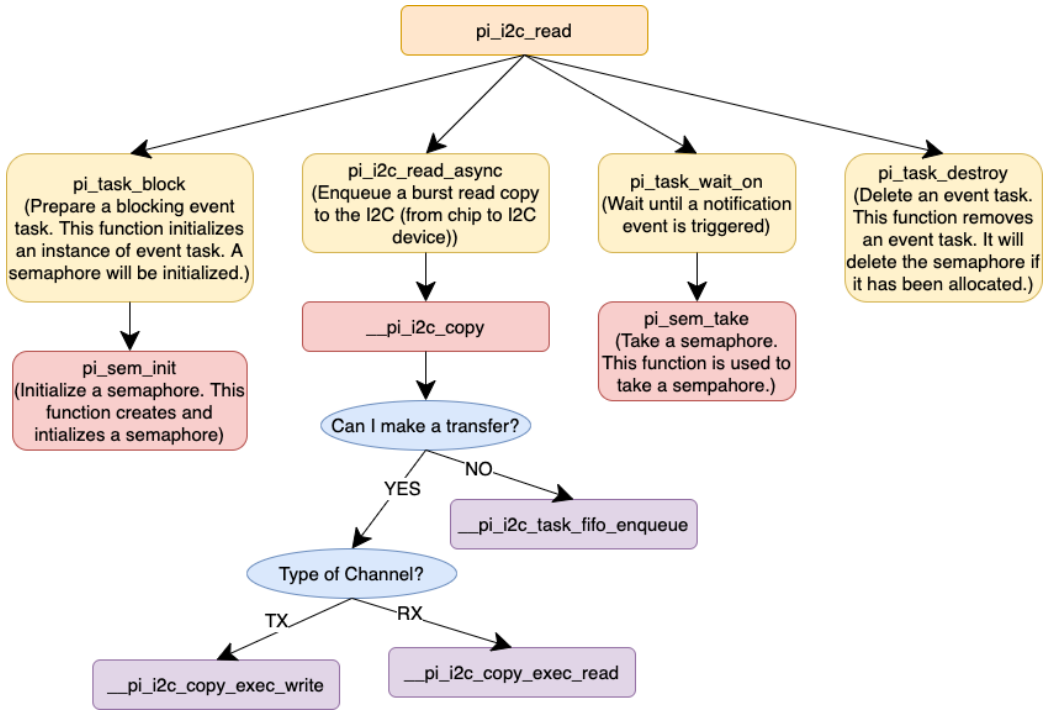

Figura 4.38: Schema pi_i2c_read per i2c in FreeRTOS

come ad esempio quelle che operano sulle liste.

La funzione __pi_i2c_handle_pending_transfer serve per gestire un trasferimento in sospeso dopo la fine della parte precedente del trasferimento.

La funzione __pi_i2c_cb_buf_empty serve per controllare se la FIFO HW del uDMA ha uno slot libero, e quindi può eseguire un trasferimento.

La funzione __pi_i2c_wait_transfer serve per controllare, con un metodo a polling, se il uDMA sta effettuando un trasferimento.

La funzione __pi_i2c_cb_buf_enqueue serve per mettere in coda un nuovo task nella FIFO hardware del uDMA. Al momento si ha un singolo slot.

La funzione __pi_i2c_cb_buf_pop restituisce il task presente nella FIFO hardware del uDMA e ne cancella il contenuto. Al momento si ha un singolo slot.

La funzione __pi_i2c_task_fifo_enqueue serve per inserire un task in

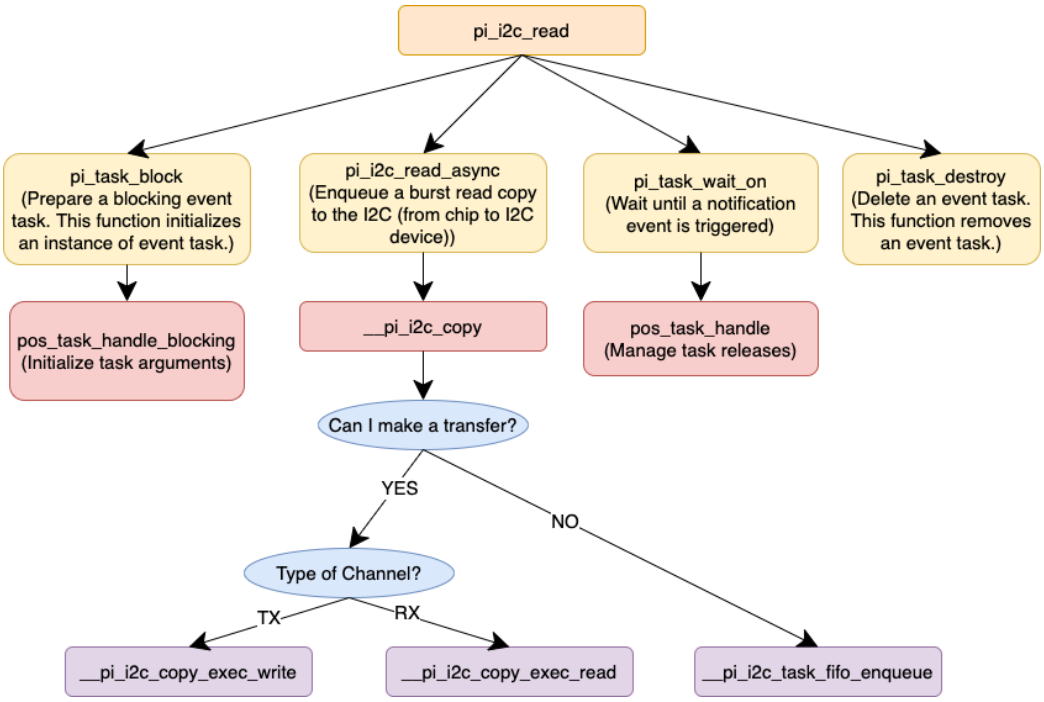

Figura 4.39: Schema pi_i2c_read per i2c in PULP-OS

una FIFO di tipo software, qualora il uDMA sia già occupato con un trasferimento.

La funzione __pi_i2c_task_fifo_pop serve per portare un task in cima alla FIFO software.

La funzione __pi_i2c_clk_div_get serve per dividere il clock.

La funzione __pi_i2c_cs_data_add si utilizza per inserire in una lista la struttura che descrive il dispositivo connesso all'interfaccia I^2C .

La funzione __pi_i2c_cs_data_remove si utilizza per cancellare da una lista la struttura che descrive il dispositivo connesso all'interfaccia I^2C .

La funzione __pi_i2c_freq_cb è una callback da eseguire quando cambia la frequenza.

La funzione __pi_i2c_baudrate_set serve per settare la baudrate dell'interfaccia I^2C aperta, una volta che è già stata inizializzata.

La funzione __pi_i2c_conf_init serve per impostare le caratteristiche

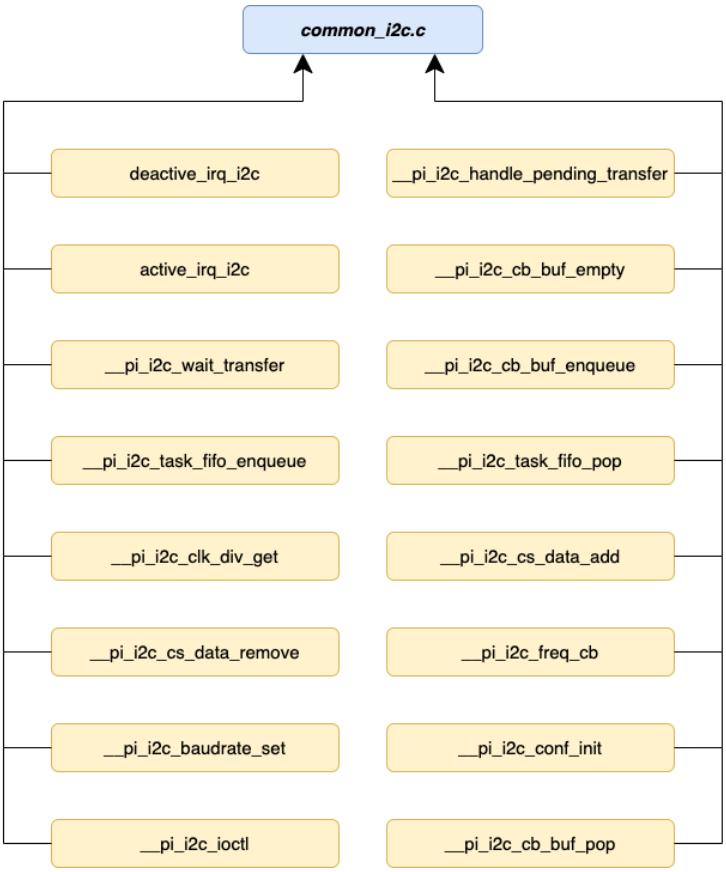

Figura 4.40: Schema del common file per I^2C

di default per l'interfaccia I^2C.

La funzione __pi_i2c_ioctl serve per modificare i valori dell'interfaccia I^2C dopo che è stata aperta.

La struttura dati i2c_cs_data_s, Fig. 4.41, contiene le informazioni del dispositivo connesso all'interfaccia I^2C.

La struttura dati i2c_itf_data_s si definisce in modo diverso nei casi in cui si utilizzi FreeRTOS, Fig. 4.42, o PULP-OS, Fig. 4.43, in entrambi i casi contiene le informazioni dell'interfaccia I^2C.

La struttura dati i2c_pending_transfer_s, Fig. 4.44, contiene le informazioni sul trasferimento in corso.

```
struct i2c_cs_data_s {
        uint8_t device_id;
        uint8_t cs;
        uint16_t clk_div;
        uint32_t max_baudrate;
        struct i2c_cs_data_s *next;
};
```

Figura 4.41: Struttura i2c_cs_data_s

```
struct i2c_itf_data_s {
        struct pi_task *buf[2];
        struct pi_task *fifo_head;
        struct pi_task *fifo_tail;
        struct i2c_pending_transfer_s *pending;
        uint32_t nb_open;

        uint32_t i2c_cmd_index;
        struct i2c_cs_data_s *cs_list;
        uint32_t i2c_cmd_seq[__PI_I2C_CMD_BUFF_SIZE];
        uint8_t i2c_stop_send;

        uint32_t i2c_stop_seq[__PI_I2C_STOP_CMD_SIZE];
        uint8_t i2c_eot_send;

        uint32_t *i2c_only_eot_seq;
        uint8_t device_id;

        uint8_t nb_events;
};
```

Figura 4.42: Struttura i2c_itf_data_s per FreeRTOS

```
struct i2c_itf_data_s{
        struct pi_task *buf[2];
        struct pi_task *fifo_head;
        struct pi_task *fifo_tail;
        struct i2c_pending_transfer_s *pending;
        uint32_t nb_open;

        uint32_t i2c_cmd_index;
        struct i2c_cs_data_s *cs_list;
        uint32_t i2c_cmd_seq[__PI_I2C_CMD_BUFF_SIZE];
        uint8_t i2c_stop_send;

        uint32_t i2c_stop_seq[__PI_I2C_STOP_CMD_SIZE];
        uint8_t i2c_eot_send;

        uint32_t *i2c_only_eot_seq;

        uint8_t device_id;

        uint8_t nb_events;
        pos_udma_channel_t *rx_channel;
        pos_udma_channel_t *tx_channel;
};
```

Figura 4.43: Struttura i2c_itf_data_s per PULP-OS

4.3.1 Abstraction Layer I^{2}C in FreeRTOS

L' abstraction layer per il driver I^{2}C in FreeRTOS utilizza le funzioni di Fig. 4.45. Di seguito verranno mostrate le implementazioni per queste funzioni.

La funzione __pi_i2c_open, Fig. 4.46, serve per poter andare ad aprire l'interfaccia I^{2}C. Questa funzione va inizialmente ad abilitare il clock per l'interfaccia I^{2}C scelta. Successivamente va ad abilitare la linea di interrupt associata agli eventi prodotti dal SoC e poi attiva gli eventi associati ai

```
1        struct i2c_pending_transfer_s *pending;
2        uint32_t nb_open;

3        uint32_t i2c_cmd_index;
4        struct i2c_cs_data_s *cs_list;
5        uint32_t i2c_cmd_seq[__PI_I2C_CMD_BUFF_SIZE];
6        uint8_t i2c_stop_send;

7        uint32_t i2c_stop_seq[__PI_I2C_STOP_CMD_SIZE];
8        uint8_t i2c_eot_send;
```

Figura 4.44: Struttura i2c_pending_transfer_s

segnali di EOT, CMD e RX della periferica I^2C. Poi viene impostato un array, pi_fc_event_handler_set, contenente l'indirizzo della ISR da eseguire in base al evento settato. Allora questo array avrà come indice il numero dell'evento, EOT, RX e CMD, e come elemento l'indirizzo della ISR da eseguire. In questa fase viene settato l'array contenente la sequenza dei comandi da inviare per ottenere il segnale di EOT e per applicare lo STOP, cioè i2c_stop_seq. Dopodiché si vanno ad allocare le strutture dati contenenti le informazioni per ogni interfaccia I^2C e le informazioni del dispositivo connesso a questa interfaccia. Per allocare queste strutture, si utilizza la funzione pi_data_malloc, con la quale si va ad allocare memoria nel heap. Successivamente con la funzione __pi_i2c_cs_data_add si aggiunge la struttura che contiene le informazioni del dispositivo connesso all'interfaccia I^2C in una lista, mentre la struttra che descrive l'interfaccia I^2C che si sta utilizzando viene salvata nel array g_i2c_itf_data.

