Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica (Computer Engineering)
A.A. 2024/2025
Sessione di Laurea Ottobre 2025

Performance Optimization of a CAN Data Server on a Heterogeneous Automotive Embedded System

Design, implementation, and optimization of embedded firmware for efficient CAN data handling in automotive cluster and infotainment systems.
Master's Thesis developed at Iveco S.p.A. in collaboration with Politecnico di Torino

Relatore:
Luciano Lavagno

Candidato:
Gabriele Russo

Abstract

English

The rapid evolution of automotive electronic architectures, driven by the increasing adoption of infotainment and digital cluster systems, has led to a significant growth in the amount of data exchanged among Electronic Control Units (ECUs) and the transition towards more powerful hardware, such as High-Performance Computers (HPCs). Despite the introduction of new technologies, the Controller Area Network (CAN) remains the predominant communication protocol, requiring efficient firmware solutions to guarantee low latency and reliable performance.

This thesis presents the design, implementation, and optimization of embedded firmware for CAN data handling on heterogeneous automotive platforms, composed of microcontrollers running hard real-time operating systems and microprocessors running general-purpose automotive operating systems. The work includes the development of a CAN codec, the integration of the firmware with higher-level applications through efficient data serialization, and the evaluation of performance across different hardware platforms.

The analysis highlights typical bottlenecks such as dynamic memory allocation, shared-memory communication, and debugging overhead. Optimization strategies were applied, leading to reduced execution times, improved stability, and more efficient resource usage, as demonstrated through illustrative performance results.

The conclusions outline the contribution of this work in enhancing the reliability and scalability of in-vehicle communication systems and discuss potential future developments.

Sommario

Italiano

La rapida evoluzione delle architetture elettroniche automobilistiche, guidata dalla crescente diffusione dei sistemi di infotainment e dei cruscotti digitali, ha portato a un significativo aumento della quantità di dati scambiati tra le Electronic Control Units (ECU) e alla transizione verso hardware più potenti, come gli High Performance Computers (HPC). Nonostante l'introduzione di nuove tecnologie, il Controller Area Network (CAN) rimane il protocollo di comunicazione predominante, richiedendo soluzioni firmware efficienti per garantire bassa latenza e prestazioni affidabili.

La presente tesi descrive la progettazione, l'implementazione e l'ottimizzazione di firmware embedded per la gestione dei dati CAN su piattaforme automotive eterogenee, composte da microcontrollori con sistemi operativi real-time hard e microprocessori con sistemi operativi automotive di tipo general-purpose. Il lavoro comprende lo sviluppo di un codec CAN, l'integrazione del firmware con applicazioni di più alto livello tramite meccanismi efficienti di serializzazione dei dati e la valutazione delle prestazioni su diverse piattaforme hardware.

L'analisi mette in evidenza i principali colli di bottiglia, come l'allocazione dinamica della memoria, la comunicazione tramite memoria condivisa e l'overhead dovuto alle attività di debug. Sono state applicate strategie di ottimizzazione che hanno portato a una riduzione dei tempi di esecuzione, a una maggiore stabilità e a un utilizzo più efficiente delle risorse, come dimostrato attraverso risultati prestazionali esemplificativi.

Le conclusioni delineano il contributo di questo lavoro nel migliorare l'affidabilità e la scalabilità dei sistemi di comunicazione veicolari e discutono i possibili sviluppi futuri.

Ringraziamenti

Ringrazio innanzitutto la mia famiglia, che mi ha accompagnato con pazienza, fiducia e generosità lungo tutto il percorso accademico. Il loro sostegno, umano ed economico, è stato determinante per affrontare con equilibrio le sfide di questi anni.

Un grazie sentito va anche agli amici, per l'affetto, l'ironia e la comprensione con cui mi hanno accompagnato, rendendo più umano e leggero un percorso spesso rigoroso. Estendo il mio ringraziamento a coloro che, per ragioni diverse, non sono più presenti nella mia vita ma che hanno avuto un ruolo significativo nella mia crescita personale e accademica.

Infine, ringrazio i docenti, tutor aziendale e colleghi incontrati lungo la strada per gli stimoli ricevuti, le discussioni costruttive e le opportunità di sviluppo tecnico e professionale. Il confronto con loro ha contribuito in modo concreto alla qualità del lavoro qui presentato e al mio metodo di studio e di ricerca.

Desidero dedicare questa tesi a mio padre, che purtroppo non ha potuto assistere alla conclusione di questo cammino. Il suo esempio e il suo incoraggiamento, presenti anche nei momenti più difficili, sono stati una guida silenziosa che mi ha sostenuto fino a questo traguardo. Questo lavoro è per lui.

Indice

Indice delle figure

1 Introduzione

Negli ultimi anni il settore automotive ha attraversato una profonda trasformazione tecnologica, guidata dall'introduzione di sistemi avanzati di infotainment, assistenza alla guida e connettività veicolo–utente. Le moderne vetture integrano un numero crescente di schermi e interfacce — dal cruscotto digitale (cluster) ai display touch di grandi dimensioni — che offrono funzionalità evolute come mappe di navigazione in tempo reale, integrazione con smartphone tramite Android Auto e Apple CarPlay, streaming multimediale e assistenti vocali.

Sebbene in precedenza molti sensori o attuatori erano cablati in maniera diretta adesso sono quasi tutti interfacciati tramite bus dati, principali o secondari, che comunicano con due o più centraline per permettere all'intero veicolo di operare in sincronia ed avere una serie di automatismi tali da richiedere un elevata complessità anche per le funzioni più semplici come alzare un finestrino.

Questa evoluzione ha comportato un aumento esponenziale della quantità di dati che devono essere scambiati in tempo reale tra le centraline elettroniche di controllo (ECU) di un veicolo. Il Controller Area Network (CAN) rimane il protocollo di comunicazione predominante in ambito automotive per la sua affidabilità, robustezza e capacità di operare in ambienti gravosi. Tuttavia, il crescente numero di sensori, attuatori e dispositivi di visualizzazione comporta un incremento del traffico dati sulla rete, rendendo sempre più necessarie ottimizzazioni lato firmware e architettura di sistema per garantire prestazioni stabili e tempi di risposta compatibili con i requisiti di sicurezza.

1.1 Iveco

Iveco Group è uno dei principali costruttori mondiali di veicoli industriali, con una gamma che spazia dai veicoli leggeri per il trasporto urbano e regionale fino ai mezzi pesanti per il lungo raggio, oltre a veicoli speciali per applicazioni difensive, antincendio e fuoristrada. Con sede centrale a Torino e una presenza globale attraverso stabilimenti produttivi e centri di ricerca e sviluppo, Iveco rappresenta un attore chiave nel settore della mobilità commerciale, con una forte attenzione all'innovazione tecnologica e alla sostenibilità.

Negli ultimi anni l'azienda ha investito significativamente nello sviluppo di soluzioni a basse e zero emissioni, includendo veicoli alimentati a gas naturale e veicoli elettrici. Parallelamente, Iveco ha intrapreso una profonda trasformazione digitale, riconoscendo l'importanza crescente del software nella gestione dei veicoli moderni, sia per i sistemi di controllo e diagnostica sia per le interfacce avanzate di

infotainment e HMI (Human Machine Interface).

In questo contesto si inserisce la creazione della nuova Software House Iveco, inaugurata a fine 2024 presso la sede di Lungo Stura Lazio 49 a Torino. Questa struttura, progettata come hub centrale per lo sviluppo software, concentra competenze in ambiti quali firmware embedded, architetture di comunicazione veicolare, infotainment e HMI. L'obiettivo è accelerare lo sviluppo di soluzioni software proprietarie e ottimizzate per la nuova generazione di veicoli Iveco, garantendo affidabilità, scalabilità e prestazioni in linea con le esigenze dei sistemi automotive moderni.

Il team specifico in cui è stato sviluppato il presente lavoro di tesi è il team HMI, dedicato allo sviluppo delle interfacce uomo-macchina dei veicoli Iveco. Il gruppo si occupa della progettazione, implementazione e ottimizzazione del software necessario per la visualizzazione di informazioni critiche sul veicolo, gestione dei dati di sensori e attuatori, e interazione con sistemi di infotainment avanzati. In questo contesto, l'ottimizzazione delle prestazioni di un CAN Data Server rappresenta un caso concreto di come il team HMI lavori per garantire tempi di risposta rapidi, affidabilità dei dati e fluidità nell'interazione utente, elementi fondamentali per un'esperienza di guida sicura e moderna

1.2 Contesto

L'architettura elettronica dei veicoli moderni è caratterizzata da una rete distribuita di ECU interconnesse tramite bus di comunicazione standard come CAN, LIN ed Ethernet automotive. Tra queste, il CAN continua a essere largamente utilizzato per la trasmissione di dati critici a bassa e media velocità, come informazioni di sensori, comandi di attuatori e segnali diagnostici.
La crescente complessità delle funzioni implementate a bordo veicolo ha introdotto nuove sfide:

- **Aumento della densità di informazioni**: i sistemi devono gestire simultaneamente dati provenienti da decine di sensori, telecamere, radar e moduli di infotainment.
- **Requisiti di tempo reale**: molte informazioni devono essere elaborate e visualizzate entro tempi molto ridotti per garantire la sicurezza e la reattività del sistema.
- **Interfacciamento eterogeneo**: i microcontrollori dedicati alle funzioni di basso livello devono comunicare in modo efficiente con processori ad alte prestazioni che gestiscono applicazioni grafiche complesse.

In questo scenario, l'ottimizzazione del firmware che gestisce i flussi di dati CAN diventa cruciale per garantire la stabilità e la scalabilità delle architetture elettroniche veicolari.

Quanto detto è fondamentale per l'**Human-Machine Interface (HMI)**, comprendente tutti i sistemi e le interfacce che permettono al conducente di interagire con il veicolo, come cruscotti digitali, display infotainment, pannelli touch, comandi vocali e feedback tattile. Lo scopo principale dell'HMI è fornire informazioni critiche in modo chiaro e sicuro, supportando il conducente nelle decisioni e nella gestione del veicolo.

Figura 1 Cluster, infotainment ed indicatori/comandi vari di un trattore stradale Iveco S-Way

Nel contesto del team HMI di Iveco, l'obiettivo era sviluppare un cluster ed infotainment completamente funzionante, integrando la raccolta dati dai sensori e dalle ECU con la visualizzazione e l'interazione dell'utente, garantendo al contempo prestazioni ottimali e tempi di risposta real-time.

1.3 Stato dell'arte

Negli ultimi decenni, l'elettronica e il software hanno progressivamente assunto un ruolo predominante nello sviluppo dei veicoli, trasformando l'architettura interna da un insieme di sistemi isolati a una rete interconnessa di **Electronic Control Unit** (**ECU**). Queste centraline, dedicate a funzioni specifiche come il controllo motore, la sicurezza attiva, il comfort e l'infotainment, devono comunicare tra loro in tempo reale per garantire funzionalità affidabili e integrate.
Lo standard **CAN** (**Controller Area Network**), introdotto da Bosch negli anni '80 [1] e successivamente normato da ISO 11898, si è affermato come punto forte della comunicazione a bordo veicolo grazie a robustezza, tolleranza ai disturbi e determinismo nella gestione delle priorità dei messaggi. La sua architettura multi-master e l'efficiente arbitraggio dei pacchetti lo rendono adatto a scenari critici, dove la latenza deve rimanere bassa anche in condizioni di traffico intenso.

L'evoluzione delle esigenze ha portato allo sviluppo del CAN FD (Flexible Data-rate), che aumenta la dimensione massima del payload (fino a 64 byte) e supporta velocità di trasmissione più elevate nella fase dati, mantenendo retrocompatibilità con il CAN tradizionale. Questa tecnologia si è rivelata cruciale per ridurre l'overhead e migliorare l'efficienza nelle comunicazioni a elevata densità di dati, ad esempio nei sistemi ADAS o nei moduli di diagnostica avanzata.
Parallelamente, l'adozione di Automotive Ethernet sta guadagnando terreno per applicazioni che richiedono larghezza di banda molto superiore, come la gestione di flussi video per telecamere a 360°, sensori Lidar o Radar ad alta risoluzione. Standard come 100BASE-T1 e 1000BASE-T1, progettati specificamente per l'ambiente automotive, offrono velocità fino a 1 Gbps su singolo doppino schermato, mantenendo requisiti di peso e ingombro compatibili con il cablaggio veicolare [2].
Oltre ai protocolli di comunicazione fisica, negli ultimi anni si è assistito alla diffusione di middleware e standard software come **AUTOSAR** (**Automotive Open System Architecture**), che fornisce un'astrazione hardware e un'architettura modulare per lo sviluppo di applicazioni automotive, facilitando la portabilità del software e la gestione delle risorse. AUTOSAR, nelle sue versioni **Classic** e **Adaptive**, è ormai un riferimento per la gestione di sistemi real-time e di applicazioni ad alte prestazioni, rispettivamente.
La crescente complessità delle architetture ha portato anche alla separazione fisica delle funzioni: sistemi con architetture di dominio o le più recenti architetture zonali integrano più funzioni in unità di calcolo centralizzate. In particolare, si tende a ridurre il numero di ECU sostituendole con **High Performance Computer** (**HPC**), capaci di gestire più funzioni simultaneamente, dalla grafica e infotainment fino al controllo real-time dei sistemi critici.
In questi scenari, la comunicazione tra domini (ad esempio, tra la parte di alto livello grafico/infotainment e il controllo real-time delle funzioni critiche) deve essere gestita con protocolli ottimizzati, come SPI o Shared-Memory, quando possibile, per minimizzare latenza e jitter.
Infine, il tema della sicurezza è diventato centrale: l'interconnessione dei veicoli con infrastrutture esterne (V2X) e servizi cloud espone i sistemi embedded a nuove minacce. Gli standard come ISO/SAE 21434 e regolamenti UNECE (WP.29 R155) impongono requisiti stringenti per la sicurezza del software e delle comunicazioni interne, influenzando la progettazione dei protocolli e delle architetture hardware.
In sintesi, lo stato dell'arte delle comunicazioni automotive è caratterizzato da:

- consolidamento del CAN e transizione verso CAN FD;
- crescente adozione di Automotive Ethernet per applicazioni ad alta banda;
- uso di middleware standardizzati come AUTOSAR;
- uso di sistemi operativi real-time;
- tendenza verso architetture centralizzate e zonali;
- attenzione crescente a cybersecurity e diagnostica avanzata.

Questi elementi, con alcune esclusioni dovute ai tempi ristretti ed alle finalità del

progetto svolto nel tirocinio, costituiscono il contesto tecnologico in cui si inserisce il presente lavoro di tesi, volto a sviluppare e ottimizzare un sistema firmware per la gestione e la trasmissione efficiente di dati su piattaforme embedded eterogenee.