La funzione __pi_i2c_close, Fig. 4.47, va ad eliminare dalla lista la struttura dati contenente le informazioni sul dispositivo connesso all'interfaccia I^2C che si sta utilizzando. Nel caso in cui non ci siano dispositivi connessi all'interfaccia I^2C, allora viene disabilitato il clock della periferica

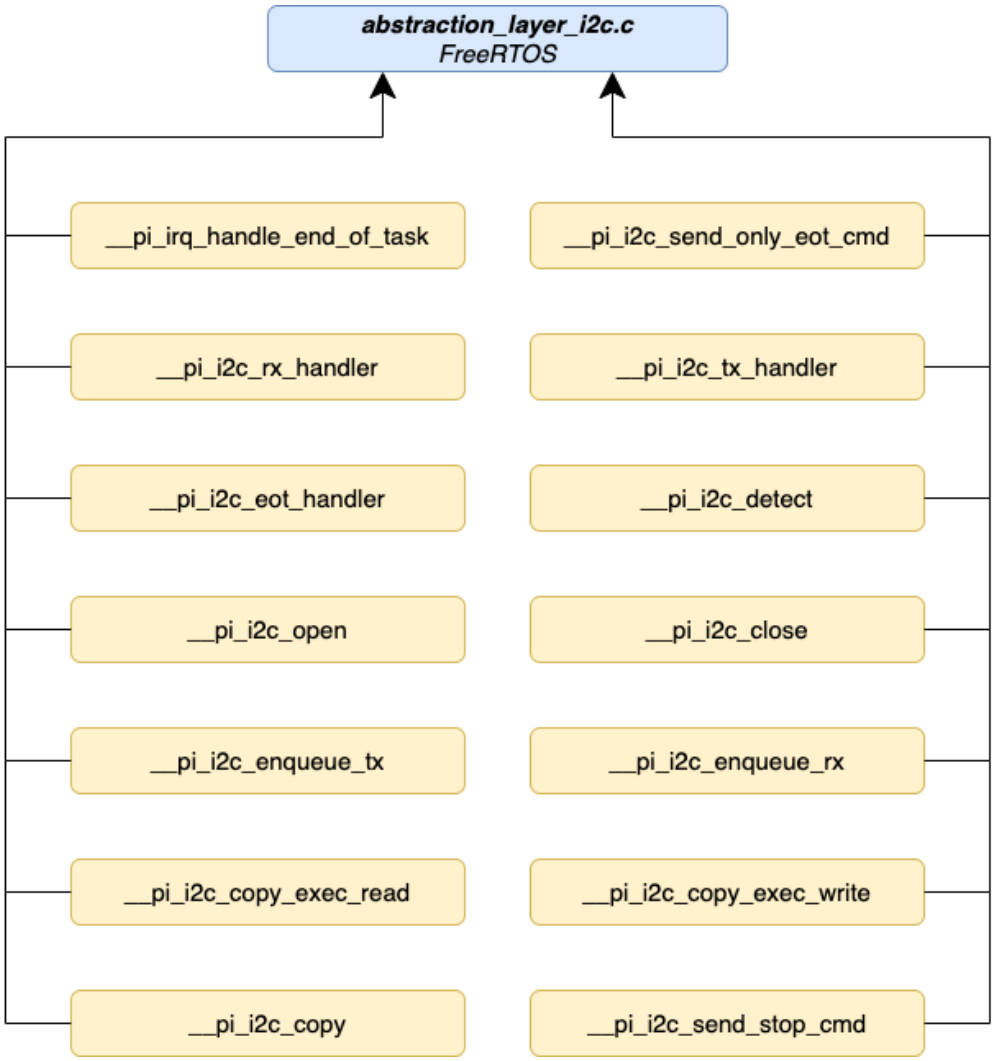

Figura 4.45: Schema del abstraction layer per I^2C in FreeRTOS

di comunicazione I^2C e ripulita la linea di interrupt. Infine vengo deallocate le risorse con la funzione pi_default_free.

La funzione __pi_i2c_copy viene chiamata a sua volta dalla funzione pi_i2c_write o pi_i2c_read. Quello che fa questa funzione è inserire nel array data della struttura pi_task_t i dati del trasferimento che si vuole fare, come l'indirizzo della memoria L2, l2_buff, che corrisponde all'indirizzo del buffer da inviare/ricevere, la dimensione del buffer, length, le informazioni

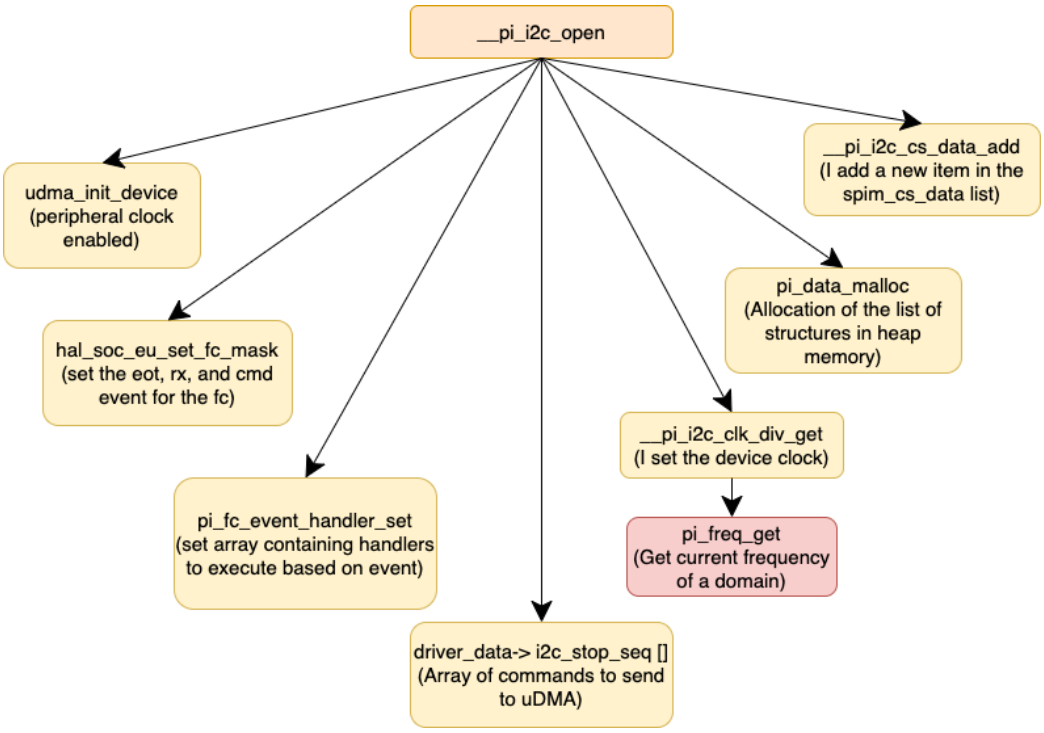

Figura 4.46: Schema Freertos per I^2C di __pi_i2c_open

sul trasferimento, flags, il canale da utilizzare, channel, che può essere TX o RX, e infine la struttura dati contenente le informazioni del dispositivo connesso all'interfaccia I^2C. Successivamente nel caso in cui il uDMA non abbia trasferimenti in corso, si controlla il tipo di trasferimento da fare, quindi se bisogna inviare o ricevere dati, e si chiama rispettivamente la funzione __pi_i2c_copy_exec_write o __pi_i2c_copy_exec_read. Nel caso in cui il uDMA sia occupato allora si inserisce il tasferimento in coda, attraverso la funzione __pi_i2c_task_fifo_enqueue.

La funzione __pi_i2c_copy_exec_write va inizialmente a riempire il buffer di comandi, driver_data->i2c_cmd_seq[], da inviare al uDMA per poter eseguire l'operazione di invio dei dati con l'interfaccia I^2C, dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.5.3. Successivamente si va a riempire il buffer buffer_to_write con i dati che si vogliono

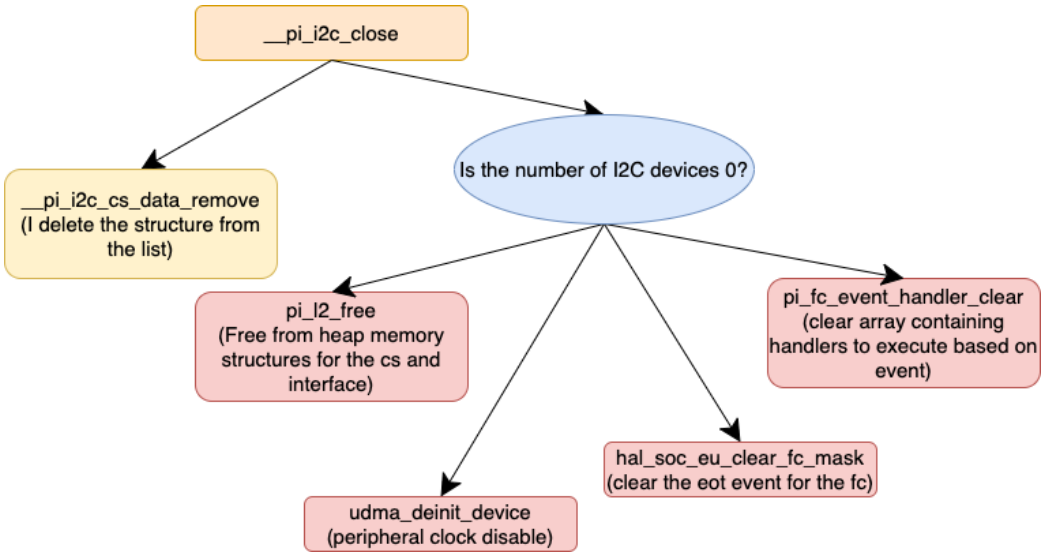

Figura 4.47: Schema Freertos per I^{2}C di __pi_i2c_clone

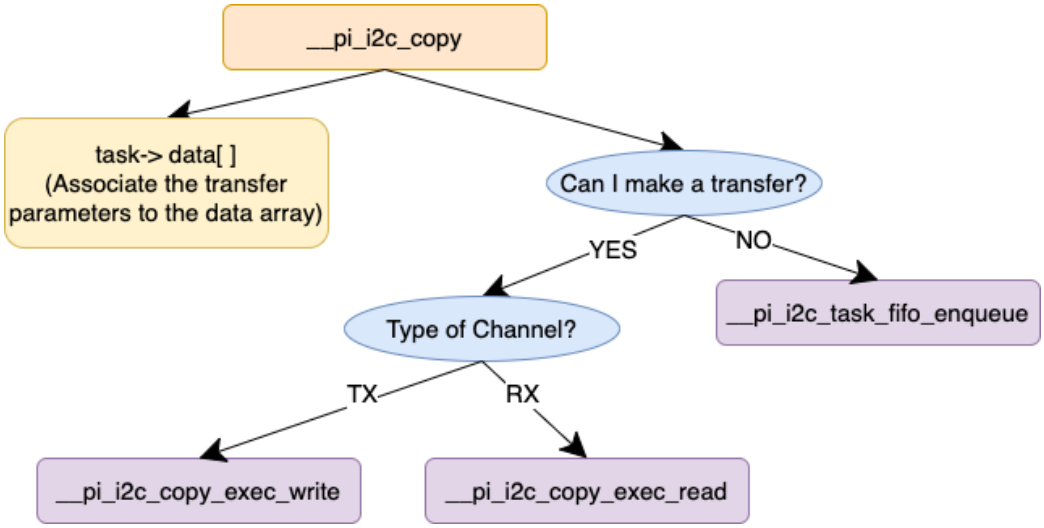

Figura 4.48: Schema Freertos per I^{2}C di __pi_i2c_copy

inviare. Dopodiché con la funzione __pi_i2c_enqueue_cmd si va prima a inviare il buffer di comandi e poi il buffer di dati.

La funzione __pi_i2c_copy_exec_read va inizialmente a riempire il buffer di comandi, driver_data->i2c_cmd_seq[], da inviare al uDMA per poter

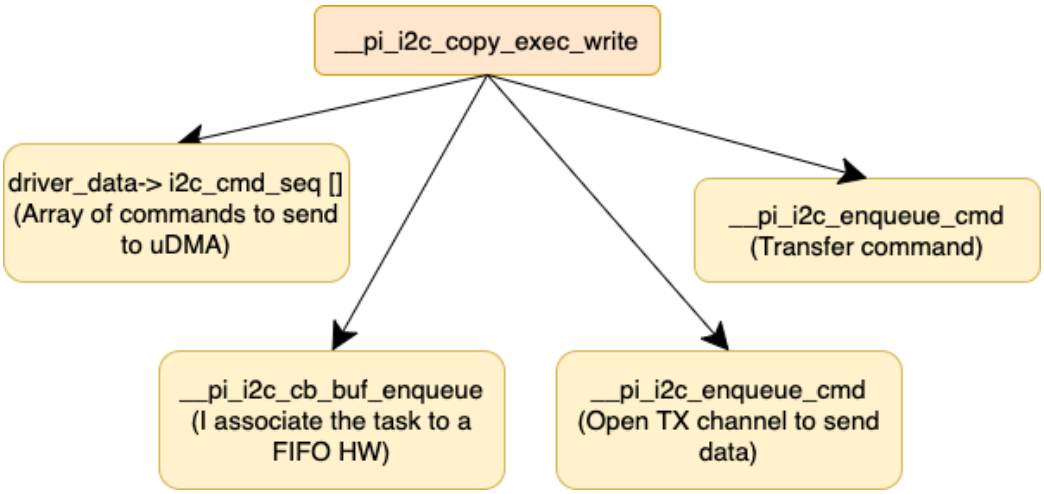

Figura 4.49: Schema Freertos per I^{2}C di __pi_i2c_copy_exec_write

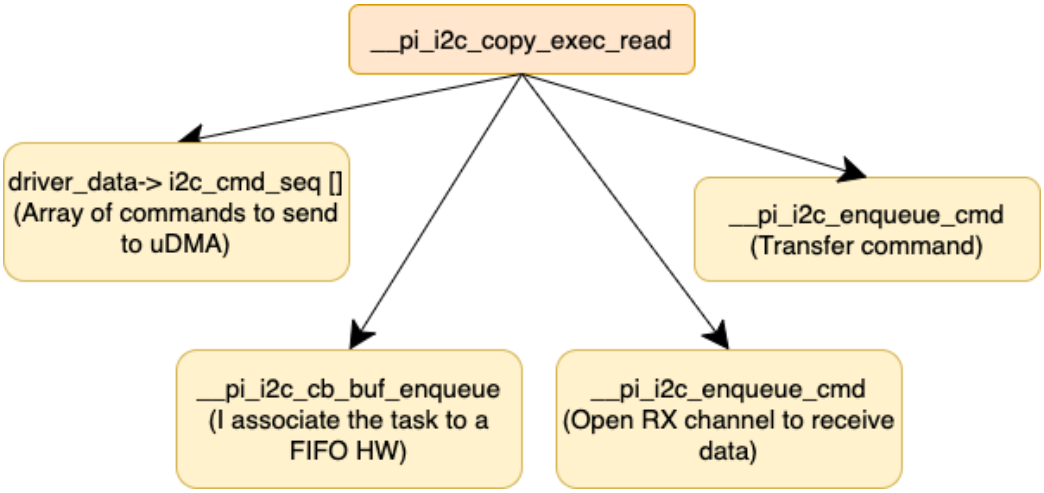

Figura 4.50: Schema Freertos per I^{2}C di __pi_i2c_copy_exec_read

eseguire l'operazione di ricezione dei dati dall' interfaccia I^{2}C, dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.5.3. Dopodiché con la funzione __pi_i2c_enqueue_cmd si va prima a inviare il buffer di comandi e successivamente con la funzione __pi_i2c_enqueue_rx si invia l'indirizzo della partizione di memoria in L2 dove si vogliono andare a collocare i dati ricevuti.

La funzione __pi_i2c_send_stop_cmd si utilizza per inviare il comando di EOT e il comando di STOP.

La funzione __pi_i2c_send_only_eot_cmd si utilizza per inviare solo il comando di STOP.

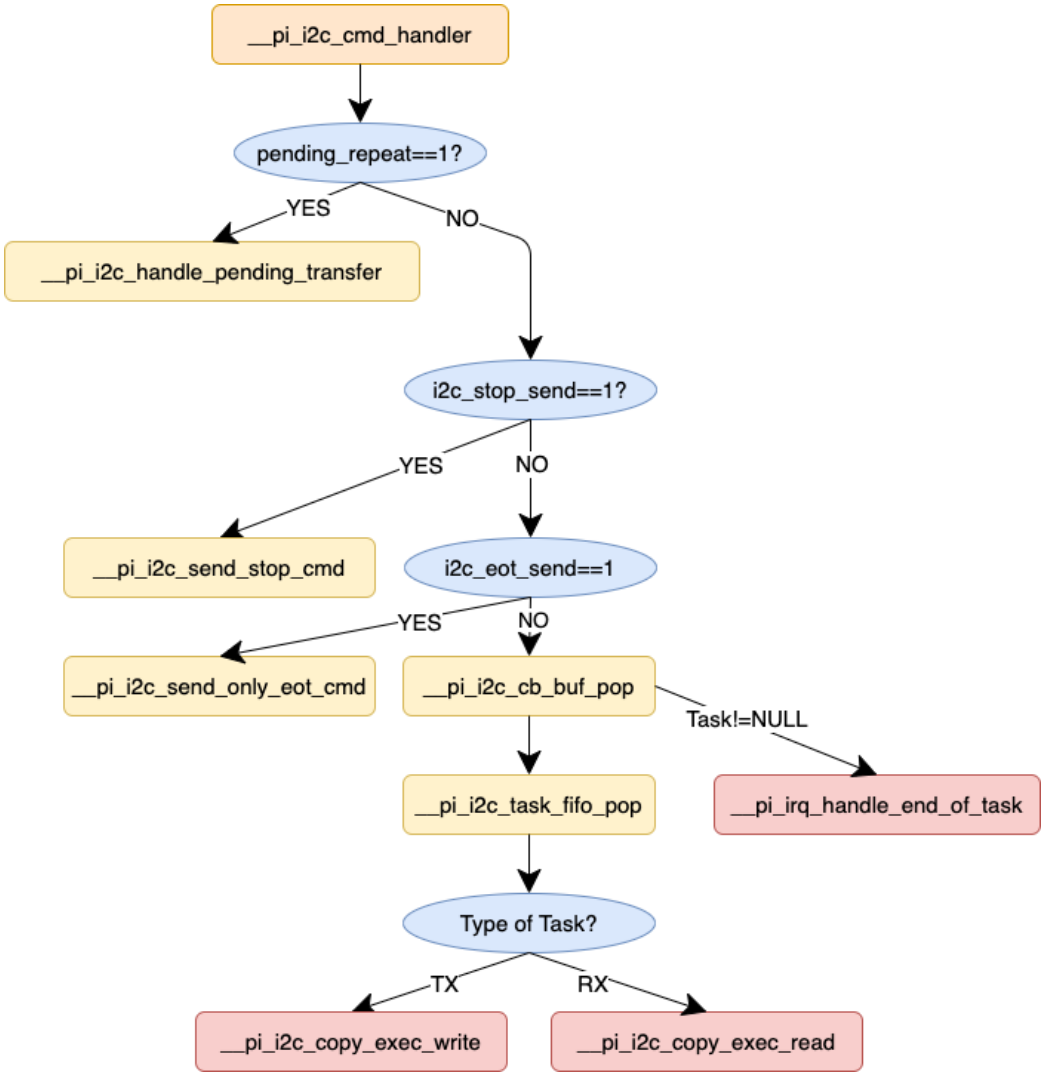

Figura 4.51: Schema Freertos per I^{2}C di __pi_i2c_cmd_handler

La funzione __pi_i2c_tx_handler, Fig. 4.51, rappresenta ISR che viene eseguita a seguito di un evento prodotto dall'invio dei comandi. In base ai flags che vengono settati nelle funzioni __pi_i2c_copy_exec_read e

__pi_i2c_copy_exec_write, questa procedura può chiamare diverse funzioni. Infatti nel caso in cui il trasferimento da eseguire abbia un bufer di dati troppo grande, allora viene settato il flag pending_repeat, e con la funzione __pi_i2c_handle_pending_transfer si esegue il trasferimento del buffer di dati in più transazioni. Nel caso in cui venga settato il flag i2c_stop_send allora viene eseguita la funzione __pi_i2c_send_stop_cmd, in caso contrario nel caso in cui __pi_i2c_cb_buf_pop contenga un task non nullo allora questo viene rilasciato con la funzione __pi_irq_handle_end_of_task. Successivamente si controlla se si ha qualche trasferimento in coda nella FIFO software. Nel caso in cui ci sia qualche trasferimento in coda, in base al tipo di trasferimento, viene chiamata la funzione __pi_i2c_copy_exec_read o __pi_i2c_copy_exec_write.

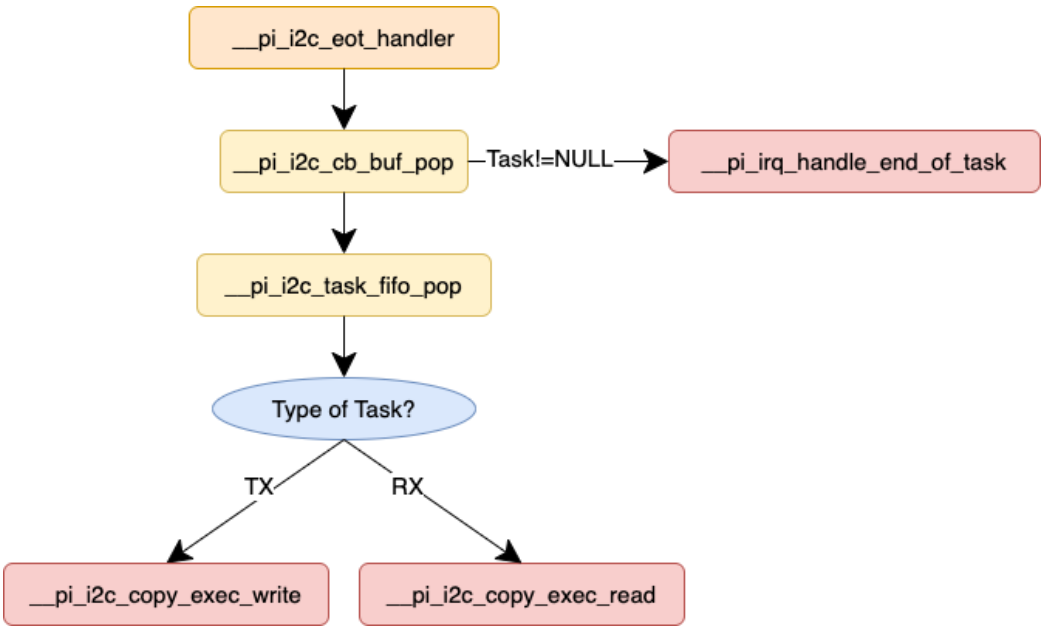

Figura 4.52: Schema Freertos per I^2C di __pi_i2c_eot_handler

La funzione __pi_i2c_eot_handler, Fig. 4.52, rappresenta ISR che viene eseguita a seguito di un evento prodotto dall'invio del comando di EOT. Questa procedura nel caso in cui __pi_i2c_cb_buf_pop contenga un task

non nullo allora questo viene rilasciato con la funzione __pi_irq_handle_end_of_task. Successivamente si controlla se si ha qualche trasferimento in coda nella FIFO software. Nel caso in cui ci sia qualche trasferimento in coda, in base al tipo di trasferimento, viene chiamata la funzione __pi_i2c_copy_exec_read o __pi_i2c_copy_exec_write.

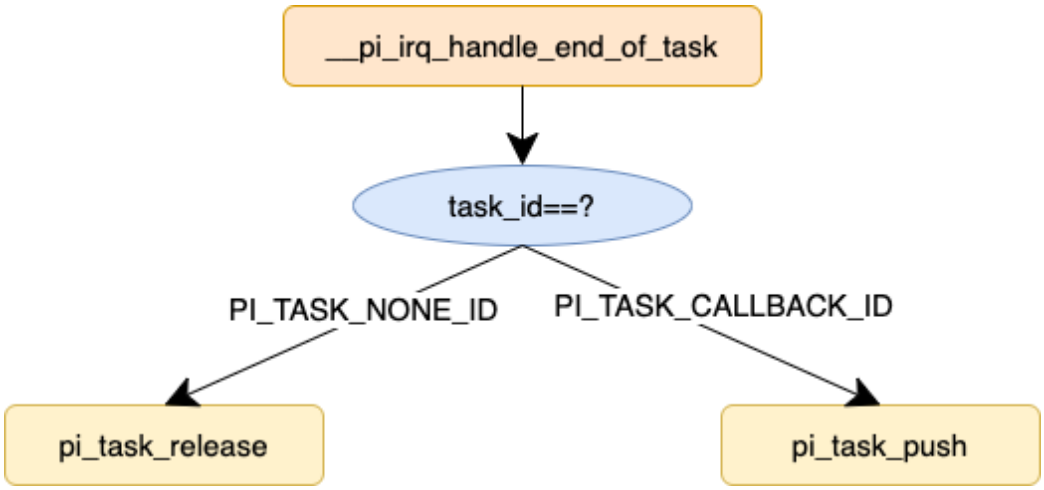

Figura 4.53: Schema Freertos per I^2C di __pi_irq_handle_end_of_task

La funzione __pi_irq_handle_end_of_task, Fig. 4.53, va a leggere id del task e in base a questo esegue la funzione pi_task_release oppure la funzione pi_task_push.