1.4 Riassunto capitoli

1.4.1 Reti veicolari e veicoli moderni

Analizza l'evoluzione delle architetture elettroniche a bordo veicolo, mettendo in evidenza il ruolo delle reti di comunicazione nella gestione della crescente complessità funzionale. Dopo una panoramica sull'aumento del numero di ECU e della mole di software necessaria nei veicoli moderni, vengono discusse le principali tipologie di architetture (distribuite, a dominio e zonali) e le relative implicazioni in termini di scalabilità e manutenzione. Infine, il capitolo descrive come la crescente centralità del software abbia portato alla nascita del paradigma dei software-defined vehicles, in cui le funzionalità del veicolo sono sempre più determinate dallo stack software e dalla capacità di gestire in modo efficiente i flussi di dati tra le diverse ECU.

1.4.2 Piattaforme hardware

Si introducono le piattaforme hardware utilizzate nello sviluppo del progetto, con particolare attenzione alle differenze tra microcontrollori e microprocessori in ambito automotive. Dopo aver presentato le principali caratteristiche delle schede impiegate, vengono discusse le problematiche incontrate durante le attività di porting del firmware su diverse MCU e MPU, con riferimento alle differenze nei driver, nei kit di sviluppo e nelle periferiche disponibili. Il capitolo affronta inoltre le sfide legate a compatibilità hardware, mancanza di componenti o SDK incompleti, illustrando le soluzioni adottate per garantire la portabilità e la stabilità del codice.

1.4.3 Organizzazione software

Viene descritta l'organizzazione del software sviluppato, analizzandone sia gli aspetti architetturali sia quelli operativi. Dopo un'introduzione ai modelli utilizzati per garantire modularità e portabilità, come l'Hardware Abstraction Layer e l'architettura esagonale, il capitolo affronta la gestione della comunicazione CAN a livello di driver e codec. Viene inoltre presentato un approfondimento sui sistemi operativi più diffusi in ambito automotive: FreeRTOS e AUTOSAR Classic per i microcontrollori, QNX e AUTOSAR Adaptive per i microprocessori. L'obiettivo è fornire una panoramica delle soluzioni software adottate e del loro ruolo nell'assicurare affidabilità, determinismo e integrazione tra piattaforme eterogenee.

1.4.4 Implementazione software

Qui si entra nel dettaglio dell'implementazione del firmware, descrivendo lo stack software e gli strumenti di sviluppo utilizzati, tra cui IDE, SDK e interfacce di debug.

Vengono illustrati i principali componenti del codice e le tecniche adottate per la gestione delle periferiche, degli interrupt e della logica di codifica/decodifica dei messaggi CAN. Per chiarezza espositiva, gli algoritmi e le soluzioni più rilevanti sono presentati tramite esempi di pubblico dominio, come i buffer di tipo ping-pong, senza riportare materiale proprietario. Il capitolo evidenzia così le scelte implementative alla base del funzionamento del firmware e le metodologie applicate per garantire robustezza ed efficienza.

1.4.5 Integrazione con sistema Host

Analizza il processo di integrazione del firmware con i sistemi host di più alto livello. Viene descritto il ruolo dei codec CAN come interfaccia tra microcontrollori e applicazioni, con particolare attenzione all'utilizzo di formati efficienti per la serializzazione dei dati, come FlatBuffers. Sono inoltre illustrate le modalità di comunicazione tra MCU e host, implementate attraverso diversi protocolli e meccanismi, tra cui SPI, UART, Ethernet e shared memory, in ambienti operativi come Linux, QNX o AUTOSAR. Il capitolo mostra come tali soluzioni permettano di garantire interoperabilità e affidabilità nello scambio di dati tra componenti eterogenei del sistema.

1.4.6 Ottimizzazione e performance

Questo capitolo è dedicato alle attività di analisi e ottimizzazione delle prestazioni del firmware. Dopo aver introdotto le metriche e gli strumenti utilizzati, come contatori hardware e timer, vengono esaminati i principali colli di bottiglia individuati, tra cui l'uso di allocazioni dinamiche, la gestione della memoria condivisa e l'impatto delle routine di debug. Vengono quindi presentate le soluzioni implementate per migliorare l'efficienza, tra cui la ristrutturazione di algoritmi, l'adozione di strutture dati più leggere e la riduzione della complessità nel codec CAN. Il capitolo si conclude mostrando i benefici ottenuti in termini di latenza, utilizzo delle risorse e stabilità complessiva del sistema.

1.4.7 Testing e validazione

Vengono illustrate le metodologie adottate per la verifica e la validazione del firmware sviluppato. Sono presentati i test condotti su banco e in scenari simulati, con l'obiettivo di riprodurre condizioni realistiche senza riportare dati sensibili. I risultati discussi, pur basati su valori esemplificativi, permettono di evidenziare i miglioramenti ottenuti dopo le attività di ottimizzazione, in termini di tempi di risposta, stabilità e affidabilità. Inoltre, il capitolo descrive le strategie di gestione degli errori, l'implementazione di watchdog hardware e software e le procedure di logging utilizzate per monitorare il comportamento del sistema e rilevare eventuali anomalie.

1.4.8 Conclusioni

Il nono capitolo raccoglie le conclusioni del lavoro di tesi, presentando una sintesi dei risultati ottenuti e delle principali ottimizzazioni introdotte nel firmware. Viene proposta una valutazione critica delle soluzioni adottate, evidenziandone i punti di forza e i limiti riscontrati durante lo sviluppo e i test. Infine, il capitolo offre una panoramica delle possibili evoluzioni future, come l'adozione dello standard CAN FD, l'implementazione di SAE J1939 e l'integrazione di meccanismi avanzati di sicurezza e diagnostica, delineando le prospettive per lo sviluppo di sistemi embedded automotive sempre più efficienti e affidabili.

2 Reti veicolari e veicoli moderni

2.1 Evoluzione delle architetture elettroniche a bordo veicolo

Come anticipato nei paragrafi precedenti, tutti i veicoli moderni sono dotati di una o più reti veicolari, necessarie per gestire l'elevata complessità elettronica a bordo. Oggi, un veicolo integra fra **70 e 100 ECU (Electronic Control Units)** — per funzioni che spaziano dal controllo motore e frenata ai sistemi di climatizzazione — rispetto a poche centraline presenti in passato, e ciò comporta una gestione dei dati molto più sofisticata.

L'altra dimensione della complessità è data dal software: un'auto moderna, secondo Porsche Engineering [3], contiene circa **100 milioni di linee di codice**, una cifra sorprendente se si considera che un Boeing 787 ne include "solo" circa 14 milioni.

In un articolo di NXP [4] si parla addirittura di "centinaia di milioni di linee di codice" necessarie per far funzionare sistemi di infotainment, parcheggio automatico, telecamere surround e radar, con un impatto crescente man mano che si evolvono connettività, elettrificazione e autonomia (intesa come l'avvento di automatismi) dei veicoli.

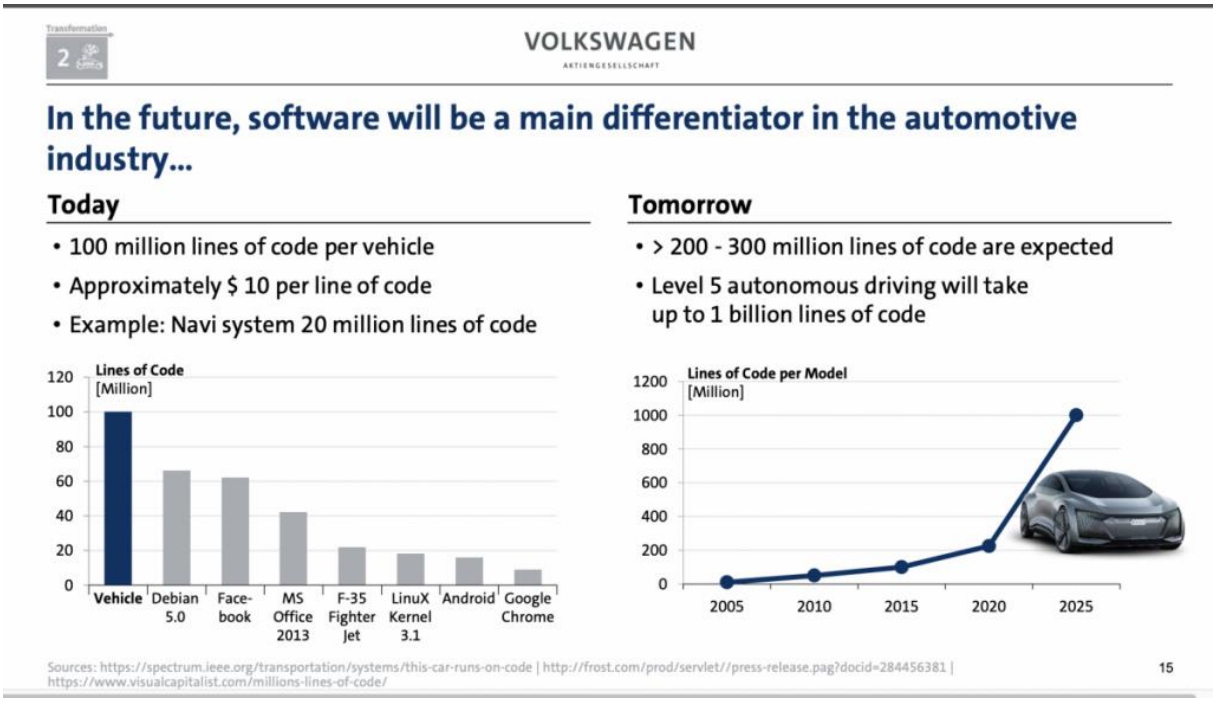


Figura 2 Slide from a 2020 presentation by Herbert Diess highlights the VW Group's software ambitions

Questa complessità non riguarda solo il numero di nodi e la dimensione del software, ma anche la **topologia dell'architettura elettronica**. Le prime generazioni di veicoli avevano un'**architettura distribuita**, con ECU dedicate a singole funzioni

e cablaggi estesi e complessi. Questa soluzione, pur semplice da sviluppare inizialmente, si è rivelata inefficiente e poco scalabile: con l'aumento delle funzioni, il cablaggio è diventato eccessivamente pesante e costoso, e la gestione dei segnali sempre più complessa.
Per superare questi limiti, si è progressivamente passati ad **architetture a dominio**, in cui le ECU vengono raggruppate in base alla funzione (es. powertrain, chassis, infotainment, ADAS, body control). Ogni dominio è gestito da un controller centrale che raccoglie e coordina i dati delle ECU locali.

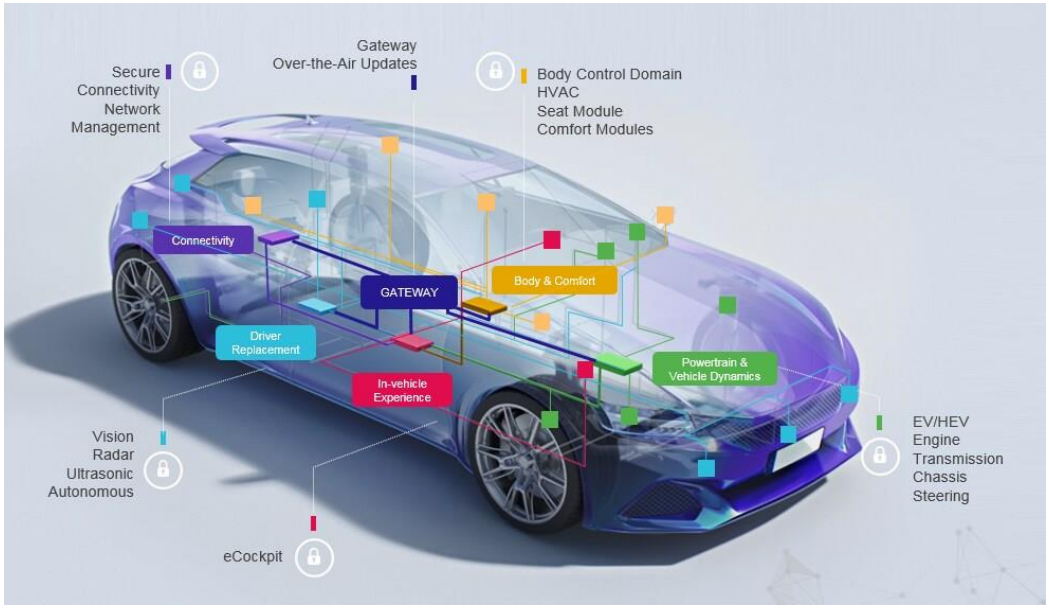


Figura 3 NXP illustration on vehicle architecture distribution [4]

Tale passaggio è motivato da esigenze operative precise: i veicoli moderni devono gestire non soltanto un'enorme mole di dati, ma anche garantirne la trasmissione in modo tempestivo e affidabile. Oggi, la tendenza più recente è quella verso **architetture zonali**, in cui le funzioni non sono più raggruppate per tipologia ma per posizione fisica nel veicolo (zona anteriore, posteriore, laterale, cabina, ecc.). Ogni Zonal ECU raccoglie i segnali dei dispositivi situati nella propria area e li inoltra a un **High Performance Computer (HPC)** centrale, responsabile dell'elaborazione ad alto livello e della gestione del software applicativo.
Questo approccio consente:

- riduzione significativa del cablaggio (meno cavi, meno peso);
- maggiore **modularità** (facilità di manutenzione e aggiornamento);
- **scalabilità** migliorata, poiché nuovi moduli possono essere aggiunti con minori modifiche all'architettura esistente;
- gestione più **efficiente** del software, grazie alla centralizzazione su piattaforme di calcolo più potenti.

2.2 Software-defined vehicle e implicazioni

Con la crescente complessità del software e l'introduzione di HPC, si sta affermando il concetto di **Software-Defined Vehicle (SDV),** in cui le funzionalità del veicolo non sono più determinate dall'hardware, ma dallo *stack software* installato. Gli aggiornamenti possono così introdurre nuove funzioni senza modificare l'hardware di base, abilitando modelli di business come *feature on demand* o aggiornamenti OTA continui.

Questa evoluzione implica:

- adozione di **sistemi operativi real-time (RTOS)** sui microcontrollori e di sistemi multitasking su processori ad alte prestazioni;
- utilizzo di **middleware standardizzati** come AUTOSAR Classic e AUTOSAR Adaptive, per gestire la complessità e garantire portabilità;
- necessità di **ottimizzare i flussi dati tra i vari livelli**, dal firmware embedded ai processi di alto livello, come nel caso del lavoro oggetto di questa tesi.

2.3 Le reti di comunicazione

Il passaggio verso architetture più centralizzate non sarebbe possibile senza reti di comunicazione affidabili e deterministiche. Le reti veicolari permettono alle diverse ECU di scambiarsi informazioni in tempo reale e coordinare le funzioni del veicolo.
I principali protocolli in uso sono:

- **CAN** (Controller Area Network): lo standard più diffuso, robusto e adatto a segnali critici in tempo reale (motore, freni, trasmissione).
- **Automotive Ethernet**: introdotto per funzioni ad alto volume di dati, come telecamere, radar, sistemi ADAS e infotainment, *sta man mano diventando sempre più di largo utilizzo*.
- **LIN** (Local Interconnect Network): usato per sottosistemi meno critici, come la regolazione dei sedili o l'illuminazione interna.
- **FlexRay**: protocollo ad alta affidabilità per funzioni di sicurezza, oggi *in parziale declino* a favore di Ethernet.

Ogni rete ha caratteristiche specifiche in termini di velocità, priorità dei messaggi, costo e robustezza.
Nel caso del CAN, ad esempio, la comunicazione è basata su un bus multi-master con arbitraggio a priorità, in cui i messaggi con identificatore più basso hanno precedenza. La velocità tipica varia da 125 kbps a 1 Mbps, mentre la variante CAN FD (Flexible Data-rate) consente di arrivare fino a 8 Mbps e di trasmettere frame fino a 64 byte, riducendo l'overhead e aumentando l'efficienza del bus.

L'introduzione di Automotive Ethernet (100BASE-T1, 1000BASE-T1) ha aperto la strada a un nuovo paradigma, in cui il veicolo diventa una vera e propria rete IP interna, capace di gestire simultaneamente streaming video, diagnostica remota e

aggiornamenti OTA (Over-The-Air). Tuttavia, la coesistenza di reti eterogenee rende necessaria una gestione accurata della gateway architecture, ovvero delle ECU che traducono e instradano i messaggi tra protocolli diversi.

2.4 Il protocollo CAN (Controller Area Network)

Il **Controller Area Network (CAN)** è lo standard di comunicazione più diffuso nei sistemi embedded automotive. Sviluppato da Bosch nel 1986 e formalizzato come ISO 11898, è stato progettato per consentire la comunicazione affidabile fra più centraline elettroniche (ECU) senza la necessità di una topologia master-slave.

Il CAN è un **bus seriale multi-master e message-oriented**, dove ogni nodo può trasmettere e ricevere messaggi in modo asincrono, condividendo lo stesso mezzo trasmissivo. La sua robustezza, affidabilità e basso costo hanno reso questa tecnologia lo standard de facto per la comunicazione veicolare.

2.4.1 Architettura del bus CAN

Tutti i nodi di una rete CAN sono collegati tra loro tramite un bus differenziale a **due fili intrecciati,** denominati **CAN High (CANH)** e **CAN Low (CANL)**. Proprio la comunicazione differenziale fornisce un'alta resistenza a disturbi elettrici esterni, fondamentali su un autoveicolo.
Ogni nodo è costituito da tre componenti principali:

- un **microcontrollore (MCU)**, che gestisce la logica applicativa;
- un **controller CAN**, spesso integrato nel microcontrollore stesso, che si occupa della codifica/decodifica dei frame;
- un **transceiver**, che converte i segnali logici (TXD/RXD) in segnali differenziali su CANH e CANL.

Questa architettura consente di collegare nodi di complessità molto variabile — da sensori digitali fino a gateway di rete o centraline di calcolo avanzato.

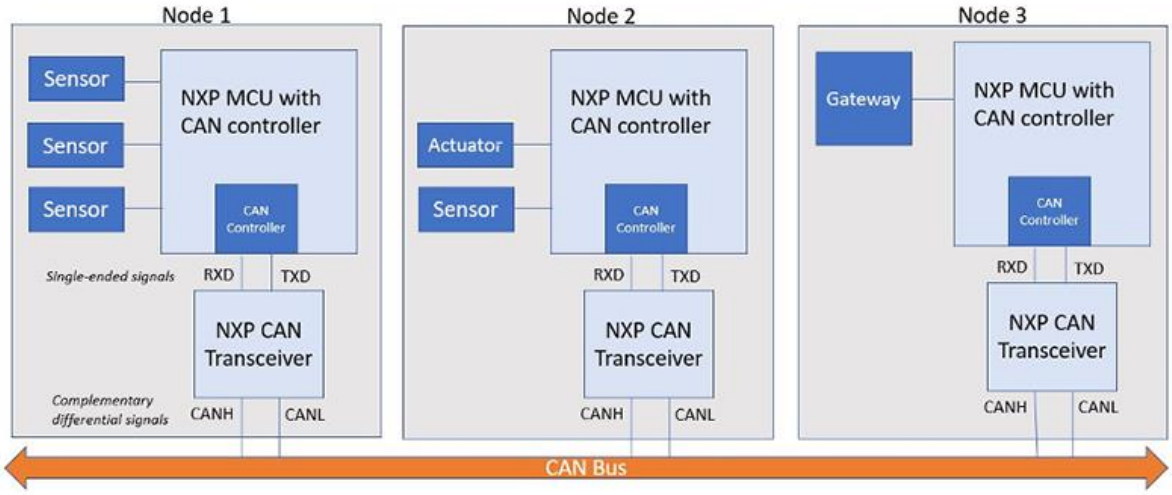


Figura 4 Architettura tipica di una rete CAN [5]

2.4.2 Struttura del frame CAN

Un **frame CAN** è l'unità base di comunicazione sul bus. Tutti i nodi ascoltano costantemente il bus e, quando il mezzo è libero, uno di essi può iniziare una trasmissione. Ogni frame contiene diversi campi funzionali, come mostrato in Figura 5

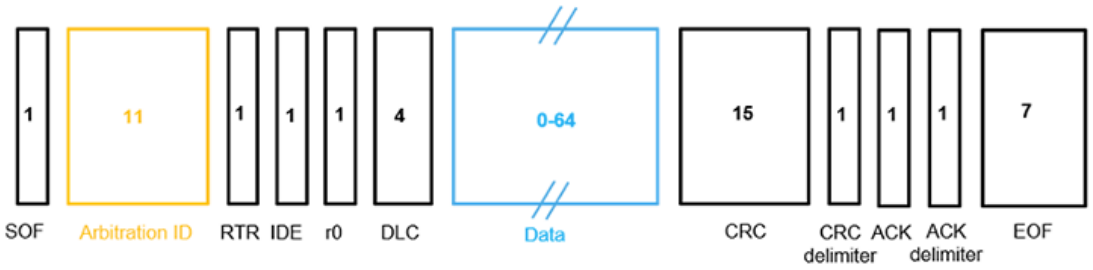


Figura 5 Struttura di un frame CAN standard [5]

Campo	Descrizione	Note
SOF	Start of Frame – bit dominante che segnala l'inizio del messaggio	1 bit
ID	Identifica il messaggio e la sua priorità	11 o 29 bit
RTR	Remote Transmission Request – bit recessivo se il frame è di richiesta dati	1 bit
IDE	Indica formato standard (11 bit) o esteso (29 bit)	1 bit
r0 / FDF	Riservato; indica un frame CAN-FD	1 bit
DLC	Data Length Code – numero di byte nel campo dati	4 bit
DATA	Dati effettivi	0–8 byte (fino a 64 in CAN FD)
CRC	Cyclic Redundancy Check – controllo d'errore	15 + 1 bit
ACK	Bit di conferma dai nodi riceventi	1+1 bit
EOF	End of Frame – 7 bit recessivi	7 bit

2.4.3 Funzionamento ed arbitraggio

Il protocollo CAN utilizza un meccanismo denominato arbitraggio non distruttivo a livello di bit (**lossless bitwise arbitration**) per determinare la priorità dei messaggi. Ogni nodo della rete è in grado sia di trasmettere sia di ricevere messaggi; tuttavia, **solo un messaggio alla volta può occupare il bus**.

L'accesso al mezzo è di tipo **event-driven**, per cui può accadere che più nodi tentino di iniziare una trasmissione nello stesso istante.

In tali casi, il protocollo stabilisce automaticamente quale messaggio debba avere la precedenza: il messaggio con priorità più alta "vince" l'arbitraggio e ottiene il controllo del bus.

La priorità è determinata confrontando i bit dell'identificativo del messaggio (Arbitration ID) uno alla volta (bitwise):
un valore binario più basso corrisponde a priorità più alta, poiché il bit logico '0' (detto dominante) prevale sul bit logico '1' (detto recessivo).

Il nodo che "vince" l'arbitraggio prosegue la trasmissione senza ritardi, perdita o corruzione del messaggio, mentre il nodo che trasmetteva un messaggio a priorità inferiore interrompe immediatamente l'invio. Una volta che il bus torna libero, quest'ultimo tenterà nuovamente la trasmissione.

Per chiarire il principio, si considerino due dispositivi che iniziano a trasmettere simultaneamente sul bus CAN.
Il Dispositivo A ha un identificativo di arbitraggio pari a 11001000111, mentre il Dispositivo B ha un identificativo 11011111111.
Poiché il quarto bit del messaggio di A è 0 (dominante), mentre quello di B è '1' (recessivo), il messaggio di A prevale e ottiene il controllo del bus.
Il dispositivo A quindi completa la trasmissione, mentre il dispositivo B attende che la linea sia nuovamente libera per ripetere il tentativo.

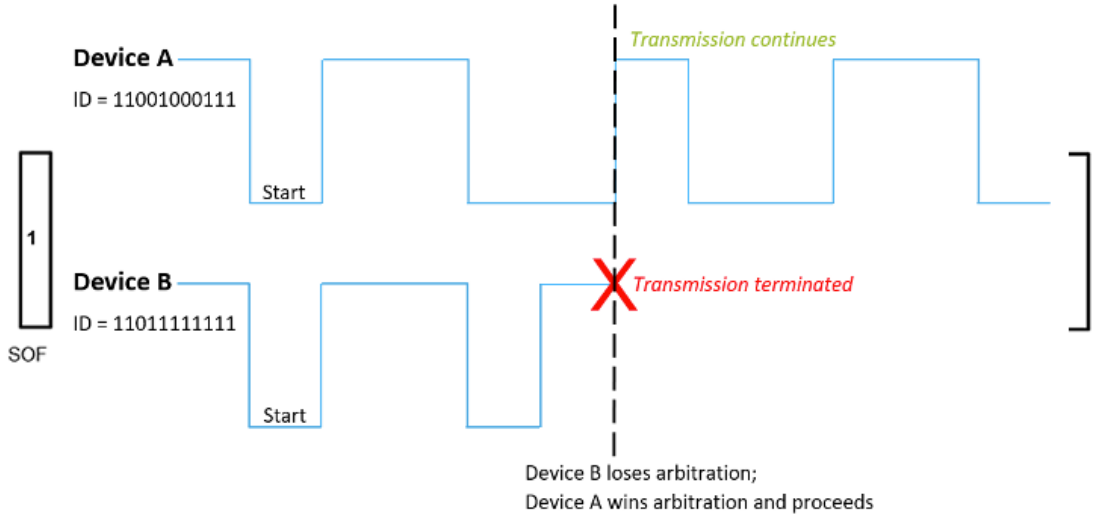


Figura 6 Esempio di arbitraggio di priorità nel bus CAN [5]

In aggiunta, oltre a identificare univocamente il messaggio, il campo di arbitraggio (ID) definisce anche la priorità con cui esso può accedere al bus: ID più basso → priorità più alta.
Quando più nodi trasmettono contemporaneamente, il protocollo assicura che solo il messaggio con priorità maggiore proceda, garantendo un accesso ordinato, deterministico (latenza massima prevedibile in base ai messaggi con priorità superiore già presenti sulla rete) e privo di collisioni distruttive. [5]

2.4.4 Tipologie di frame nel protocollo CAN

Un messaggio trasmesso sulla rete CAN (Controller Area Network) è denominato frame. Il protocollo definisce quattro tipologie principali di frame, ciascuna con una funzione specifica all'interno della comunicazione tra nodi.

- **Data frame:** rappresenta la tipologia più comune e costituisce l'unico frame contenente dati effettivi. La sua struttura comprende:
  - **Arbitration ID**, che identifica la priorità del messaggio;
  - **Campo dati**, contenente le informazioni da trasmettere;
  - **CRC,** utilizzato per il rilevamento degli errori;
  - **ACK,** che conferma la corretta ricezione del messaggio da parte dei nodi.
- **Remote Frame:** consente a un nodo di richiedere dati specifici a un altro nodo della rete. Le differenze principali rispetto al Data frame sono due:
  - la presenza di un bit RTR recessivo all'interno del campo di arbitraggio, che identifica esplicitamente il frame come richiesta;
  - l'assenza del campo dati, poiché il frame non trasporta informazioni ma funge esclusivamente da richiesta.
- **Error Frame:** Quando un nodo rileva un'anomalia, trasmette un Error frame a tutti gli altri nodi della rete. Questi, a loro volta, propagano lo stesso frame, garantendo che l'errore venga riconosciuto globalmente. Per evitare un ciclo infinito di messaggi di errore (bus flooding), i transceiver CAN implementano contatori hardware di errore, che limitano la ritrasmissione automatica di tali frame.
- **Overload Frame:** ha la funzione di introdurre un ritardo tra un Data o Remote frame e il successivo, richiedendo implicitamente la ritrasmissione del messaggio in un secondo momento. Tuttavia, con l'evoluzione dei controller CAN, questa tipologia è stata progressivamente abbandonata: i dispositivi moderni sono infatti in grado di gestire autonomamente i tempi di elaborazione, evitando così un traffico superfluo sul bus.

2.4.5 Rilevamento degli errori nel protocollo CAN

La robustezza del bus CAN deriva in larga parte dall'ampia gamma di meccanismi di rilevamento degli errori che esso implementa. In particolare, il protocollo CAN integra cinque metodi di controllo, di cui **tre a livello di messaggio** e **due a livello di bit**. Qualora un messaggio non superi anche solo uno di questi controlli, esso viene scartato e il nodo ricevente genera un **Error frame**.

- **Controlli a livello di messaggio**
  - **CRC e ACK**: i campi Cyclic Redundancy Check e Acknowledgement includono rispettivamente un checksum e bit delimitatori, che consentono di verificare l'integrità del messaggio e la corretta ricezione da parte dei nodi.
  - **Form check**: alcuni campi del messaggio devono obbligatoriamente contenere bit recessivi. Tra questi rientrano l'End of Frame (EOF), il delimitatore dell'ACK e quello del CRC. L'eventuale rilevamento di un bit dominante in tali posizioni genera un errore.
- **Controlli a livello di bit**
  - **Bit monitoring**: i trasmettitori monitorano i propri messaggi bit per bit. Se il valore scritto sul bus non coincide con quello letto, viene generato un errore.
  - **Bit stuffing**: per garantire la sincronizzazione della rete, dopo cinque bit consecutivi dello stesso livello logico viene inserito automaticamente un bit di livello opposto. I nodi riceventi provvedono poi a rimuovere i bit "stuffed". Questa regola si applica a tutti i campi del frame, ad eccezione del delimitatore CRC, del campo ACK e dell'EOF. La presenza di sei bit consecutivi della stessa polarità costituisce una violazione della regola di bit stuffing e determina la generazione di un errore.