La funzione __pi_i2c_detect scansiona il bus i2c per rilevare i dispositivi collegati. Questa procedura va inizialmente a riempire il buffer di comandi, driver_data->i2c_cmd_seq[], da inviare al uDMA per poter eseguire l'operazione di ricezione dei dati dall' interfaccia I^2C, dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.5.3. Dopodiché con la funzione __pi_i2c_enqueue_cmd si va prima a inviare il buffer di comandi e successivamente con la funzione __pi_i2c_enqueue_rx si invia l'indirizzo della partizione di memoria in L2 dove si vuole andare a collocare il dato ricevuto.

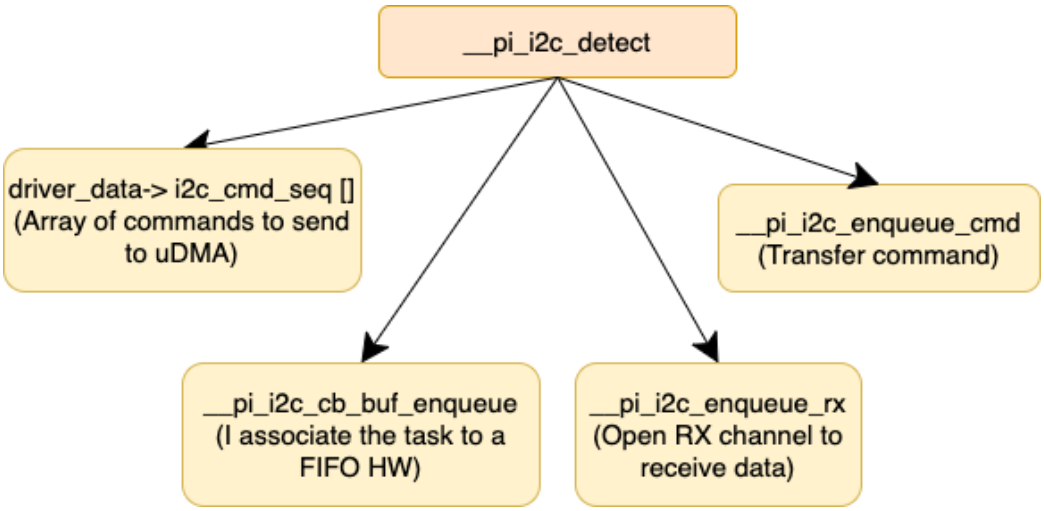

Figura 4.54: Schema Freertos per I^{2}C di __pi_i2c_detect

4.3.2 Abstraction Layer I^{2}C in PULP-OS

L' abstraction layer per il driver I^{2}C in PULP-OS utilizza le funzioni di Fig. 4.55. Di seguito verranno mostrate le implementazioni per queste funzioni.

La funzione __pi_i2c_open, Fig. 4.56, serve per poter andare ad aprire l'interfaccia I^{2}C. Questa funzione va inizialmente ad abilitare il clock per l'interfaccia I^{2}C scelta, con la funzione plp_udma_cg_set. Successivamente va a creare un canale tx e uno rx, con la funzione pos_i2c_create_channel. All'interno della funzione pos_i2c_create_channel viene chiamata la procedura pos_soc_event_ register_callback. Quest'ultima in caso di evento, di tipo EOT, CMD o RX andrà a fare la callback della funzione che gli si passa come parametro, cioè la pos_i2c_handle_copy, la quale va ad eseguire ISR. Poi va ad abilitare la linea di interrupt associata agli eventi prodotti dal SoC e attiva gli eventi associati al segnale di EOT, CMD e RX della periferica I^{2}C, con la funzione soc_eu_fcEventMask_setEvent. In questa fase viene settato l'array contenente la sequenza dei comandi da inviare per ottenere il segnale di EOT e per applicare lo STOP, cioè i2c_stop_seq[]. Dopodiché

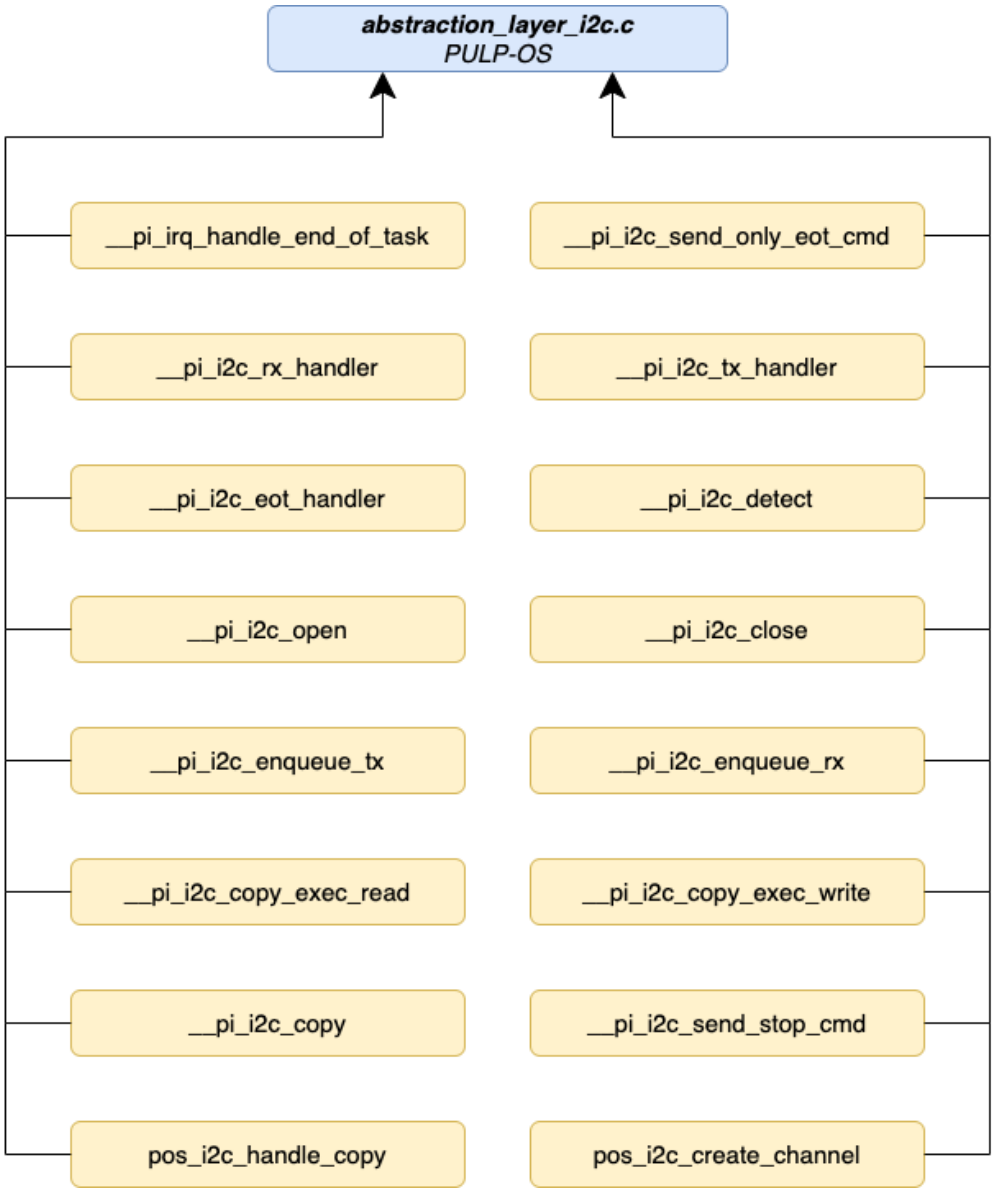

Figura 4.55: Schema del abstraction layer per I²C in PULP-OS

si vanno ad allocare le strutture dati contenenti le informazioni per ogni interfaccia I²C e le informazioni del dispositivo connesso a questa interfaccia. Per allocare queste strutture, si utilizza la funzione pi_l2_malloc, con la quale si va ad allocare memoria nel heap e successivamente con la funzione

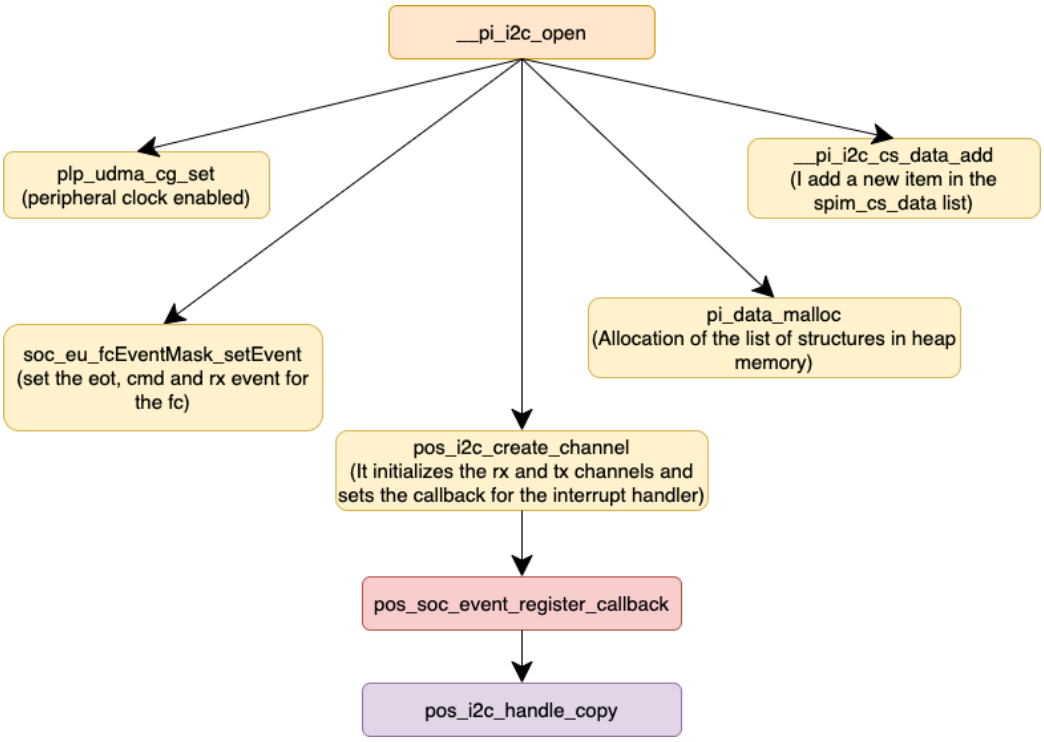

Figura 4.56: Schema PULP-OS per I^2C di __pi_i2c_open

__pi_i2c_cs_data_add si aggiunge la struttura che contiene le informazioni del dispositivo connesso all'interfaccia I^2C in una lista, mentre la struttra che descrive l'interfaccia I^2C che si sta utilizzando viene salvata nel array g_i2c_itf_data.

La funzione __pi_i2c_close, Fig. 4.57, va ad eliminare dalla lista la struttura dati contenente le informazioni sul dispositivo connesso all'interfaccia I^2C che si sta utilizzando. Nel caso in cui non ci siano dispositivi connessi all'interfaccia I^2C, allora viene disabilitato il clock della periferica di comunicazione I^2C, con plp_udma_cg_set, e ripulita la linea di interrupt, con la funzione soc_eu_fcEventMask_clearEvent. Infine vengo deallocate le risorse dalla memoria L2 con la funzione pi_l2_free.