2.4.6 Svantaggi del protocollo CAN

Nonostante la sua ampia diffusione e i numerosi punti di forza, il **CAN classico** presenta alcune limitazioni che ne riducono l'efficienza in applicazioni moderne e ad alta intensità di dati.

- **Overhead significativo**: uno dei principali svantaggi del CAN classico è l'elevato rapporto tra bit di controllo e bit utili. A fronte di un payload massimo di soli **8 byte**, la struttura del frame include numerosi campi di arbitraggio, controllo e delimitazione. Questo comporta un **overhead proporzionalmente molto alto**, che riduce l'efficienza della trasmissione, soprattutto quando i messaggi devono essere inviati con elevata frequenza o contengono informazioni numericamente consistenti.
- **Larghezza di banda limitata**: la velocità massima di trasmissione è pari a **1 Mbps**, un valore adeguato a molte applicazioni tradizionali, ma insufficiente per sistemi avanzati che richiedono throughput più elevati.
- **Scarsa scalabilità**: all'aumentare del numero di nodi e del traffico sul bus, cresce la probabilità di ritardi dovuti all'arbitraggio delle priorità, con possibili conseguenze negative sulla prevedibilità temporale in applicazioni real-time.
- **Assenza di meccanismi di sicurezza nativi**: il protocollo non integra funzioni di autenticazione o crittografia, rendendo necessarie soluzioni esterne per garantire la protezione da attacchi informatici. [6]

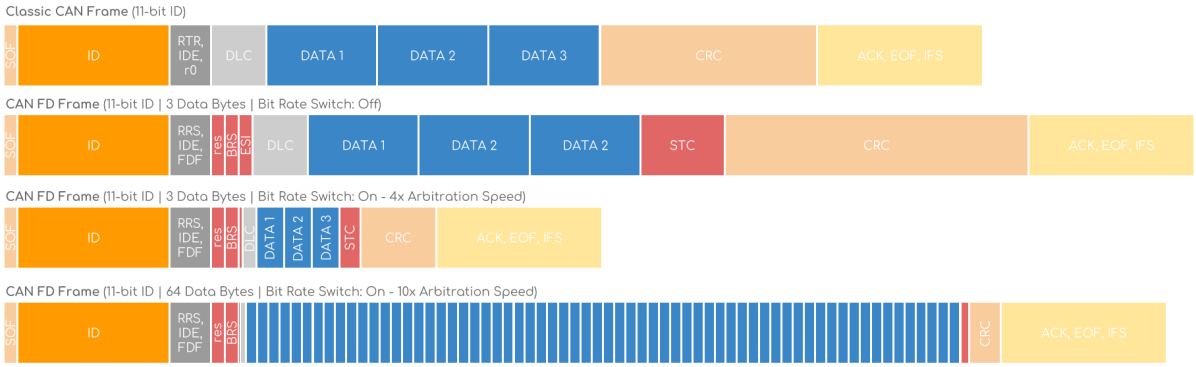


Figura 7 comparazione di diversi frame classici ed FD [7]

Nel contesto specifico delle **attività di tirocinio e della presente tesi**, è importante sottolineare che non è stato adottato il **CAN FD (Flexible Data Rate)**, il quale avrebbe permesso di estendere il payload fino a **64 byte** e di incrementare la velocità di trasmissione. A mio avviso, questa scelta ha rappresentato una **limitazione significativa**, poiché l'uso del CAN classico ha imposto vincoli stringenti in termini di efficienza e capacità di trasferimento dati, influenzando direttamente le possibilità di ottimizzazione del sistema sviluppato.

3 Piattaforme hardware

Nei veicoli moderni, il software e l'elettronica hanno un ruolo centrale, e la scelta della piattaforma hardware è determinante per le prestazioni e l'affidabilità del sistema. In questo contesto, due categorie principali di dispositivi emergono: **microcontrollori (MCU)** e **microprocessori (MPU/CPU)** anche se molto spesso vedremo come gli MPU in ambito automotive siano realmente dei **System on Chip (SoC).**

3.1 Microcontroller Unit e Microprocessor Unit

Un **microcontrollore (MCU, MicroController Unit)** è un circuito integrato che combina al suo interno CPU, memoria e periferiche di input/output ed è progettato per controllare dispositivi elettronici in modo autonomo e deterministico. A differenza dei microprocessori, le MCU sono pensate per applicazioni embedded, dove il controllo diretto di sensori, attuatori e altri moduli elettronici è fondamentale, con vincoli stringenti su tempi di risposta, consumi energetici e spazio fisico.

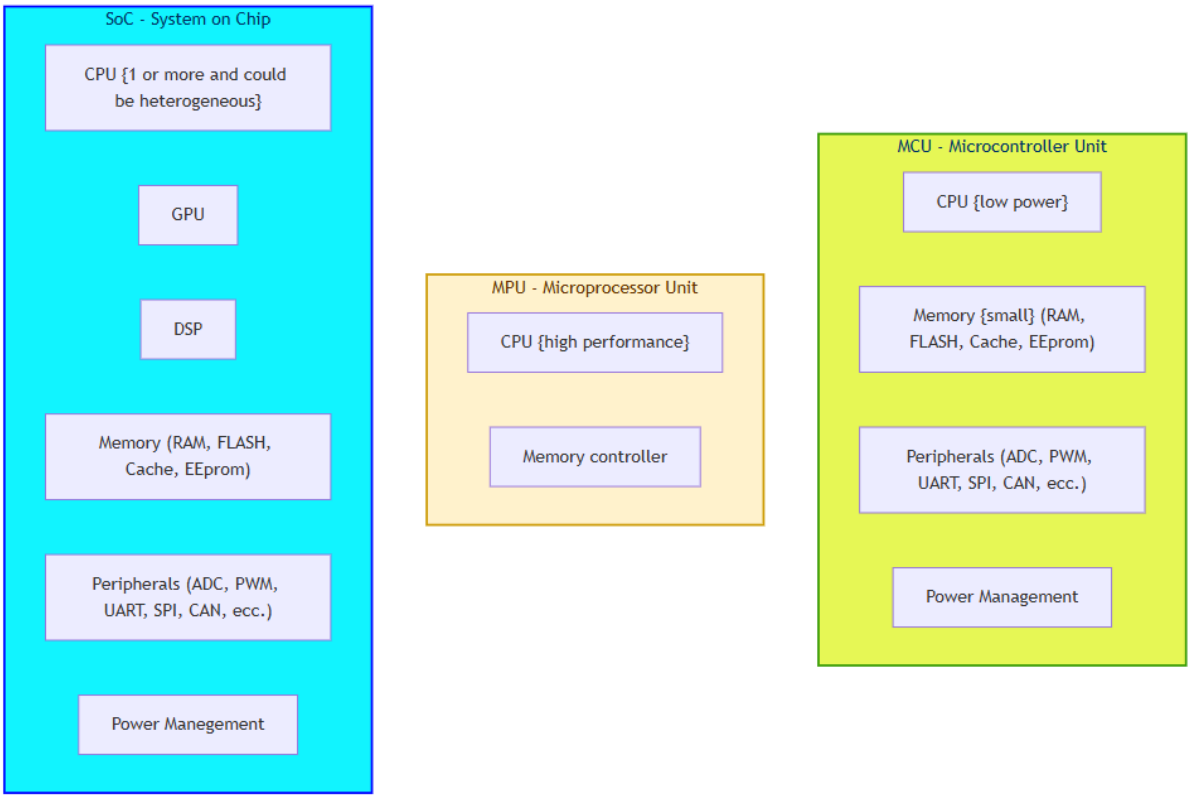


Figura 8 MCU, MPU, SoC

3.1.1 Componenti principali di un microcontrollore

Secondo la classificazione proposta da IBM [8], i microcontrollori possono essere descritti attraverso i seguenti blocchi funzionali:

- **Unità di elaborazione (CPU)**: esegue le istruzioni del programma e coordina le operazioni interne.

- **Memoria**:
    - **RAM** per i dati temporanei,
    - **Flash/EEPROM** per il firmware, che rimane memorizzato anche senza alimentazione.
- **Periferiche integrate**: moduli hardware che permettono l'interazione diretta con il mondo esterno, come convertitori analogico/digitali, timer, contatori, interfacce seriali UART ed altri protocolli di comunicazione come SPI e CAN.
- **Moduli ausiliari**: in alcune MCU avanzate possono essere presenti controller per display LCD, Ethernet, USB, ecc.

3.1.2 Tipologie di microcontrollori

L'evoluzione tecnologica ha portato alla nascita di diverse famiglie di MCU, differenziate per architettura e capacità [8]:

- **8-bit**: soluzioni semplici, a basso costo, usate in piccoli elettrodomestici, giocattoli e telecomandi.
- **16-bit**: più potenti, adatte a dispositivi medicali, sistemi industriali e applicazioni automotive di medio livello.
- **32-bit**: le più diffuse oggi, basate principalmente su architetture **ARM Cortex-M**, utilizzate in robotica, automotive, IoT e sistemi industriali avanzati.
- **RISC (Reduced Instruction Set Computer)**: architetture semplificate che eseguono istruzioni più velocemente, come gli **ARM (Advanced RISC Machines)**, oggi standard de facto nel settore.
- **PIC (Peripheral Interface Controller)**: microcontrollori compatti e versatili, molto usati in automazione e robotica.
- **DSP/FPGA-based MCU**: soluzioni ibride per applicazioni che richiedono elaborazione di segnali, video processing o networking ad alta velocità.

3.1.3 Microprocessor Unit (MPU)

Tradizionalmente, un MPU viene definito come un'unità di calcolo pura, progettata per eseguire istruzioni e coordinare le operazioni logiche e aritmetiche di un sistema. In questa concezione classica, il microprocessore non integra né memoria né periferiche: per funzionare necessita di componenti esterni, come RAM, memoria non volatile e controller di I/O. Questa architettura modulare è tipica dei computer general-purpose, dove la CPU è separata dagli altri sottosistemi e si limita a svolgere il ruolo di "cervello" centrale.

Tuttavia, nel mondo **embedded** e in particolare in ambito automotive, questa distinzione netta tende a sfumare. Le esigenze di riduzione dei costi, dei consumi energetici e della complessità di integrazione hanno portato all'evoluzione verso **System-on-Chip (SoC)**, in cui anche i microprocessori includono al loro interno blocchi funzionali che un tempo erano esclusivamente appannaggio delle MCU.

Oggi molte MPU destinate all'automotive integrano:

- **memorie on-chip** (cache di grandi dimensioni, RAM integrata, talvolta anche Flash),
- **periferiche di comunicazione** (CAN, Ethernet automotive, USB, PCIe),
- **moduli grafici o DSP dedicati** per l'elaborazione di segnali e immagini,
- **core eterogenei**: accanto ai core ad alte prestazioni (ARM Cortex-A, tipici delle MPU) vengono spesso affiancati core a basso consumo e deterministici (ARM Cortex-M), che svolgono il ruolo di microcontrollori integrati.

Questa architettura eterogenea consente di combinare in un unico dispositivo la **potenza di calcolo elevata** necessaria per applicazioni complesse (infotainment, ADAS, sensor fusion) con la **deterministicità real-time** tipica delle MCU, indispensabile per la gestione di periferiche critiche e funzioni di sicurezza.
In ambito automotive, tali soluzioni vengono spesso utilizzate come **piattaforme centralizzate**: la parte "MPU" esegue sistemi operativi complessi (Linux, QNX, AUTOSAR Adaptive) e gestisce applicazioni ad alto livello, mentre i core "MCU-like" integrati si occupano di attività a bassa latenza, come la gestione del bus CAN o il monitoraggio di funzioni di sicurezza.
In questo modo, anche se per definizione un microprocessore dovrebbe essere un sistema privo di periferiche e memoria, nella pratica delle applicazioni embedded moderne si tende a **integrare tutto su un singolo chip**, dando vita a soluzioni ibride che uniscono i vantaggi di MCU e MPU in un'unica piattaforma. Vedi la Figura 8 per avere un paragone visuale di queste 3 strutture.

3.2 Board utilizzate

Dopo aver introdotto le differenze concettuali tra microcontrollori e microprocessori, è utile descrivere le piattaforme hardware concretamente impiegate nello sviluppo del progetto. L'attività, infatti, si è articolata in più fasi, ciascuna caratterizzata dall'uso di una scheda differente, scelta in base alle esigenze di sperimentazione, budget e portabilità del firmware.

3.2.1 NXP i.MX8QM MEK

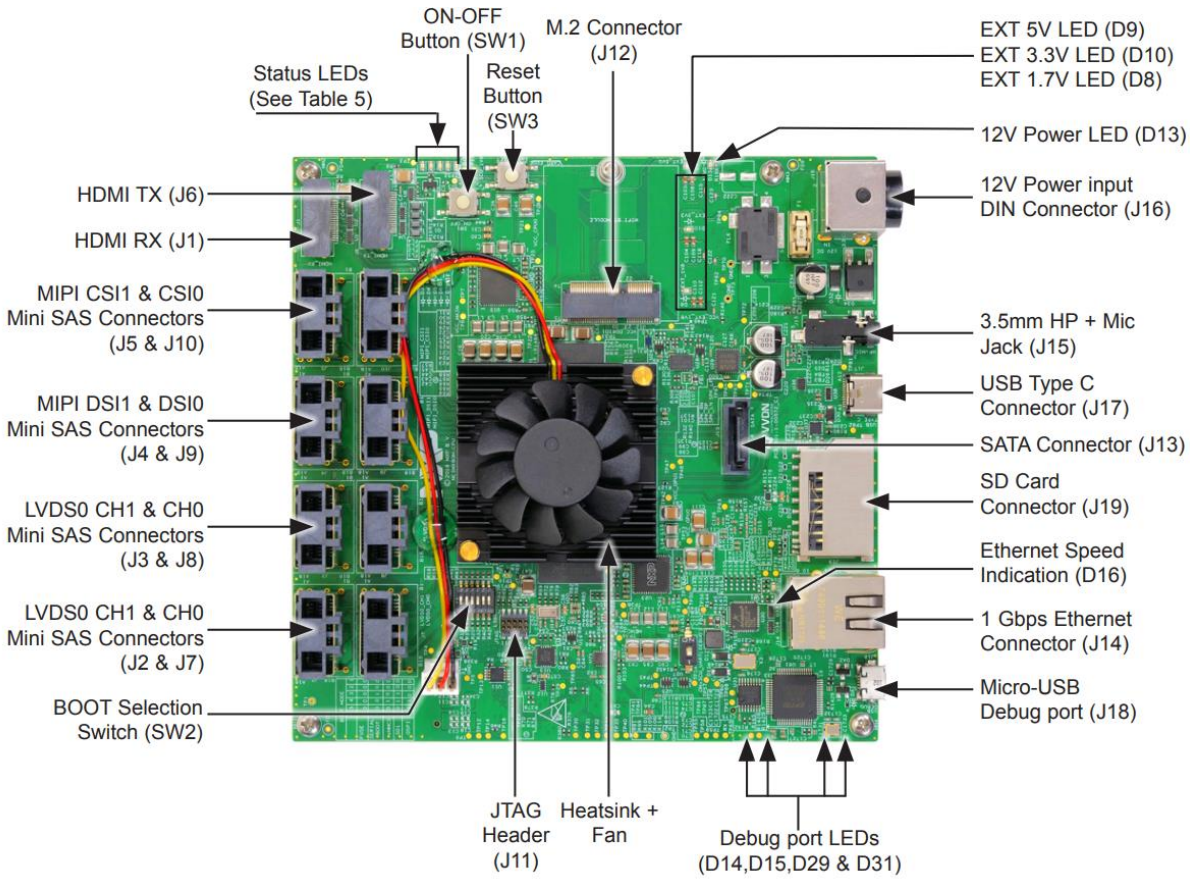


Figura 9 i.MX 8QM MEK [9]

La prima piattaforma utilizzata è stata la **NXP i.MX8QM Multisensory Enablement Kit (MEK)**, una scheda di sviluppo basata su un SoC eterogeneo con core ARM Cortex-A ad alte prestazioni e core Cortex-M integrati. Questa board rappresentava l'hardware originale del progetto e costituiva il punto di partenza per la mia attività di sviluppo. Il lavoro iniziale ha riguardato la **modifica e l'adattamento del firmware esistente**, con l'obiettivo di comprenderne la struttura, studiarne i punti forti/deboli e di predisporre il codice per una successiva portabilità su diverse piattaforme. La scheda è stata utilizzata in abbinamento a un modulo di espansione per il **CAN bus**, necessaria per la comunicazione con altri dispositivi non avendo nativamente i transceiver CAN integrati.
Per una descrizione più dettagliata è più intuitivo con attenzione la struttura interna dell'i.MX 8 rappresentata in Figura 10, dove possiamo notare la presenza di 6 core

ad alte prestazioni (4+2) designati al livello applicativo e 2 core a "basse prestazioni" per applicazioni Real Time.

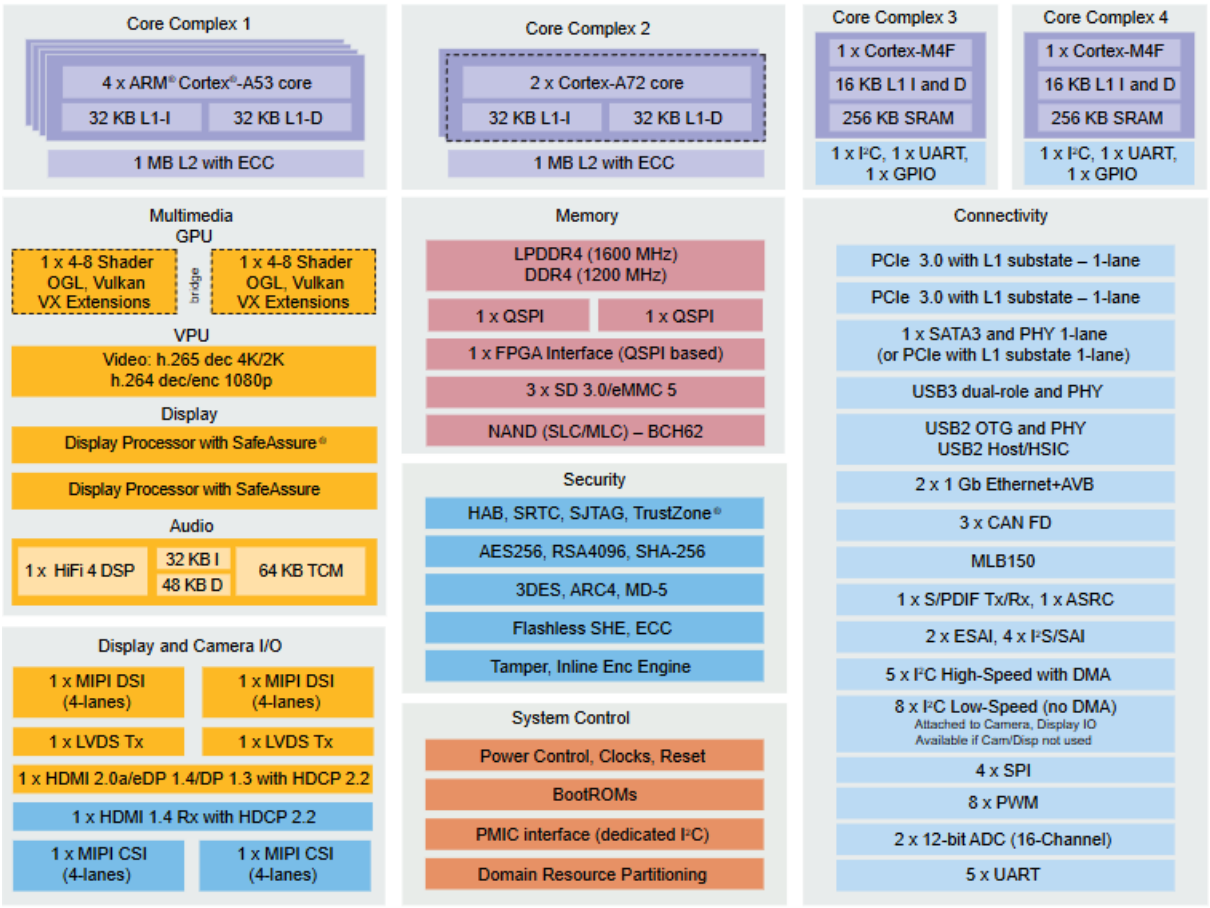
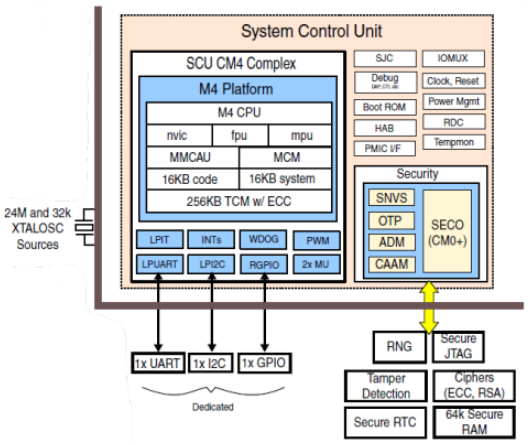


Figura 10 i.MX8 SoC structure [10]

Inoltre, anche se poco documentato e non presente nell'immagine appena vista l'i.MX8 presenta un terzo cortex-M4 ed un cortex-M0 "non utilizzabile" [11].



- System Controller Unit subsystem is comprised of
  - 1x Cortex-M4 processor
  - A set of peripherals
    - 1x TPM, 1x UART, 1x I2C, 8x GPIOs, 4x MUs
  - This is the first processor to boot in the design
- Security subsystem is made of
  - 1x Cortex-M0 processor running at 133MHz
    - Handles security services for other processors
  - A set of cryptographic related hardware accelerators
    - RSA, ECC, AES, DES/3DES, SHA-1, SHA-2, MD5, HMAC, RNG

Figura 11 SCU [11]

L'M4 svolge esclusivamente il ruolo di **System Control Unit (SCU)** con il suo relativo

System Control Unit Firmware (SCFW), firmware non completamente disponibile ma di cui è possibile modificare alcune parti e ricompilarle per personalizzare leggermente il comportamento della board.
Ulteriori approfondimenti verranno trattati nel capitolo dedicato ai problemi riscontrati e allo sviluppo del software; per una trattazione completa, comprendente gli aspetti hardware e software di SCU ed SCFW, si rimanda al documento [11] .

3.2.2 NXP S32K312 EVB

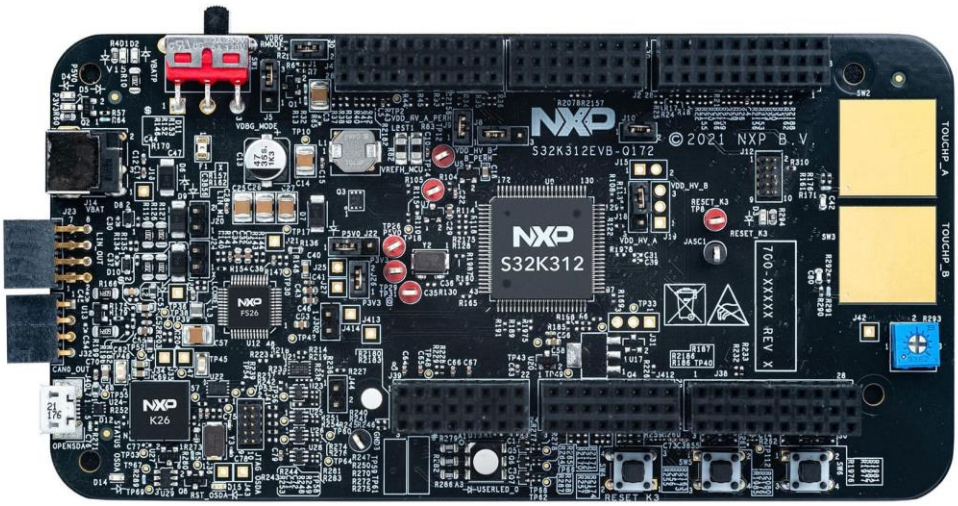


Figura 12 S32k312 [12]

La **S32K312EVB-Q172** è una scheda di valutazione sviluppata da NXP per applicazioni automotive basate sulla famiglia di microcontrollori **S32K3**. A differenza di piattaforme più complesse come la i.MX8, questa board non integra processori a livello applicativo (ad esempio core ARM Cortex-A), ma si basa esclusivamente su un **ARM Cortex-M7 a 32 bit**, progettato per garantire affidabilità, basso consumo e capacità di elaborazione real-time.
Questa caratteristica la colloca pienamente nella categoria delle **MCU automotive**, pensate per il controllo diretto di periferiche e la gestione deterministica di segnali e comunicazioni, piuttosto che per l'esecuzione di sistemi operativi complessi.
Dal punto di vista hardware, la scheda offre:

- alimentazione flessibile 12-48V certificata per sicurezza funzionale;
- interfacce di comunicazione tipiche del settore automotive, come **CAN FD** e **LIN**;
- periferiche di test a bordo (LED, pulsanti, potenziometro ADC, pad capacitivi);
- slot compatibile **Arduino UNO** per l'espansione con shield esterni.

Sul piano software, la board è supportata da driver **AUTOSAR e non-AUTOSAR**, librerie di sicurezza e tool di sviluppo integrati nell'ambiente **S32 Design Studio**, con estensioni per MATLAB/Simulink.
Nel contesto del progetto, la **S32K312EVB** era stata inizialmente selezionata come piattaforma di migrazione del firmware, con l'obiettivo di verificare la portabilità del codice da un ambiente basato su MPU (i.MX8) a un microcontrollore puro. Sono stati avviate le prime fasi del porting, ma la scheda è stata **abbandonata in una fase precoce**, principalmente per concentrare gli sforzi su una piattaforma più adatta all'integrazione finale e per altri motivi interni all'azienda.
Il suo ruolo, quindi, è rimasto marginale: utile come fase esplorativa per valutare le differenze tra un sistema eterogeneo ibrido ed uno puramente real-time.

3.2.3 RT1064

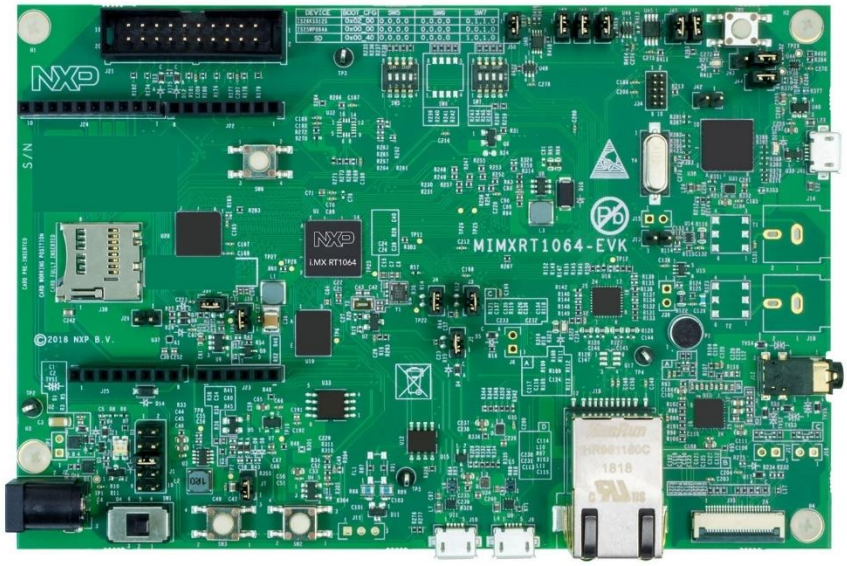


Figura 13 i.MX RT1064 Evaluation Kit [13]

La piattaforma che ha avuto un ruolo centrale nello sviluppo del progetto è stata la **NXP MIMXRT1064-EVK**, scheda di valutazione basata sul microcontrollore **i.MX RT1064**, appartenente alla famiglia delle cosiddette *crossover MCU*. Questi dispositivi rappresentano una categoria intermedia tra microcontrollori e microprocessori: pur mantenendo le caratteristiche tipiche delle MCU (architettura deterministica, basso consumo, periferiche integrate), offrono frequenze di clock e capacità di memoria paragonabili a quelle di sistemi più complessi.
Il cuore della scheda è un **ARM Cortex-M7** operante fino a **600 MHz**, che la colloca tra le MCU a più alte prestazioni disponibili sul mercato. Il microcontrollore integra **4 MB di Flash on-chip** e **1 MB di RAM interna**, di cui una parte configurabile come **TCM (Tightly Coupled Memory)** per garantire latenze estremamente ridotte nelle applicazioni real-time. Questa caratteristica ha reso la piattaforma

particolarmente adatta al porting del firmware, riducendo la dipendenza da memorie esterne e semplificando la gestione del codice.
Dal punto di vista delle periferiche, la MIMXRT1064 offre un set molto ricco:

- **2 interfacce Ethernet 10/100 con supporto IEEE 1588**,
- **2 controller USB 2.0 OTG con PHY integrato**,
- **fino a 8 UART, 4 I²C, 4 SPI**,
- **2 moduli FlexCAN con supporto CAN FD**,
- **interfacce per memoria esterna** (SDRAM, NAND, NOR, eMMC, QuadSPI),
- **moduli audio e grafici** (SAI, SPDIF, LCD controller, 2D graphics engine),
- **convertitori ADC a 12 bit** e comparatori analogici.