La funzione __pi_i2c_copy viene chiamata a sua volta dalla funzione pi_i2c_write o pi_i2c_read. Quello che fa questa funzione è inserire nel

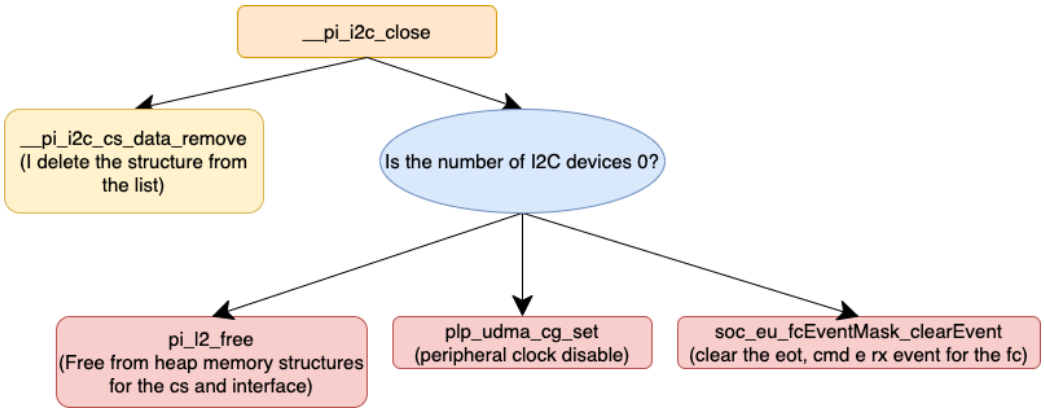

Figura 4.57: Schema PULP-OS per I^{2}C di __pi_i2c_clone

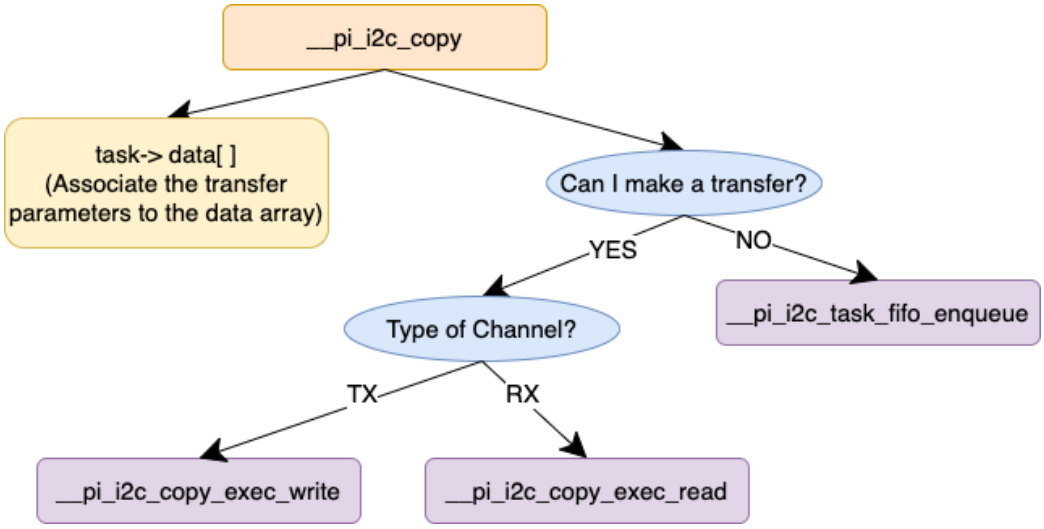

Figura 4.58: Schema PULP-OS per I^{2}C di __pi_i2c_copy

array data della struttura pi_task_t i dati del trasferimento che si vuole fare, come l'indirizzo della memoria L2, l2_buff, che corrisponde all'indirizzo del buffer da inviare/ricevere, la dimensione del buffer, length, le informazioni sul trasferimento, flags, il canale da utilizzare, channel, che può essere TX o RX, e infine la struttura dati contenente le informazioni del dispositivo connesso all'interfaccia I^{2}C. Successivamente nel caso in cui il uDMA non

abbia trasferimenti in corso, si controlla il tipo di trasferimento da fare, quindi se bisogna inviare o ricevere dati, e si chiama rispettivamente la funzione __pi_i2c_copy_exec_write o __pi_i2c_copy_exec_read. Nel caso in cui il uDMA sia occupato allora si inserisce il tasferimento in coda, attraverso la funzione __pi_i2c_task_fifo_enqueue.

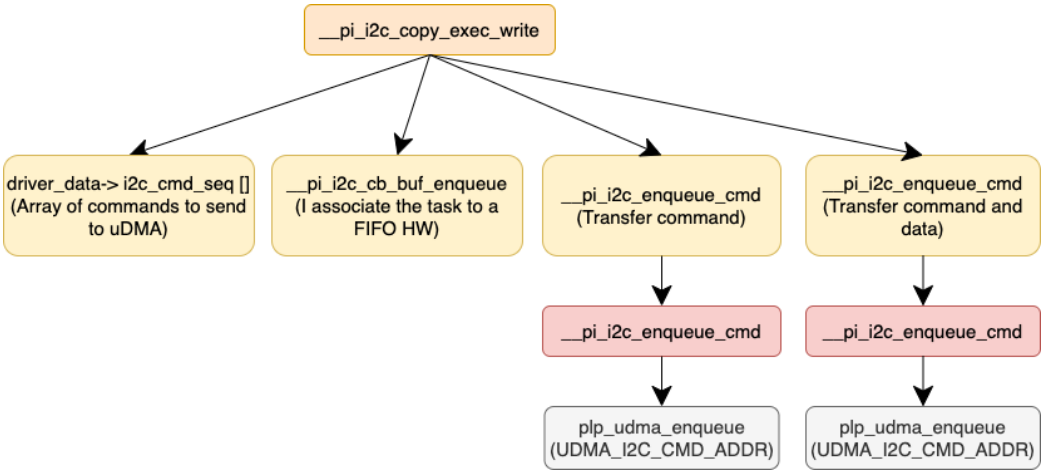

Figura 4.59: Schema PULP-OS per I^2C di __pi_i2c_copy_exec_write

La funzione __pi_i2c_copy_exec_write va inizialmente a riempire il buffer di comandi, driver_data->i2c_cmd_seq[], da inviare al uDMA per poter eseguire l'operazione di invio dei dati con l'interfaccia I^2C, dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.5.3. Successivamente si va a riempire il buffer buffer_to_write con i dati che si vogliono inviare. Dopodiché con la funzione __pi_i2c_enqueue_cmd si va prima a inviare il buffer di comandi e poi il buffer di dati. La funzione __pi_i2c_enqueue_cmd a sua volta va a chiamare la funzione plp_udma _enqueue, la quale accoda un nuovo trasferimento sul canale command del uDMA.

La funzione __pi_i2c_copy_exec_read va inizialmente a riempire il buffer di comandi, driver_data->i2c_cmd_seq[], da inviare al uDMA per poter eseguire l'operazione di ricezione dei dati dall' interfaccia I^2C, dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.5.3. Dopodiché con

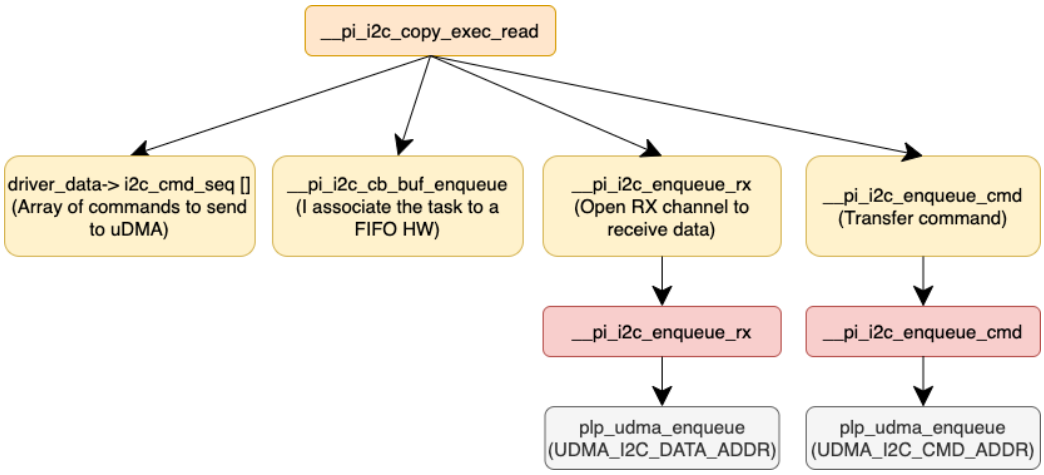

Figura 4.60: Schema PULP-OS per I^{2}C di __pi_i2c_copy_exec_read

la funzione __pi_i2c_enqueue_cmd si va a inviare il buffer di comandi e successivamente con la funzione __pi_i2c_enqueue_rx si invia l'indirizzo della partizione di memoria in L2 dove si vogliono andare a collocare i dati ricevuti. La funzione __pi_i2c_enqueue_cmd e __pi_i2c_enqueue_rx chiamano a loro volta la funzione plp_udma_enqueue, la quale accoda un nuovo trasferimento rispettivamente sul canale rx e command del uDMA.

La funzione __pi_i2c_send_stop_cmd si utilizza per inviare il comando di EOT e il comando di STOP. Per fare questo si utilizza la funzione __pi_i2c_enqueue_cmd che a sua volta chiama la funzione plp_udma_enqueue, la quale accoda un nuovo trasferimento sul canale command del uDMA.

La funzione __pi_i2c_send_only_eot_cmd si utilizza per inviare solo il comando di EOT. Per fare questo si utilizza la funzione __pi_i2c_enqueue_cmd che a sua volta chiama la funzione plp_udma_enqueue, la quale accoda un nuovo trasferimento sul canale command del uDMA.

La funzione __pi_i2c_cmd_handler, Fig. 4.61, rappresenta ISR che viene eseguita a seguito di un evento prodotto dall'invio dei comandi. In base ai flags che vengono settati nelle funzioni __pi_i2c_copy_exec_read e __pi_i2c_copy_exec_write, questa procedura può chiamare diverse funzioni. Infatti nel caso in cui il trasferimento da eseguire abbia un bufer

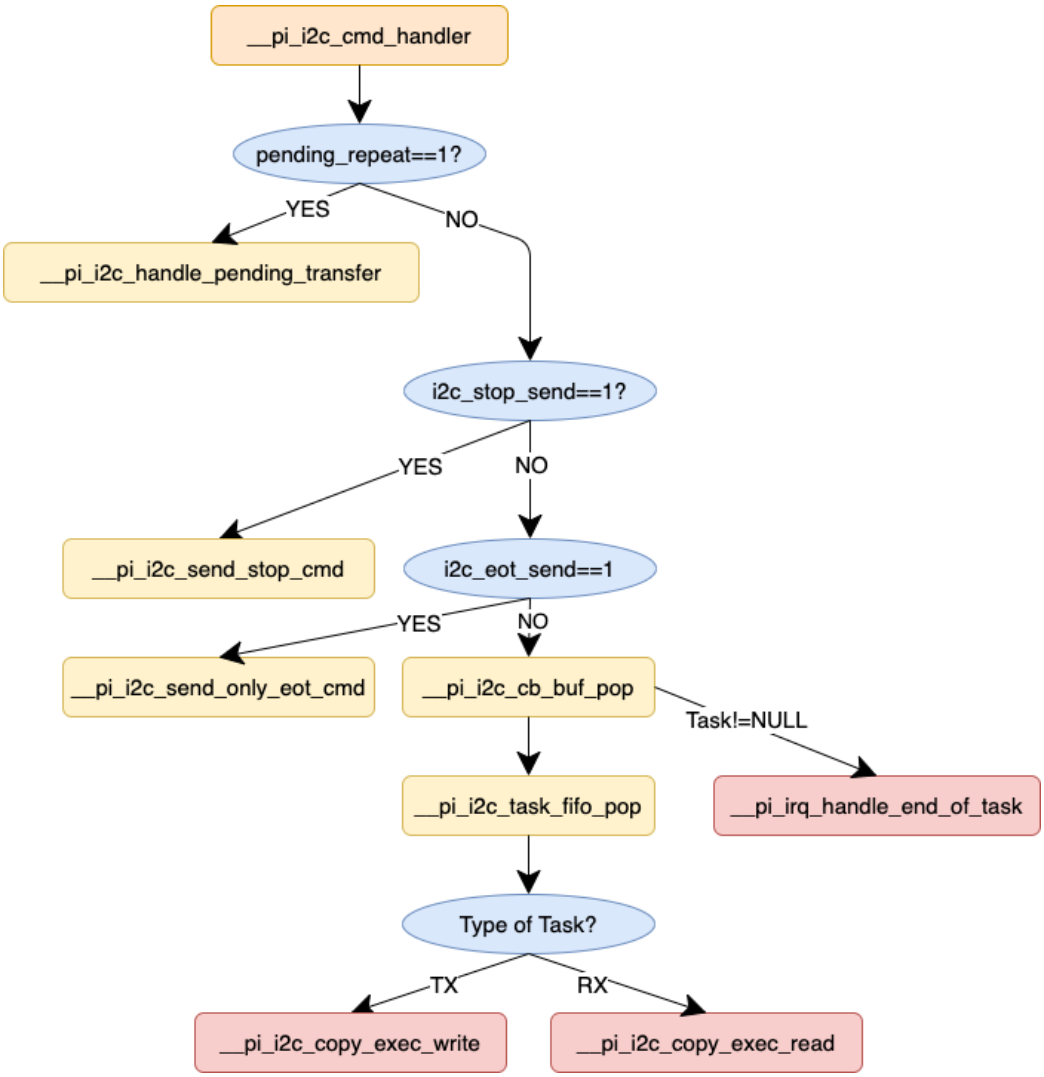

Figura 4.61: Schema PULP-OS per I^2C di __pi_i2c_cmd_handler

di dati troppo grande, allora viene settato il flag pending_repeat, e con la funzione __pi_i2c_handle_pending_transfer si esegue il trasferimento del buffer di comandi in più transazioni. Nel caso in cui venga settato il flag i2c_stop_send allora viene eseguita la funzione __pi_i2c_send_stop_cmd, in caso contrario se la funzione __pi_i2c_cb_buf_pop contiene un task non nullo allora questo viene rilasciato con la funzione __pi_irq_handle_end_of_task. Successivamente si controlla se si ha qualche trasferimento in co-

da nella FIFO software. Nel caso in cui ci sia qualche trasferimento in coda, in base al tipo di trasferimento, viene chiamata la funzione __pi_i2c_copy_exec _read o __pi_i2c_copy_exec_write.