La scheda di valutazione **MIMXRT1064-EVK** integra inoltre:

- **256 Mb di SDRAM esterna**,
- **512 Mb di HyperFlash** e **64 Mb di QSPI Flash**,
- connettori per **LCD parallelo**, **camera**, **audio codec** e **CAN transceiver**,
- compatibilità con lo **standard Arduino UNO pinout**, che ne facilita l'espandibilità.

3.3 Porting

Nel contesto dello sviluppo software per sistemi embedded, il termine **porting** indica il processo di **trasferimento e adattamento di un'applicazione o di un firmware da una piattaforma hardware a un'altra**, mantenendone il comportamento funzionale e le prestazioni attese. Questo processo non si limita alla semplice ricompilazione del codice sorgente, ma richiede una **revisione approfondita dell'interazione tra software e hardware**, in particolare per quanto riguarda la gestione delle periferiche, dei driver e delle risorse di sistema.
Il porting diventa necessario quando si cambia architettura (ad esempio da una MPU a una MCU), si sostituisce la scheda di sviluppo, oppure si desidera rendere il software compatibile con più target hardware. In ambito embedded, dove il software è spesso strettamente legato all'hardware sottostante, il porting può comportare modifiche significative a livello di:

- **driver di basso livello**,
- **configurazione dei registri e delle periferiche**,
- **allocazione della memoria**,
- **gestione del clock e delle interruzioni**,
- **compatibilità con toolchain e ambienti di sviluppo differenti**.

Nel settore automotive, il porting assume un ruolo strategico: consente di riutilizzare codice già validato su nuove piattaforme, riducendo i tempi di sviluppo e facilitando la scalabilità del software tra diverse ECU. Tuttavia, comporta anche sfide legate alla sicurezza funzionale e alla gestione di ambienti eterogenei, come nel caso di SoC che integrano core di tipo MPU e MCU.
Nel progetto descritto in questa tesi, il porting ha rappresentato una fase cruciale: partendo da un firmware originariamente sviluppato per una piattaforma MPU/SoC

(i.MX8), si è proceduto alla sua migrazione verso microcontrollori più leggeri (S32K312, i.MX RT1064), affrontando le problematiche legate alla compatibilità, alla disponibilità dei driver e alla gestione delle periferiche. Le soluzioni adottate e le scelte architetturali saranno descritte nei paragrafi successivi.

3.4 Problemi collegati all'hardware

Durante l'integrazione della **MIMXRT1064-EVK** con il **Raspberry Pi 4** sono emerse due principali criticità. Da un lato, sul piano hardware, la comunicazione SPI non funzionava inizialmente perché gli header della scheda non erano collegati ai pin dell'MCU: nello schematico erano infatti presenti resistenze da 0 Ω marcate come *DNP* (Do Not Place).
Dall'altro lato, sul piano software, è stato riscontrato che nel **BSP di QNX per Raspberry Pi** il driver SPI era implementato solo in modalità **master**, mentre anche la RT1064 era stata configurata come master. È stato quindi necessario **invertire i ruoli**, mantenendo il Raspberry Pi come master e riconfigurando la RT1064 come slave, così da rendere possibile la comunicazione.
I dettagli tecnici e le implicazioni di queste scelte verranno approfonditi nel Capitolo 6.

4 Organizzazione software

4.1 Modelli architetturali

Nello sviluppo di sistemi embedded complessi, la scelta di un modello architetturale adeguato è fondamentale per garantire **scalabilità, manutenibilità e portabilità** del codice.
Due approcci particolarmente rilevanti sono:

- **Hardware Abstraction Layer (HAL)**, che fornisce un'interfaccia uniforme tra hardware e software applicativo, molto usata durante lo svolgimento di questa tesi nello sviluppo del firmware;
- **pattern architetturali a strati** (come l'*adapter pattern* o l'architettura esagonale), che separano la logica di business dalla gestione delle periferiche e dei servizi esterni. Usata principalmente dal team per lo sviluppo a livello applicativo.

Questi modelli hanno lo scopo comune di ridurre le dipendenze dal sistema ospitante e di rendere il software più robusto ai cambiamenti di piattaforma.

4.2 Hardware Abstraction Layer

L'adozione di modelli architetturali orientati alla separazione delle responsabilità è stata determinante per garantire portabilità e manutenibilità del firmware. Tra questi, **l'Hardware Abstraction Layer (HAL)** ha svolto un ruolo centrale, consentendo di disaccoppiare in modo sistematico la logica applicativa dai driver specifici di piattaforma (SDK). L'impiego dell'HAL fin dalle fasi iniziali di sviluppo ha reso possibile un percorso di migrazione ordinato tra piattaforme eterogenee, riducendo lo sforzo necessario al minimo a fronte di uno sforzo maggiore per lo sviluppo del firmware originario.

4.2.1 Cos'è realmente l'HAL

L'HAL è uno strato software intermedio che espone all'applicazione interfacce stabili (API) per l'accesso alle periferiche e ai servizi di base (ad esempio SPI, CAN, GPIO, timer, watchdog), nascondendo driver, il pin multiplexing, la configurazione dei clock e la gestione degli interrupt specifici di ciascun dispositivo.
In questo modo, il codice applicativo dipende da contratti funzionali chiari, anziché da implementazioni concrete.
Ne discende un duplice **beneficio**:

- da un lato si ottiene **portabilità**, perché l'implementazione dell'HAL può essere sostituita al variare della piattaforma;
- dall'altro si migliora la **manutenibilità**, perché la correzione di anomalie o l'ottimizzazione delle prestazioni resta confinata nello strato di astrazione, preservando l'integrità della logica superiore.

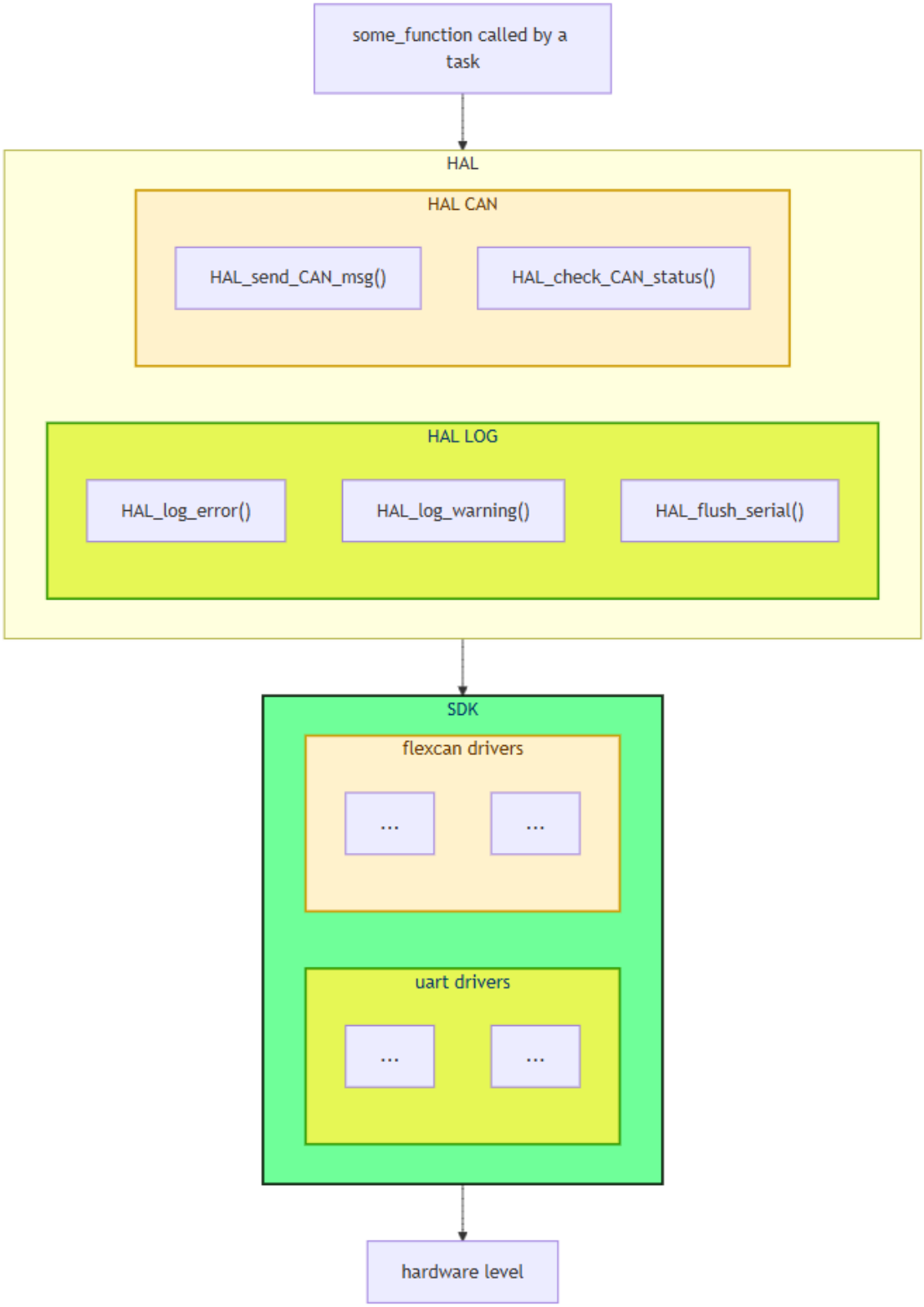


Figura 14 esempio di HAL

Nella Figura 14 è riportato un esempio di una possibile struttura di HAL. In questo schema, la funzione <*some_function*> — invocata da un task — utilizza gli HAL **CAN** e **LOG** per eseguire operazioni quali, ad esempio, l'invio di un messaggio sul bus CAN, la lettura del suo stato e la registrazione dell'evento sulla seriale, senza avere conoscenza diretta delle implementazioni sottostanti.

Se, nello stesso contesto, si volesse testare una diversa metodologia di logging, sarebbe sufficiente modificare l'HAL corrispondente, senza dover intervenire sul codice della funzione *<some_function>* o sul task che la utilizza.

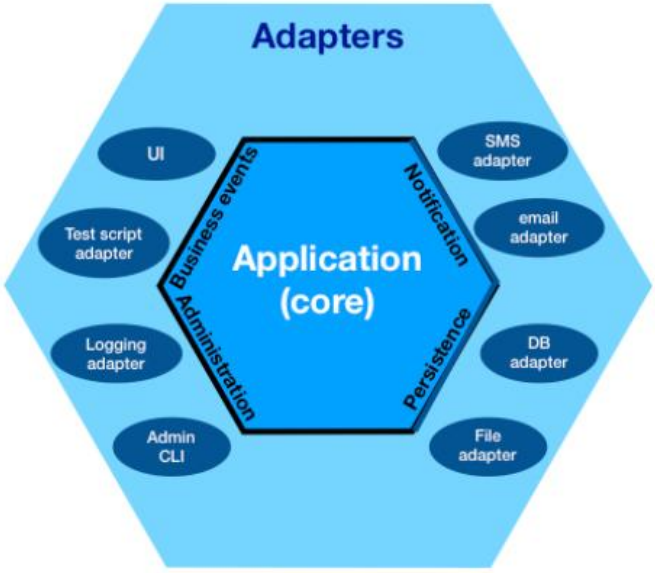
4.2.2 Progettazione modulare ed impatto sul porting

La scelta di strutturare fin da subito il firmware su i.MX8 intorno a interfacce HAL ha permesso di stabilire una chiara separazione tra la logica di alto livello (gestione dei messaggi, log, codifica) e i dettagli dell'SDK di piattaforma.
Il codice interagisce esclusivamente con API coerenti e documentate, evitando accessi diretti ai registri o dipendenza da nomi di funzione specifici dell'SDK sottostante.
Quando si è reso necessario migrare verso RT1064, il perimetro del cambiamento si è limitato, dove possibile, alla sostituzione dell'implementazione delle stesse interfacce: le chiamate applicative sono rimaste immutate, mentre sono stati riallineati i parametri di configurazione e i collegamenti ai driver del nuovo ambiente.

Questo ovviamente non è stato sempre possibile, ad esempio nel passaggio da un metodo di comunicazione (RPMsg) a un altro (SPI). Tuttavia, il principio resta valido: anche in questi casi l'HAL risulta utile per l'ottimizzazione e il test di differenti implementazioni, poiché è sufficiente modificare il codice sottostante al layer, senza intervenire sulla logica applicativa superiore.

4.3 Architettura esagonale

L'**Architettura Esagonale** (nota anche come *Ports and Adapters*) è un pattern architetturale proposto da Alistair Cockburn nel 2005 [14] con l'obiettivo di separare il **core logico** di un'applicazione da tutte le dipendenze esterne come interfacce utente, database, servizi esterni, protocolli di comunicazione, infrastruttura.



La metafora dell'esagono serve principalmente a rappresentare graficamente che il "cuore" dell'applicazione può avere più "facce" (alcune verso gli input, altre verso gli output) tramite cui interagire con il mondo esterno, mediante *ports* (interfacce astratte) e *adapters* (implementazioni concrete).

Componenti principali [14]**:**

- Il **core** (Dominio / Business Logic) è il centro dell'architettura. Qui risiedono gli use case, le regole di dominio, le entità (o strutture dati) centrali, insomma tutto ciò che rappresenta la logica di funzionamento indipendente dalla tecnologia. Il core **non dipende** da dettagli esterni: né driver hardware, né librerie, né protocolli di comunicazione.
- Le ***porte*** sono interfacce astratte esposte dal core, che definiscono i punti di ingresso e uscita per tutte le interazioni con l'esterno. Si distinguono principalmente due tipi:
    - **Inbound ports** (anche chiamati *primary* o *driving*): definiscono come il core può essere "guidato" dall'esterno (es. tramite un task, una richiesta, un interrupt, un messaggio).
    - **Outbound ports** (o *secondary / driven*): definiscono come il core si aspetta di comunicare verso l'esterno per servizi che non sono implementati nel core (es. persistenza, comunicazione esterna, logging, interfacce di rete)
- Gli ***adapters*** sono le implementazioni concrete delle porte: traducono richieste verso il core o comportamenti del core verso componenti esterni. Ci sono adapter inbound (ricevono input esterno e lo adattano al formato che il core si aspetta) e adapter outbound (implementano servizi esterni su interfacce fornite dal core).
- **Flusso delle dipendenze / principio di inversione**
  Un principio fondamentale è che le dipendenze devono andare **verso l'interno**, cioè il core non deve conoscere dettagli degli adapter o tecnologie concrete. Questo è strettamente legato al *Dependency Inversion Principle* (DIP). Le interfacce (porte) sono definite nel core, mentre gli adapter risiedono "all'esterno". In questo modo, è possibile sostituire un adapter (es. cambiare database, cambiare metodo di comunicazione) senza toccare il core.