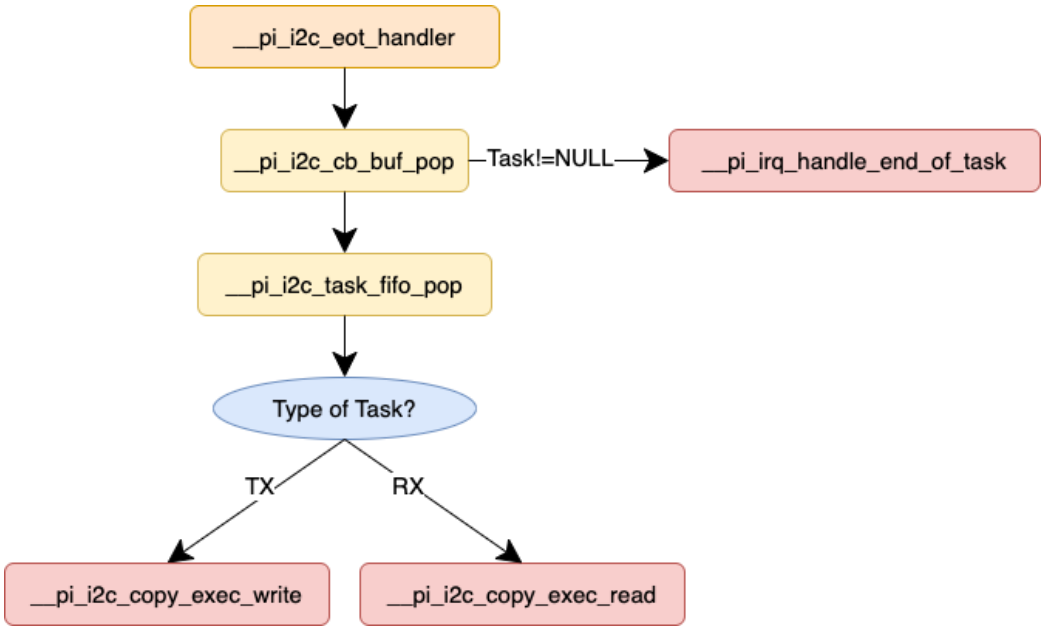

Figura 4.62: Schema PULP-OS per I^2C di __pi_i2c_eot_handler

La funzione __pi_i2c_eot_handler, Fig. 4.62, rappresenta ISR che viene eseguita a seguito di un evento prodotto dall'invio del comando di EOT. Nel caso in cui __pi_i2c_cb_buf_pop contenga un task non nullo allora questo viene rilasciato con la funzione __pi_irq_handle_end_of_task. Successivamente si controlla se si ha qualche trasferimento in coda nella FIFO software. Nel caso in cui ci sia qualche trasferimento in coda, in base al tipo di trasferimento, viene chiamata la funzione __pi_i2c_copy_exec_read o __pi_i2c_copy_exec_write.

La funzione __pi_irq_handle_end_of_task, Fig. 4.63, va a leggere id del task e in base a questo esegue la funzione pos_task_push_locked oppure la funzione pi_task_push.

La funzione __pi_i2c_detect scansiona il bus i2c per rilevare i dispositi-

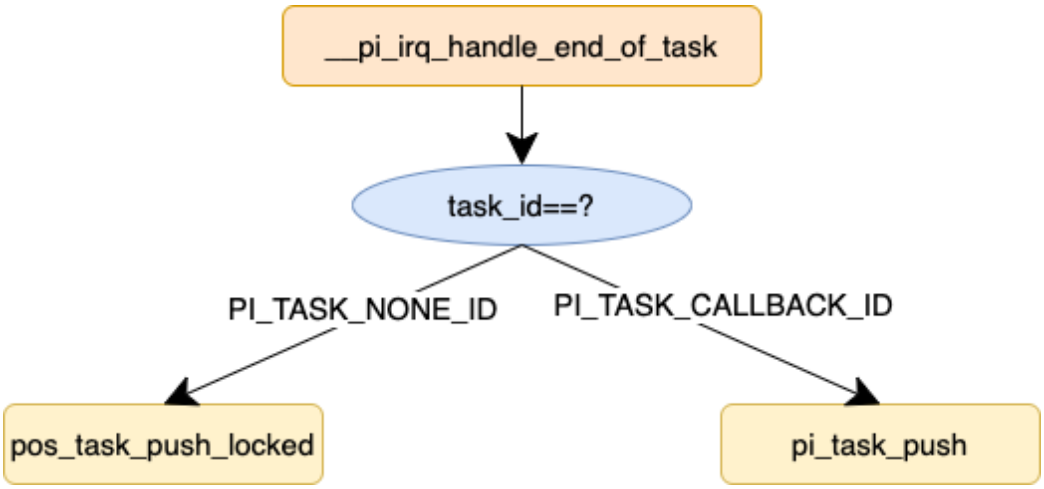

Figura 4.63: Schema PULP-OS per I²C di __pi_irq_handle_end_of_task

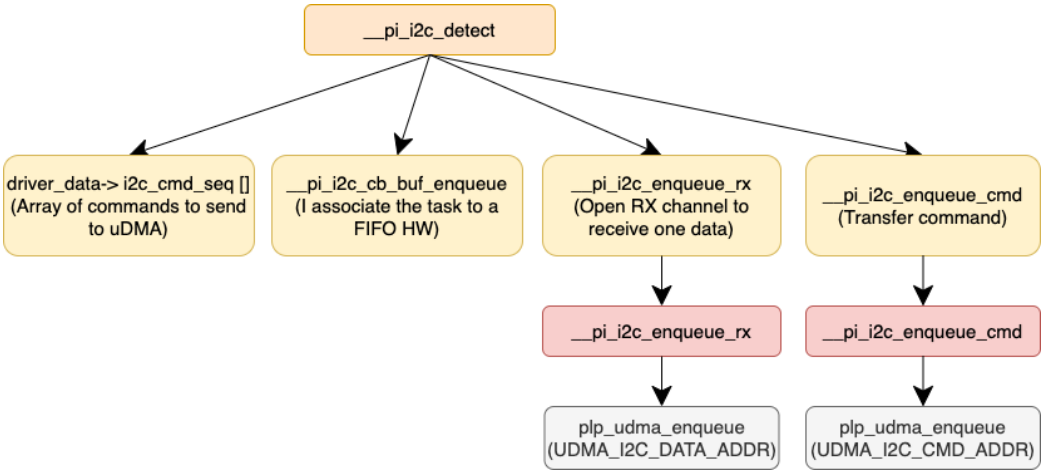

Figura 4.64: Schema PULP-OS per I²C di __pi_i2c_detect

vi collegati. Questa procedura va inizialmente a riempire il buffer di comandi, driver_data->i2c_cmd_seq[], da inviare al uDMA per poter eseguire l'operazione di ricezione dei dati dall' interfaccia I²C, dove i comandi che vengono utilizzati sono stati descritti nel capitolo 2.5.3. Dopodiché con la funzione __pi_i2c_enqueue_cmd si va prima a inviare il buffer di comandi e successivamente con la funzione __pi_i2c_enqueue_rx si invia l'indirizzo

della partizione di memoria in L2 dove si vuole andare a collocare il dato ricevuto. La funzione __pi_i2c_enqueue_cmd e __pi_i2c_enqueue_rx chiamano a loro volta la funzione plp_udma_enqueue, la quale accoda un nuovo trasferimento rispettivamente sul canale rx e command del uDMA.

4.3.3 Utilizzo del driver I^{2}C su una memoria EEPROM

Di seguito vengono mostrati i test eseguiti per validare quelli che sono i driver per i due RTOS. I test realizzati sono di due tipi, si ha infatti il test i2c_eeprom_sync, dove la sua caratteristica principale è quella di fare trasferimenti sincroni, e il test i2c_eeprom_async, dove la sua caratteristica principale è quella di fare trasferimenti asincroni. Questi due test non cambiano nel caso si utilizzi FreeRTOS o PULP-OS. I test per essere eseguiti utilizzeranno il file i2c-v2.c, comune a entrambi gli RTOS. Si suppone di voler utilizzare l'interfaccia I2C0 per andare a scrivere e leggere un buffer di dati su una memoria EEPROM. La memoria EEPROM utilizzata in questo caso è la 24AA1025 della Microchip.

L'indirizzo della memoria EEPROM ha il formato mostrato in Fig. 4.65. Questo è composto da 7 bit, dei quali solo tre sono modificabili. I primi quattro bit sono quelli di Control Code e non possono essere cambiati. Dato che ciascun dispositivo ha limiti di indirizzamento interno, allora si divide ogni blocco di memoria in due segmenti da 512 Kbit. Il Block Select Bits, B0, controlla l'accesso a ciascuna "metà". Infine settando opportunamente i Chip Select Bits è possibile collegare fino a quattro memorie EEPROM alla stessa interfaccia I^{2}C. Nel caso dei due test, il Client Address è settato ad 0x50, quindi 0b1010000. A questo Client Address sarà poi aggiunto il bit 0 se si vuole eseguire un operazione di scrittura oppure 1 se si vuole fare un operazione di lettura.

Dal protocollo della memoria EEPROM, per scrivere una pagina su questa memoria bisogna eseguire le istruzioni mostrate in Fig. 4.66. Allora prima bisognerà inviare lo start bit e l'indirizzo, dove il direction bit deve essere già compreso. Dopodiché si aspetterà di ricevere il segnale di acknow-

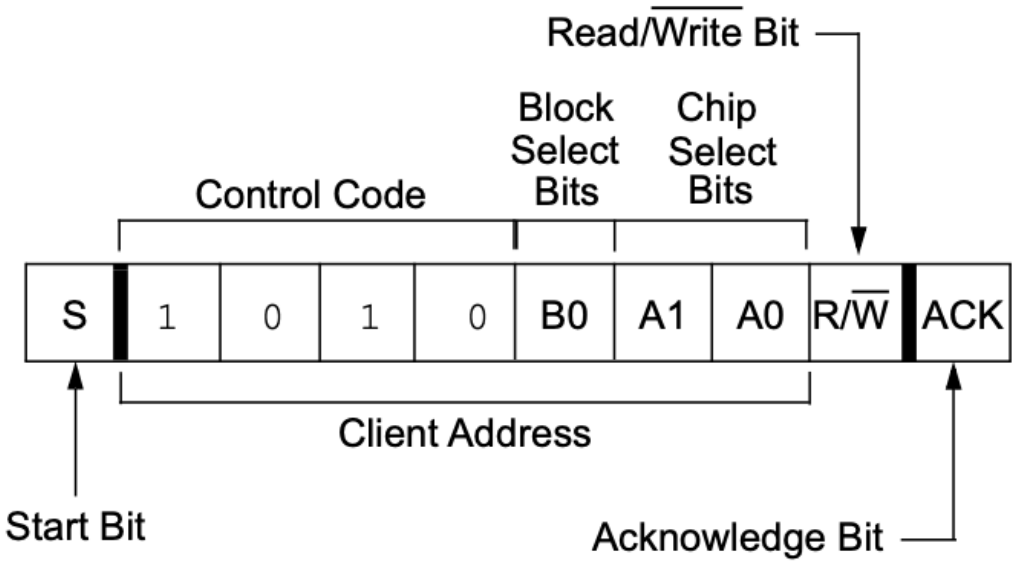

Figura 4.65: Address EEPROM Memory 24AA1025

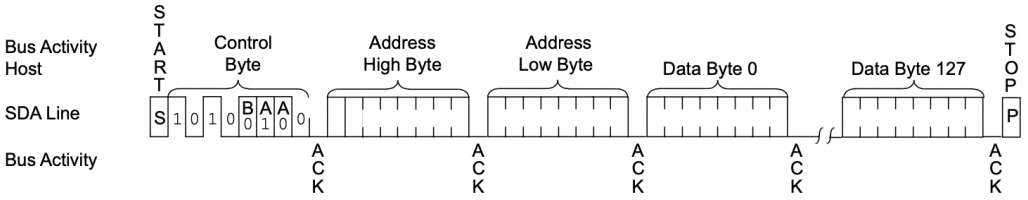

Figura 4.66: Page Write EEPROM Memory 24AA1025

ledge. Ricevuto questo segnale bisognerà inviare l'indirizzo della partizione di memoria che si vuole scrivere, dato che l'indirizzo è a 16 bit si faranno due trasferimenti da 8 bit, dove per ognuno di essi bisognerà aspettare il segnale di acknowledge. Poi si invieranno i dati a 8 bit che si vogliono scrivere sulla memoria, dove per ognuno di essi bisognerà aspettare il segnale di acknowledge. Infine si invierà il bit di stop per concludere il trasferimento.

Dal protocollo della memoria EEPROM, per leggere una pagina casua-

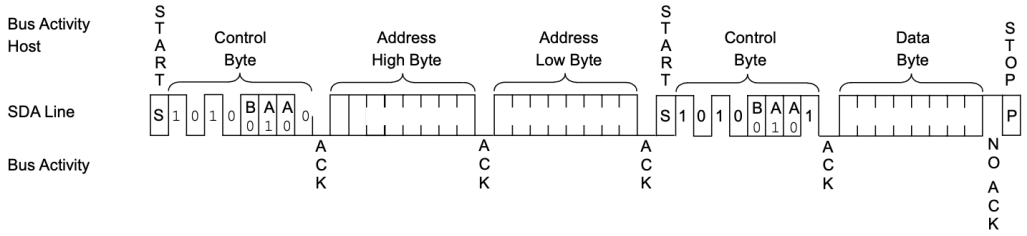

Figura 4.67: Random Read EEPROM Memory 24AA1025

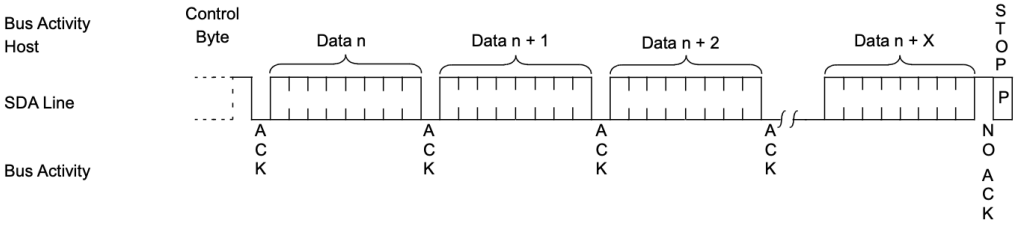

Figura 4.68: Sequential Read EEPROM Memory 24AA1025

le da questa memoria bisogna eseguire le istruzioni mostrate in Fig. 4.67, mentre per leggere in modo sequenziale da questa memoria, partendo da un certo indirizzo, bisogna eseguire le istruzioni mostrate in Fig. 4.68. Allora per leggere in modo sequenziale dalla memoria EEPROM bisognerà prima inviare lo start bit e l'indirizzo, dove il direction bit deve essere già compreso. Dopodiché si aspetterà di ricevere il segnale di acknowledge. Ricevuto questo segnale bisognerà inviare l'indirizzo della partizione di memoria che si vuole leggere, dato che l'indirizzo è a 16 bit si faranno due trasferimenti da 8 bit, dove per ognuno di essi bisognerà aspettare il segnale di acknowledge. Una volta ricevuto bisognerà inviare di nuovo lo start bit seguito dall'indirizzo della memoria EEPROM. Dopo aver ricevuto un dato bisogna aspettare che il master invii il bit di acknowledge al dispositivo. Qunado si è inviato l'ultimo dato, allora il master deve inviare il bit di acknowledge al dispositivo e poi inviare il bit di stop per concludere la ricezione dei dati.

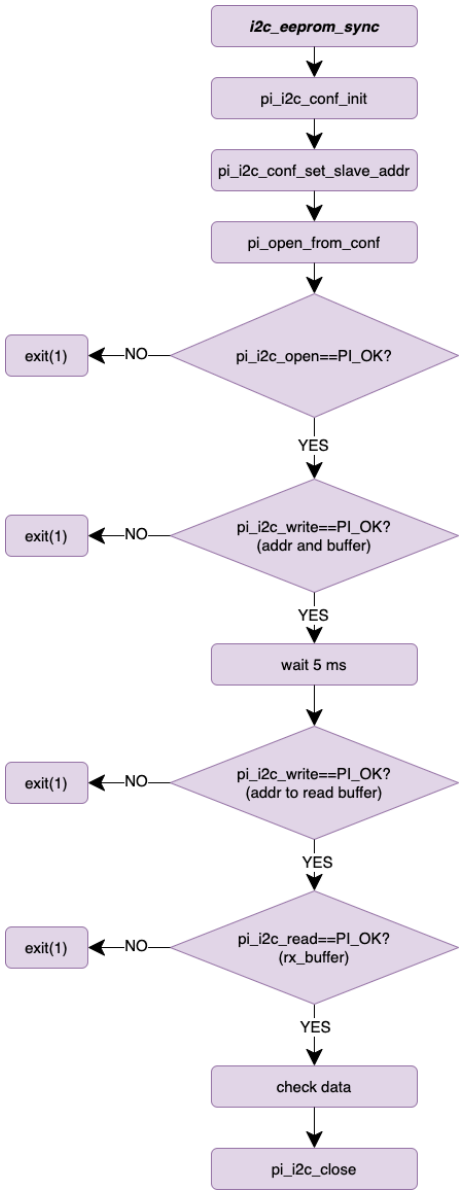

Figura 4.69: Diagramma di flusso test i2c_eeprom_sync

Di seguito viene mostrato il flusso del codice nel caso del test i2c_eeprom_sync, Fig. 4.69. Con la funzione pi_i2c_conf_init si va ad associare, alla struttura dati che descrive l'interfaccia I^2C, la configurazione iniziale dei suoi parametri. Con la funzione pi_i2c_conf_set_slave_addr si va a salvare

l'indirizzo della memoria EEPROM all'interno della struttura che descrive il dispositivo. Con la funzione pi_open_from_conf si associa alla struttura device, la configurazione dell'interfaccia I^2C. Con la funzione pi_i2c_open si va ad inizializzare l'interfaccia I^2C. Nel caso in cui durante l'apertura dell'interfaccia ci sia un errore, ad esempio non si riesce ad allocare memoria nel heap, allora questa ritorna un valore diverso da PI_OK e di conseguenza si esce dall'esecuzione del test. Dopo questa prima parte di configurazioni, si ha il blocco di istruzioni con cui si vuole andare a scrivere sulla memoria EEPROM il buffer di dati. Per farlo si utilizzerà la funzione pi_i2c_write, le cui sotto implementazioni sono diverse in base ai due RTOS, come mostrato in Fig. 4.36 e Fig. 4.37. Infatti le funzioni pi_task_block, pi_task_wait_on e __pi_i2c_copy_exec_write avranno implementazioni diverse nei due RTOS. Questa funzione non restituerà PI_OK nel caso in cui, dopo aver inviato i dati, il valore del NACK è 1. Per poter andare a scrivere una pagina della memoria EEPROM bisogna seguire le istruzioni mostrare in Fig. 4.66. Allora prima si invierà l'indirizzo della memoria EEPROM sul bus I^2C, poi l'indirizzo della partizione di memoria che si vuole scrivere, seguito dai dati che si vogliono scrivere. Una volta inviati i dati alla memoria bisogna aspettare che questa abbia finito di scriversi su di essa. Una volta completata l'operazione di scrittura sulla memoria EEPROM si può passare a leggerne il contenuto. Per farlo si utilizzerà la funzione pi_i2c_read, la cui sotto implementazione è diversa in base ai due RTOS, come mostrato in Fig. 4.38 e Fig. 4.39. Infatti le funzioni pi_task_block, pi_task_wait_on e __pi_i2c_copy_exec_read avranno implementazioni diverse nei due RTOS. Questa funzione non restituerà PI_OK nel caso in cui, dopo aver ricevuto i dati, il valore del NACK è 1. Per poter andare a leggere una pagina della memoria EEPROM bisogna seguire le istruzioni mostrare in Fig. 4.67 e Fig. 4.68. Allora prima si invierà l'indirizzo della memoria EEPROM sul bus I^2C, poi l'indirizzo della partizione di memoria che si vuole leggere, successivamente si invierà di nuovo l'indirizzo della memoria EEPROM seguito dal invio del comando del uDMA per leggere i

dati. Dopodiché avverrà il check data, in cui si controlla che i valori ricevuti coincidano con quelli inviati. Infine si chiude l'interfaccia I^2C con la funzione pi_i2c_close. Queste istruzioni, cioè pi_i2c_write e la pi_i2c_read, sono di tipo sincrono, quindi il processore aspetterà l'event task di ognuna di queste istruzioni prima di passare alla prossima, come descritto nel capitolo 3.3 e 3.4.

Di seguito viene mostrato il flusso del codice nel caso del test i2c_eeprom_async, Fig. 4.70. Con la funzione pi_i2c_conf_init si va ad associare, alla struttura dati che descrive l'interfaccia I^2C, la configurazione iniziale dei suoi parametri. Con la funzione pi_i2c_conf_set_slave_addr si va a salvare l'indirizzo della memoria EEPROM all'interno della struttura che descrive il dispositivo. Con la funzione pi_open_from_conf si associa alla struttura device, la configurazione dell'interfaccia I^2C. Con la funzione pi_i2c_open si va ad inizializzare l'interfaccia I^2C. Nel caso in cui durante l'apertura dell'interfaccia ci sia un errore, ad esempio non si riesce ad allocare memoria nel heap, allora questa ritorna un valore diverso da PI_OK e di conseguenza si esce dall'esecuzione del test. Dopo questa prima parte di configurazioni, si ha il blocco di istruzioni nel quale si vanno a inizializzare i task, questo andando a utilizzare la struttura pi_task_t e la funzione pi_task_block. Il numero di task utilizzati saranno uguali al numero di funzioni di tipo pi_i2c_write_async e pi_i2c_read_async utilizzate. Successivamente si vuole andare a scrivere sulla memoria EEPROM il buffer di dati. Per farlo si utilizzerà la funzione pi_i2c_write_async, la cui sotto implementazione è diversa in base ai due RTOS, come mostrato in Fig. 4.36 e Fig. 4.37. Per poter andare a scrivere una pagina della memoria EEPROM bisogna seguire le istruzioni mostrare in Fig. 4.66. Allora prima si invierà l'indirizzo della memoria EEPROM sul bus I^2C, poi l'indirizzo della partizione di memoria che si vuole scrivere, seguito dai dati che si vogliono scrivere. Una volta inviati i dati alla memoria bisogna aspettare che questa abbia finito di scriverli su di essa. Una volta completata l'operazione di scrittura sulla memoria EEPROM si può passare a leggerne il contenuto. Per farlo si utilizzerà la funzione pi_i2c_read_async,

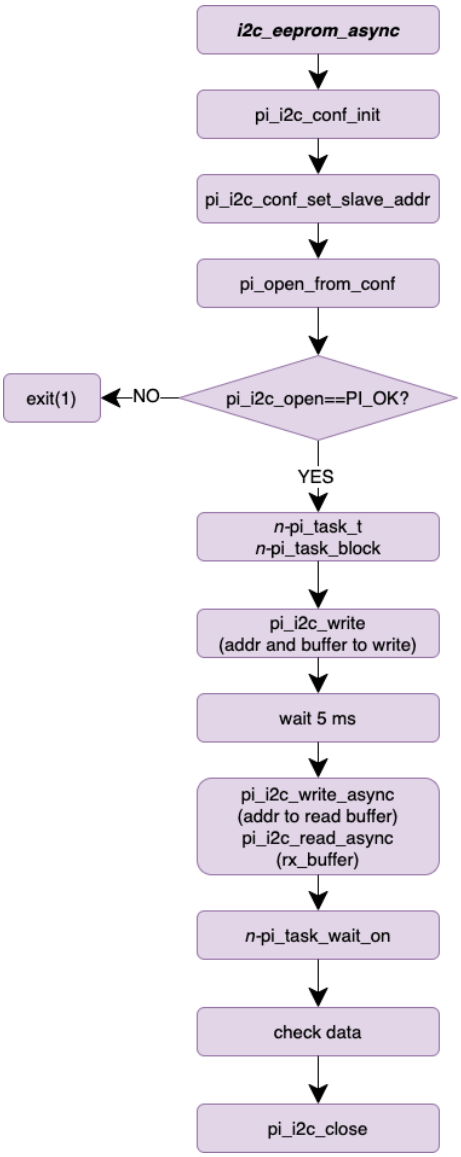

Figura 4.70: Diagramma di flusso test i2c_eeprom_async

la cui sotto implementazione è diversa in base ai due RTOS, come mostrato in Fig. 4.38 e Fig. 4.39. Allora prima si invierà l'indirizzo della memoria EEPROM sul bus I^2C, poi l'indirizzo della partizione di memoria che si vuole leggere, successivamente si invierà di nuovo l'indirizzo della memoria EEPROM seguito dal invio del comando per leggere i dati del uDMA. Con la

funzione pi_task_wait_on, si aspetta di ricevere l'event task per ogni trasferimento asincrono fatto. Dopodiché avverrà il check data, in cui si controlla che i valori ricevuti coincidano con quelli inviati. Infine si chiude l'interfaccia I^2C con la funzione pi_i2c_close. Le istruzioni pi_i2c_write_async e pi_i2c_read_async, sono di tipo asincrono, quindi il processore non aspetta l' event task per ogni trasferimento ma solo alla fine verificherà, con la funzione pi_task_wait_on, di averli ricevuti tutti, come descritto nel capitolo 3.3 e 3.4.

4.3.4 Utilizzo del driver I^2C per BUS Scan

Lo scopo del test seguente è quello di verificare se sul bus dell' interfaccia I^2C0 siano presenti dispositivi oppure no. Per fare questo si utilizza il test che segue il diagramma di flusso di Fig. 4.71. Dato che ogni dispositivo connesso all'interfaccia I^2C ha un indirizzo di 7 bit, allora gli indirizzi che si andranno a leggere vanno dallo 0x0 fino al 0x7F.