4.4 FlexCAN

Il controller FlexCAN è un'interfaccia CAN integrata nei SoC NXP progettata per offrire flessibilità nella gestione del traffico CAN/CAN FD pur mantenendo le garanzie di affidabilità richieste in ambito automotive. FlexCAN espone meccanismi hardware per il buffering, il filtraggio e il time-stamping dei frame, consentendo al software di delegare al controller le operazioni più onerose e di concentrare la logica applicativa sul trattamento semantico dei messaggi.

4.4.1 Architettura funzionale essenziale

- **Message Buffers (MB):** unità di memoria mappate hardware utilizzate per la trasmissione e la ricezione; ogni MB può essere configurato singolarmente come trasmettitore o ricevitore con relativo filtro/maschera.

- **Rx FIFO**: modalità alternativa di ricezione che consente di accumulare frame in coda in modo efficiente per scenari con traffico burst.
- **Filtri e maschere** hardware: permettono al controller di accettare solo gli ID di interesse, riducendo il carico della CPU.
- **Registri di stato e interrupt**: segnali dedicati per eventi di MB completato, errori di bus, wake-up e condizioni di overflow.
- **Timestamping**: marcatura temporale dei frame in ingresso, utile per analisi temporali e diagnostica.
- **Supporto CAN FD** (nelle revisioni compatibili): gestione di DLC esteso e di bit-rate differenziati tra arbitraggio e fase dati.

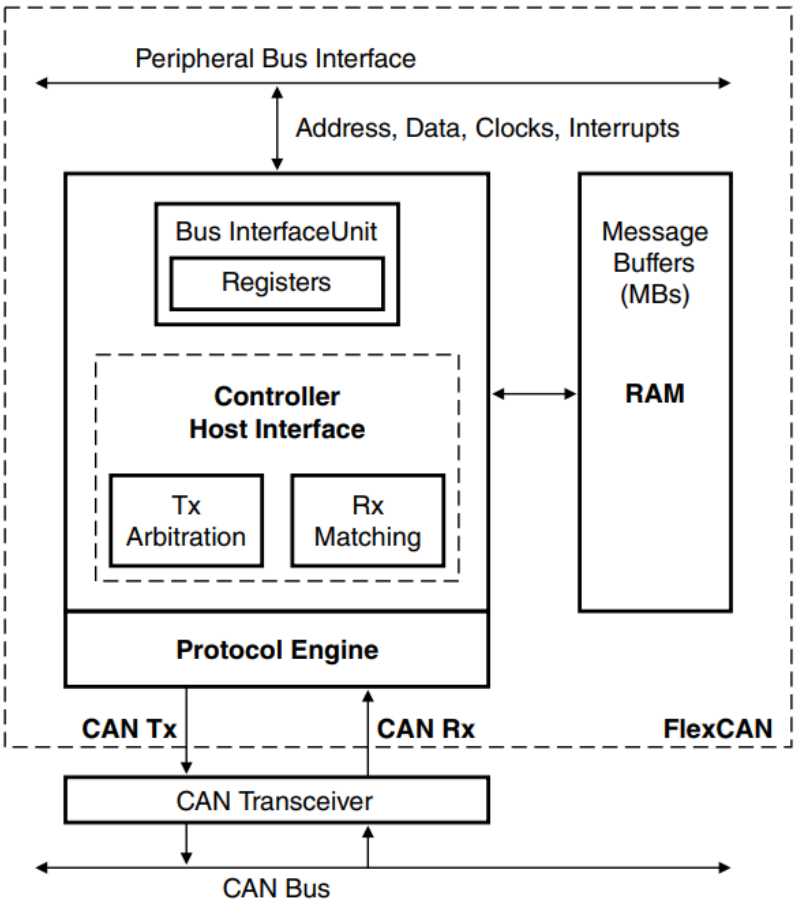


Figura 15 FlexCAN block diagram [27]

4.4.2 Pattern operativo adottato nel progetto

Nel firmware è stato realizzato un HAL specifico per FlexCAN che centralizza le operazioni di inizializzazione e gestione del controller senza esporre i dettagli dei registri al livello applicativo. Le attività principali svolte dal HAL sono: configurazione del bit timing e del pinmux, allocazione e configurazione dei Message Buffers (ruolo, filtri, DLC), abilitazione della Rx FIFO se necessaria e impostazione delle sorgenti di interrupt rilevanti.

Per minimizzare il tempo di esecuzione in interrupt e garantire determinismo, l'ISR gestita dal HAL svolge operazioni minime: riconosce l'evento sul MB o sulla FIFO, esegue una lettura rapida dello stato e delle dimensioni, e segnala l'avvenuto evento a un task dedicato tramite meccanismi RTOS (queue/semaphore). Il task di smaltimento, eseguito in contesto thread, provvede a leggere i frame completi dal driver HAL, applicare le operazioni di decodifica e parsing tramite un secondo HAL di livello superiore e inoltrare i payload alla pipeline applicativa per l'elaborazione.

4.4.3 Vantaggi di questo approccio

- ISR breve e prevedibile, riduzione del jitter nelle priorità real-time.
- Filtraggio precoce in hardware che riduce il carico CPU.
- Isolamento delle differenze di controller tra piattaforme: cambiando l'implementazione del HAL non cambiano le API esposte all'applicazione.
- Possibilità di ottimizzare buffering (MB vs FIFO), uso di DMA e politiche di retry senza alterare la logica di decoding.

Questa impostazione mantiene la complessità del controller confinata nello strato HAL e rende l'intera gestione CAN modulare, testabile e facilmente adattabile alle diverse revisioni hardware o ai requisiti di performance. I dettagli implementativi del driver FlexCAN e le scelte adottate saranno presentati nel capitolo 5 (Implementazione software).

Per motivi di riservatezza e in ottemperanza all'accordo di non divulgazione, i dettagli tecnici e le scelte implementative relative all'analisi e all'impiego dei Message Buffers del controller FlexCAN non sono riportati in questa sede.
Si può però affermare, senza entrare nel merito operativo, che una significativa porzione del lavoro sperimentale è stata dedicata allo studio delle politiche di buffering e allo studio delle modalità di ricezione/accodamento dei frame; tali attività hanno fornito gli elementi necessari per ottimizzare la latenza e la robustezza della catena di ricezione e decodifica.
I dettagli sperimentali, i risultati misurati e il codice sorgente relativo a queste attività non sono inclusi nella tesi per vincoli contrattuali, ma le conclusioni operative e gli effetti sul comportamento del sistema sono discussi sommariamente nei capitoli 7 e 8.

4.5 RTOS su MCU

Questa sezione confronta due approcci diffusi per l'esecuzione di software su microcontrollori: **FreeRTOS**, un kernel real-time leggero e largamente utilizzato in ambito embedded, e **AUTOSAR Classic**, uno standard industriale volto a fornire un framework completo per lo sviluppo di ECU compliant ai requisiti automotive. Lo scopo è chiarire punti di forza, limiti e motivazioni che hanno guidato la scelta di FreeRTOS per l'implementazione del firmware oggetto della tesi.

4.5.1 FreeRTOS

FreeRTOS (Free Real-Time Operating System) è un sistema operativo **real-time** open source progettato specificamente per microcontrollori e sistemi embedded. Nasce con l'obiettivo di fornire un ambiente leggero, efficiente e affidabile, capace di gestire l'esecuzione concorrente di più attività (task) in maniera controllata e prevedibile, caratteristiche imprescindibili nei sistemi a tempo reale. [15]

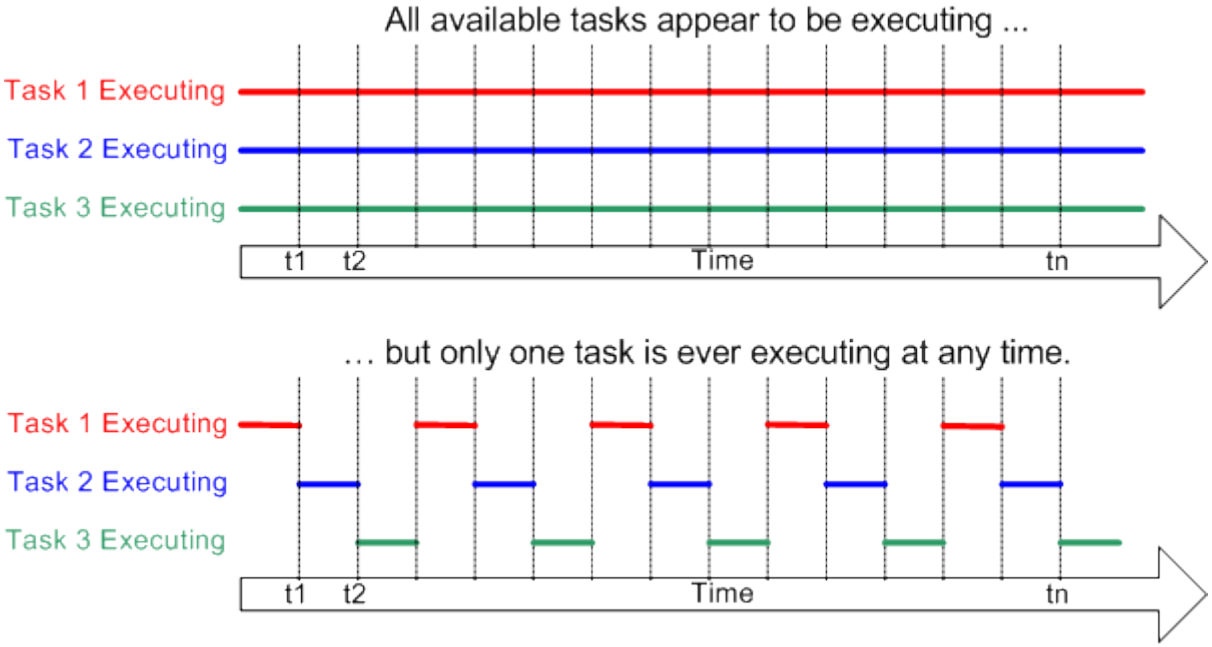


Figura 16 Multi tasking on single core applications [16]

A differenza dei sistemi operativi general-purpose, come Linux o Windows, FreeRTOS è pensato per dispositivi con risorse estremamente limitate: pochi kilobyte di memoria RAM, capacità di calcolo ridotta e assenza di unità di memoria virtuale. Nonostante la sua leggerezza, offre funzionalità tipiche dei sistemi multitasking, quali la gestione dei task, la pianificazione (scheduling), la comunicazione e la sincronizzazione tra processi, oltre alla gestione delle risorse condivise.

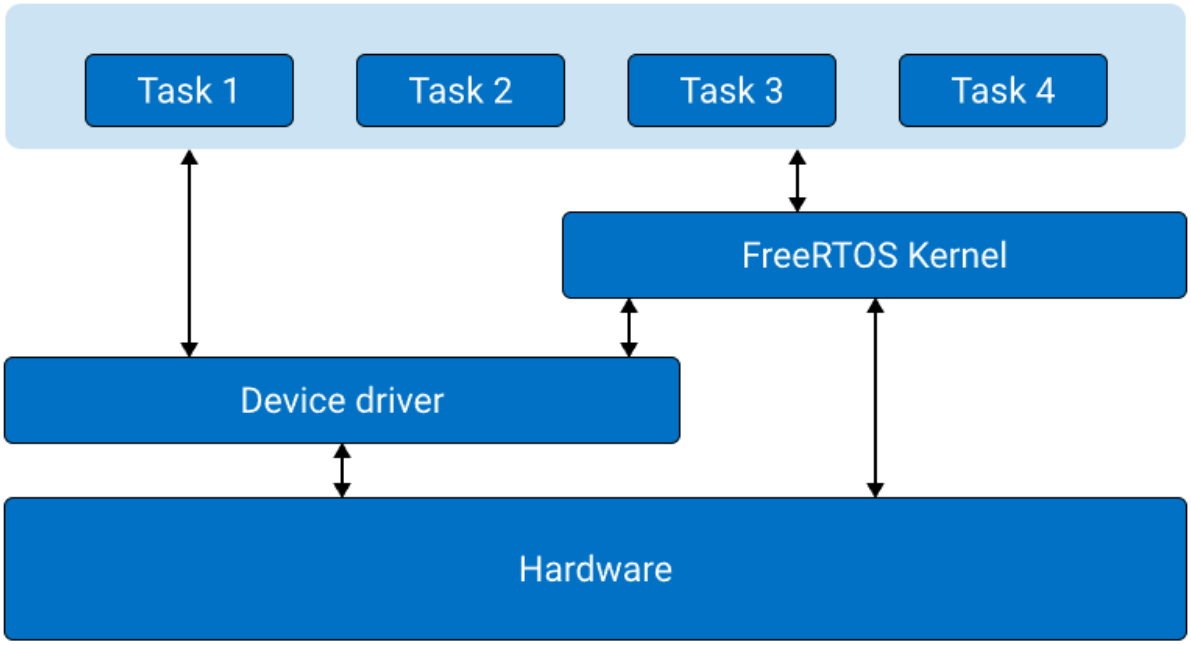


Figura 17 FreeRTOS architecture example

Il cuore di FreeRTOS è costituito dal **kernel**, responsabile della pianificazione dei task in base a politiche di priorità. Ogni task rappresenta un **thread** indipendente di esecuzione, che può essere sospeso, ripreso o terminato in funzione delle esigenze del sistema. Lo **scheduler** adotta un meccanismo di **priorità preemptive**: il task con priorità più alta viene eseguito immediatamente non appena diventa pronto, garantendo tempi di risposta deterministici, requisito fondamentale nei sistemi real-time. [17]

FreeRTOS mette inoltre a disposizione diversi strumenti per la sincronizzazione e la comunicazione tra task, come code di messaggi, semafori, mutex ed event group. Questi meccanismi consentono di coordinare le attività concorrenti ed evitare condizioni di competizione (race conditions), favorendo la progettazione di applicazioni modulari e sicure dal punto di vista della concorrenza.