Quello che fa questo test è inizialmente andare a configurare l'interfaccia I^2C0 e il device ipoteticamente connesso ad essa, questo con la funzione i2c_init. Nel primo caso allora si supporrà che all'interfaccia I^2C0 sia connesso un dispositivo che ha un indirizzo pari a 0x0. Successivamente si andrà a scrivere a questo indirizzo il valore 0, attraverso la funzione i2c_write. Nel caso in cui non ci fosse nessun dispositivo connesso, allora il segnale di acknowelgment sarrebbe 0 il che vuol dire che il segnale di SDA rimane a 1. Infatti come si può vedere dalla Fig. 4.72, all'indirizzo 0x0 non è connesso nessun dispositivo e il segnale di NACK rimane a 1. Nel caso in cui ci fosse un dispositivo connesso, allora il segnale di acknowelgment, ack, sarrebbe 1, mentre il segnale di not acknowelgment a 0, NACK, il che vuol dire che il segnale di SDA rimane a 0. Infatti come si può vedere dalla Fig. 4.73, all'indirizzo 0x50 è connesso un dispositivo e il segnale di NACK rimane a 0. Dopodichè si passa alla funzione i2c_close, con la quale si andrà a chiudere l'interfaccia I^2C0 e il device al indirizzo 0x0. Questo ciclo continua fino a che non si arriva all'indirizzo 0x7F. Dopo aver letto tutti gli indirizzi

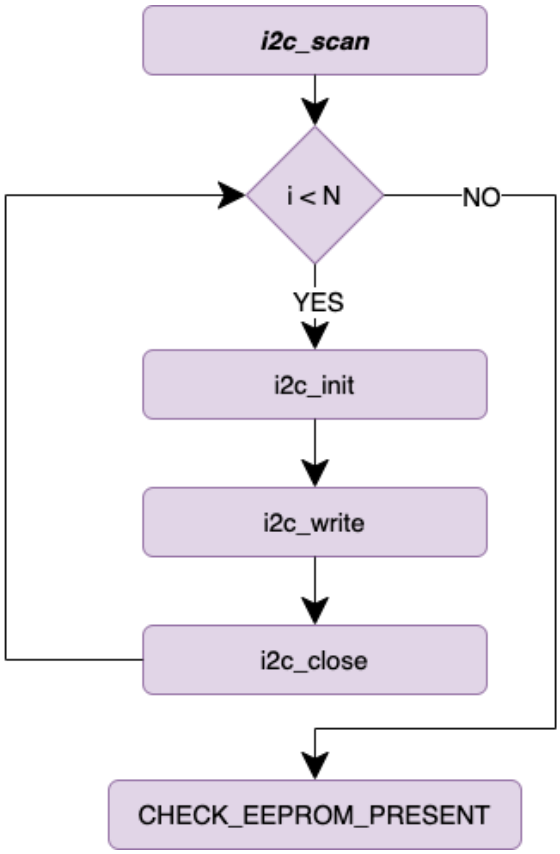

Figura 4.71: Diagramma di flusso test i2c bus scan, N = 0x0 a 0x7F

è possibile verificare se è presente la memoria EEPROM, oppure no, con il CHECK_EEPROM_PRESENT.

Figura 4.72: Segnale di ACK a 0 dalle waveforms

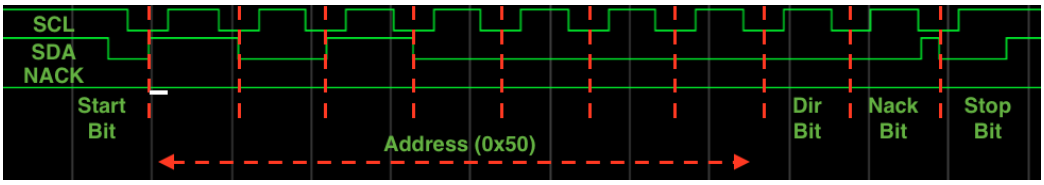

Figura 4.73: Segnale di ACK a 1 dalle waveforms

4.4 Mantenibilità del Codice

In questo sezione si vuole esporre, attraverso alcuni grafici, i vantaggi che si sono avuti, in termini di righe di codice, andando a realizzare un abstraction layers per lo sviluppo dei driver per le periferiche di comunicazione SPI e I^2C. Dato che i driver vengono utilizzati su due RTOS differenti, ma comunque nello stesso SDK, allora andare a ripetere completamente l'intero codice dei driver per i due RTOS, a meno delle procedure OS dipendenti, avrebbe portato una eccessiva ridondanza. Allora attraverso l'abstraction layer presentato si sono potuti realizzare dei driver OS-Indipendent e successivamente è stato possibile ridurre la ripetizione di linee di codice. Per fare questo l'abstraction layer è stato realizzato come mostrato in Fig. 4.3 e Fig. 4.4. Dalle figure si evince che il driver è stato diviso principalmente in tre parti. Si ha una parte comune, dove si inseriscono le parti comuni del driver rispetto ai due RTOS, una seconda parte in cui si implementano le funzioni diverse dei driver per i due RTOS, ad esempio qui si implementano le funzioni OS dipendenti, e infine si ha una terza parte in cui si implementano le dichiarazioni delle PMSIS per i driver.

Il grafico mostrato in Fig. 4.74 mostra il numero di righe di codice usate nei vari file che compongono l'abstraction layer. Con il termine driver inziale si fa riferimento al driver che contiene l'implementazione sia per FreeRTOS che per PULP-OS, mentre con il Common File si fa riferimento ai file che descrivono la parte comune per i driver, invece con A-L FreeRTOS e PULP-OS si fa riferimento ai file che contengono le funzioni OS-dipendent, infine con Driver PMSIS si hanno le definizioni delle dichiarazioni per le PMSIS.

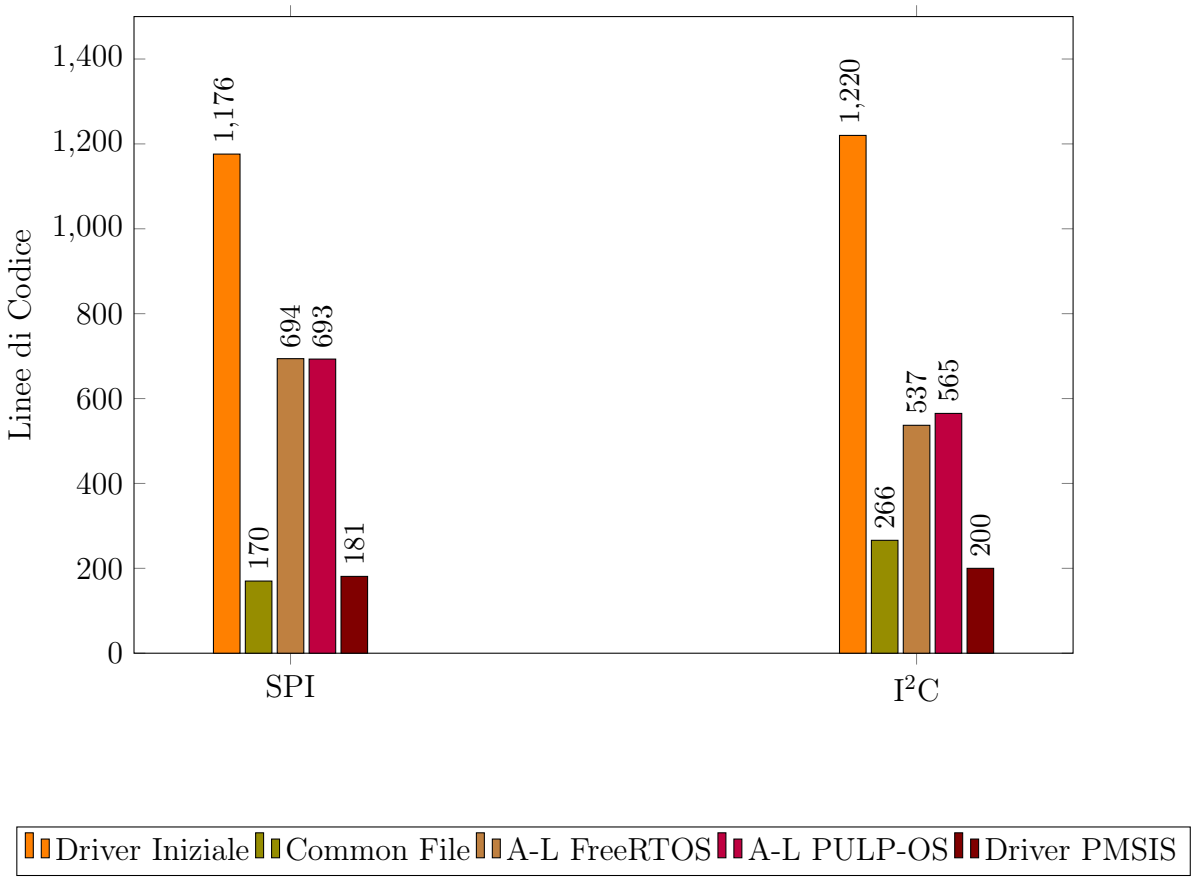

Figura 4.74: Confronto righe di codice

Come si può vedere se non si fosse applicato un abstraction layer allora i driver per le periferiche di comunicazione SPI e I^2C sarebbero composti da circa 1200 righe di codice, il che porterebbe a una loro difficile comprensione, ripetizione di codice e a una difficile procedura di debug. Applicando invece quello che è un abstraction layer si ottengono più file di dimensioni ridotte. In questo modo la comprensione del driver aumenta insieme alla facilità con la quale effettuare le operazioni di debug. In particolare, il driver per la periferica di comunicazione SPI e I^2C hanno dei file che verranno utilizzati da entrambi gli RTOS, mentre i file A-L FreeRTOS e A-L PULP-OS sono implementati in base al RTOS scelto. Allora facendo la differenza tra le righe di codice del Driver Iniziale e la somma dei file che compongono l'abstraction layer, cioè Common File, A-L FreeRTOS, o PULP-OS e i Driver PMSIS si riesce ad ottenere un codice più compatto del 11% per il driver SPI e del 16% per il driver I^2C. Questo porta a un migliore utilizzo e mantenimento del codice.

Conclusioni

Il lavoro presentato attraverso l'elaborato di questa tesi ha descritto inizialmente la piattaforma PULP, che ha lo scopo di ridurre l'onere computazionale e l'utilizzo di risorse, per i nodi finali IoT. Per raggiungere questo obiettivo si è sviluppato un core compatibile con RISC-V ISA e una gerarchia di memoria scalabile. Nel RISC-V ISA si sono aggiunte delle estensioni come hardware-loop, pointer post-increment, Multiply And Accumulate (MAC) e vettorizzazione SIMD. In questo modo si è riusciti a raggiungere prestazioni e densità di codice simili agli MCU basati su un ISA proprietario, come i core della serie ARM Cortex-M.

Per soddisfare i requisiti Hard Real-Time delle applicazioni a cui è sottoposta la piattaforma PULP si sono utilizzati quelli che sono gli RTOS, FreeRTOS e PULP-OS. Questi sono stati utilizzati per gestire gli event task dovuti ai trasferimenti prodotti dalle periferiche SPI e I^2C.

Dato che nel SDK sono presenti due diversi RTOS, attraverso questa tesi si è proposto un metodo per realizzare dei driver per le periferiche di comunicazione SPI e I^2C in modo che siano OS-Indipendent. All'interno delle PMSIS allora è stato realizzato un abstraction-layers con lo scopo di poter rendere l'utilizzo dei driver indipendente dal RTOS scelto. In questo modo è stato possibile rendere il driver per la periferica di comunicazione SPI 11x più compatto, mentre per la periferica di comunicazione I^2C 16x più compatto, rispetto al caso in cui non si fosse utilizzato l'abstractioni layer. Questo va a facilitare la comprensione del driver e il suo debug.

Grazie alla struttura del abstraction-layers creato per i driver delle pe-

riferiche di comunicazione SPI e I^2C, in futuro l'applicazione di un nuovo RTOS all'interno del SDK porterà allo sviluppo di solo file OS-Dipendent semplificando in maniera consistente implementazione di nuovo codice.

Bibliografia

[1] PULP Platform.
https://pulp-platform.org/projectinfo.html

[2] PULPissimo Datasheet.
https://github.com/pulp-platform/pulpissimo/blob/master/doc/datasheet/datasheet.pdf

[3] PULP Datasheet.
https://github.com/pulp-platform/pulp/blob/master/doc/datasheet.pdf

[4] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, e L.Benini "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices", IEEE Transactions on very large scale integration (VLSI) systems, vol. 25, no. 10, OCTOBER 2017.
https://ieeexplore.ieee.org/abstract/document/7864441

[5] A. Pullini, D. Rossi, G. Haugou e Luca Benini, "μDMA: An autonomous I/O subsystem for IoT end-nodes", International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2017.
https://ieeexplore.ieee.org/abstract/document/8106971

[6] Understanding the I2C Bus.
https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1642650606307

Ringraziamenti

Ringrazio il mio relatore, che mi ha dato la possibilità di poter realizzare questo lavoro e di poter imparare molto, e i miei correllatori che mi hanno aiutato nello svolgimento di questa tesi.