Un ulteriore aspetto di rilievo è la **portabilità**. FreeRTOS è stato adattato a centinaia di architetture di microcontrollori e processori, tra cui ARM Cortex-M, RISC-V, AVR e molte altre. Il kernel è scritto in C standard e viene accompagnato da un piccolo strato di porting che ne consente l'integrazione con l'hardware specifico. Questa caratteristica lo rende una scelta diffusa in ambito industriale, automotive, aerospaziale e IoT.
Infine, la licenza open source (MIT) e il supporto di una vasta community — insieme al sostegno commerciale offerto da Amazon Web Services (AWS) [15] con Amazon FreeRTOS — hanno contribuito a consolidare FreeRTOS come uno degli **RTOS più utilizzati al mondo.**

4.5.2 Perché scegliere FreeRTOS

La combinazione di **semplicità**, **affidabilità** e ampia **compatibilità** lo rende uno strumento **ideale** per lo sviluppo di sistemi embedded complessi ma efficienti, in cui il controllo preciso del tempo e delle risorse è cruciale.

Nel contesto di questa tesi, FreeRTOS è stato scelto come sistema operativo per la MCU in virtù della sua semplicità di utilizzo, della disponibilità di numerosi esempi integrati negli SDK dei produttori e, non ultimo, della **gratuità**. Questi fattori hanno permesso di ridurre i tempi di sviluppo e di concentrare gli sforzi sull'architettura del firmware e sull'integrazione con l'HAL, senza introdurre la complessità di framework più strutturati.

4.5.3 AUTOSAR Classic

AUTOSAR Classic (AUTomotive Open System ARchitecture) è uno standard internazionale nato dalla collaborazione tra costruttori automobilistici, fornitori e aziende del settore elettronico. Il suo obiettivo principale è fornire un'architettura software comune per le centraline elettroniche (ECU), capace di affrontare la crescente complessità dei sistemi embedded automotive e di garantire interoperabilità, riusabilità e scalabilità del software.

4.5.3.1 Architettura a strati

Una delle caratteristiche distintive di AUTOSAR Classic è la sua architettura stratificata:

- **Microcontroller Abstraction Layer (MCAL):** driver standardizzati per le periferiche del microcontrollore.
- **Basic Software (BSW):** moduli di servizio comuni (comunicazione, diagnostica, gestione memoria).
- **Runtime Environment (RTE):** middleware che collega i componenti applicativi con i servizi sottostanti.
- **Software Components (SWC):** moduli applicativi che implementano le funzioni specifiche del veicolo.

Questa organizzazione consente di separare nettamente la logica applicativa dai dettagli hardware, favorendo portabilità e collaborazione tra fornitori diversi

4.5.3.2 Vantaggi e limiti

Vantaggi:

- **Standardizzazione:** interfacce comuni che semplificano l'integrazione multi-fornitore.
- **Supporto ai requisiti automotive:** diagnostica, sicurezza funzionale (ISO 26262), configurabilità.

- **Determinismo:** comportamento prevedibile e adatto a sistemi real-time critici.

Limiti:

- **Curva di apprendimento ripida:** richiede conoscenze specifiche e tool dedicati.
- **Costi e tempi di sviluppo:** licenze commerciali e processi di configurazione articolati.
- **Overhead:** footprint maggiore, meno adatto a MCU con risorse molto limitate.

4.5.4 AUTOSAR e FreeRTOS a confronto

FreeRTOS e AUTOSAR Classic rappresentano due approcci profondamente diversi alla gestione del software su microcontrollori.

- **FreeRTOS** è un kernel real-time open source, leggero e facilmente integrabile, pensato per dispositivi con risorse limitate. È ideale in contesti di prototipazione, ricerca o applicazioni embedded che richiedono determinismo ma non necessitano di processi di certificazione complessi.
- **AUTOSAR Classic**, al contrario, è uno standard industriale che definisce un'architettura software completa. È stato concepito per il settore automotive, dove interoperabilità e conformità normativa sono vincoli imprescindibili. Garantisce elevato determinismo e supporta funzionalità avanzate ma richiede tool dedicati, competenze specifiche e introduce costi e complessità maggiori.

In sintesi, **FreeRTOS privilegia la rapidità di sviluppo, la leggerezza e la gratuità**, risultando adatto a progetti sperimentali e a sistemi embedded a risorse contenute. **AUTOSAR Classic privilegia la standardizzazione e la robustezza industriale**, risultando la scelta naturale per ECU destinate alla produzione di serie e soggette a requisiti di safety e compliance.

4.6 QNX ed AUTOSAR Adaptive

QNX fornisce il sistema operativo real-time affidabile e certificabile, mentre **AUTOSAR Adaptive fornisce lo standard architetturale e i servizi di alto livello** per costruire applicazioni complesse e scalabili su CPU moderne. La loro relazione è quindi complementare: QNX rappresenta la base esecutiva, Adaptive l'ambiente standardizzato che governa lo sviluppo e l'integrazione del software in veicoli di nuova generazione.

4.6.1 QNX

QNX è un sistema operativo **real-time Unix-like, POSIX-compliant e basato su microkernel**, sviluppato originariamente negli anni '80 e oggi mantenuto da BlackBerry. Come riportato nella documentazione ufficiale, è stato concepito per

fornire una piattaforma affidabile e sicura per sistemi embedded critici, ed è oggi utilizzato in settori che spaziano dall'automotive alla robotica, dai dispositivi medicali all'automazione industriale [18].
La caratteristica distintiva di QNX è la sua **architettura a microkernel**: il kernel include soltanto le funzioni essenziali (scheduling, gestione degli interrupt, comunicazione inter-processo e timer), mentre tutti gli altri servizi — driver, file system, stack di rete — vengono eseguiti come processi in spazio utente. Come sottolinea Wikipedia, "l'idea base è quella di costruire il sistema operativo come un insieme di piccole parti che offrono uno specifico servizio" [19]. Questo approccio consente di isolare i guasti: un errore in un driver non compromette l'intero sistema, ma solo il servizio interessato, che può essere riavviato senza impatti globali.
Dal punto di vista delle prestazioni, QNX implementa meccanismi di **comunicazione inter-processo (IPC)** estremamente efficienti, basati sul passaggio di messaggi sincroni. Questo modello, denominato *message passing*, permette di attivare immediatamente il processo destinatario senza passare dallo scheduler, riducendo la latenza e garantendo tempi di risposta deterministici.
Il sistema è inoltre **scalabile e modulare**: l'utente può assemblare un'immagine del sistema operativo includendo solo i componenti necessari, senza dover ricompilare il kernel. Questa flessibilità lo rende adatto a contesti molto diversi, dai sistemi embedded a risorse ridotte fino alle piattaforme multi-core ad alte prestazioni.
Secondo il sito ufficiale, QNX è oggi adottato in oltre **255 milioni di veicoli** e rappresenta una delle basi software più diffuse per lo sviluppo di sistemi automotive sicuri e certificabili [18]. Le sue varianti pre-certificate accelerano i processi di conformità a standard come **ISO 26262** (automotive) rendendolo una scelta privilegiata per applicazioni safety-critical.
In sintesi, QNX si distingue per:

- **Affidabilità e sicurezza**, grazie all'architettura microkernel e alla tolleranza ai guasti.
- **Determinismo real-time**, garantito da uno scheduler a priorità e da un IPC ottimizzato.
- **Scalabilità**, che consente di adattarlo a diversi domini applicativi.
- **Certificabilità**, con supporto a standard di sicurezza funzionale.

Queste caratteristiche ne hanno fatto uno dei sistemi operativi embedded più longevi e consolidati, nonché una piattaforma di riferimento per l'industria automotive e per i sistemi embedded critici.

4.6.2 AUTOSAR Adaptive

AUTOSAR Adaptive rappresenta l'evoluzione dello standard AUTOSAR, pensata per le **High Performance Computing Units (HPC)** e per le ECU di nuova generazione. A differenza di AUTOSAR Classic, che si concentra su microcontrollori e sistemi deterministici, Adaptive è progettato per CPU multi-core e piattaforme ad alte prestazioni, spesso basate su Linux o QNX come sistema operativo sottostante.

Le sue caratteristiche principali sono:

- **Architettura orientata ai servizi (Service-Oriented Architecture, SOA):** i componenti software comunicano tramite servizi dinamici, utilizzando protocolli come SOME/IP o DDS.
- **Supporto POSIX:** garantisce portabilità e compatibilità con librerie e strumenti standard.
- **Flessibilità:** consente aggiornamenti dinamici del software (over-the-air), requisito fondamentale per veicoli connessi e funzioni ADAS/AD.
- **Focus su applicazioni complesse:** è pensato per gestire algoritmi di percezione, fusione sensoriale, intelligenza artificiale e comunicazioni V2X.

AUTOSAR Adaptive non sostituisce QNX o Linux, ma si appoggia a questi sistemi operativi come base, fornendo un framework standardizzato per lo sviluppo e l'integrazione di applicazioni automotive di nuova generazione.

5 Implementazione software

Questo capitolo presenta l'implementazione software del sistema su rete CAN, a partire dagli strumenti di sviluppo adottati: le toolchain per piattaforme ARM e QNX, l'ambiente **MCUXpresso IDE** con il relativo SDK per i.MX RT1064 e l'utilizzo di **Visual Studio Code** come supporto per una gestione modulare del progetto. Viene quindi descritta l'architettura a Hardware Abstraction Layer (HAL), concepita per isolare i driver di periferica e fornire API stabili per UART, log, CAN, SPI, watchdog e timer generici, insieme ai principi di gestione delle interruzioni hardware (IRQ) e dei rispettivi handler, in un'ottica di preemption e determinismo temporale. La parte conclusiva introduce i criteri di **codifica e decodifica dei messaggi CAN**, l'impiego del file **DBC** come contratto di comunicazione tra ECU e firmware e l'utilizzo, per il codec, di una versione modificata della libreria open-source ***CoderDBC***.

5.1 Requisiti del sistema

In ingegneria i requisiti rappresentano il contratto tra esigenze iniziali e realizzazione finale. La loro definizione chiara abilita tracciabilità, verificabilità e coerenza lungo l'intero ciclo di sviluppo, riducendo ambiguità e interpretazioni divergenti.

- **Requisiti funzionali**: descrivono *cosa* il sistema deve fare (funzionalità, servizi, comportamenti).
- **Requisiti non funzionali**: definiscono *come* il sistema deve essere (prestazioni, affidabilità, sicurezza, portabilità, vincoli di risorse).

Ogni requisito è identificato da un ID univoco, ha un criterio di accettazione oggettivo e viene tracciato verso i casi di test (cap. 8) e le misure prestazionali (cap. 7).

5.1.1 I requisiti del progetto

La base funzionale resta coerente con la versione originaria (comunicazione CAN, RX/TX, filtraggio ID, diagnostica essenziale).
La novità architetturale è l'introduzione di un microcontrollore dedicato NXP i.MX RT1064 che gestisce la comunicazione CAN e le logiche real-time, con Raspberry Pi 4 come host applicativo. La comunicazione **MCU↔Host** avviene via **SPI**.
Questa scelta impone requisiti aggiuntivi su:

- **interfaccia MCU↔Host** (formato messaggi, sincronizzazione);
- **separazione delle responsabilità** (tempo reale su MCU, funzioni applicative/logging su host);
- **determinismo** e **affidabilità** della gestione CAN indipendentemente dal carico del Raspberry Pi 4;
- **prestazioni** uguali o migliori rispetto alla versione precedente su i.MX8.

5.1.2 Esempio di requisiti

Di seguito viene riportata una tabella esemplificativa di una serie di requisiti e criteri di accettazione (dati pseudo reali):

Tabella 1 Esempio di requisiti software

ID	Tipo	Descrizione	Criterio di accettazione
FR-01	FUNZ	Ricezione e trasmissione di frame **CAN**	RX/TX affidabile @ 500 kbps con 0 perdite su 5000 frame ad un bus load del 60%
FR-02	FUNZ	**Filtraggio** per ID e maschere	Filtri per ID configurabili; solo gli ID attesi vengono ricevuti
FR-03	FUNZ	Self-recovery in caso di guasti critici.	Recupero automatico di tutte le funzionalità nel caso di un evento critico.
FR-04	FUNZ	**Configurazione runtime** parametri CAN	Filtri aggiornabili senza reboot; applicazione entro 1 s
NFR-01	PRE	**Latenza E2E** MCU→Host (RX CAN → consegna su Raspberry)	≤ 10.0 ms con carico CAN BUS ~60%
NFR-02	PRE	**Throughput sostenibile**	Fino al 100% di utilizzo bus a 500 kbps senza overflow di code/buffer
NFR-03	AFF	**Recovery bus-off**	Rientro in stato operativo in ≤ 150 ms
NFR-04	RIS	**Footprint memoria MCU**	Picco RAM ≤ 64 KiB, Flash ≤ 256 KiB
NFR-05	COM	**Affidabilità link SPI** MCU↔Host	BER≈0 su 1 h a 10 MHz, MTU 256 B; retry < 1‰ pacchetti
NFR-06	QUAL	**Coding standard**	Violazioni MISRA-C:2012
NFR-07	PORT	**Portabilità** su board target	Build e test OK su RT1064; eventuale documentazione (linee guida incluse)
NFR-08	OBS	**Osservabilità** (logging/metrics)	Log strutturati con timestamp; esportazione contatori errori; metriche prestazionali

Legenda Tipo: FUNZ = funzionale; PRE = prestazioni; AFF = affidabilità; RIS = risorse; COM = comunicazione; QUAL = qualità; PORT = portabilità; OBS = osservabilità.

5.2 Ambienti di sviluppo e Software Development Kit

Lo sviluppo del firmware per sistemi embedded richiede strumenti integrati che semplifichino la configurazione dell'hardware, la scrittura del codice e il debug. In questo contesto, NXP mette a disposizione la suite **MCUXpresso**, un ecosistema di tool e librerie progettato per accelerare lo sviluppo su microcontrollori **Arm® Cortex-M**. Tale ecosistema comprende IDE, SDK, strumenti di configurazione e debug, oltre a un'ampia gamma di esempi e middleware. In particolare, sono stati utilizzati MCUXpresso IDE, specificamente per il microcontrollore i.MX RT1064, e **Visual Studio Code**, adottato come ambiente comune e trasversale per tutte le piattaforme coinvolte (i.MX8, i.MX RT1064, Raspberry Pi, Arduino, ecc.).

5.2.1 IDE

Un **IDE (Integrated Development Environment)** è un'applicazione che combina editor del codice, strumenti di build/compilazione e debugger in un'unica interfaccia grafica per aumentare produttività e qualità del software. Pur non essendo indispensabile (si potrebbe usare solo editor + riga di comando), l'integrazione riduce tempi di setup e facilita debug e gestione del progetto. [20]

5.2.2 SDK

Un **SDK (Software Development Kit)** è un insieme di driver, HAL, librerie/middleware, esempi, script di build e documentazione pensati per una specifica famiglia di MCU/board, così da accelerare lo sviluppo e ridurre il rischio di errori nell'integrazione con l'hardware. L'MCUXpresso SDK di NXP, ad esempio, include driver periferici, numerosi esempi, integrazioni opzionali con RTOS (come FreeRTOS) e middleware utili per la prototipazione rapida. [21]

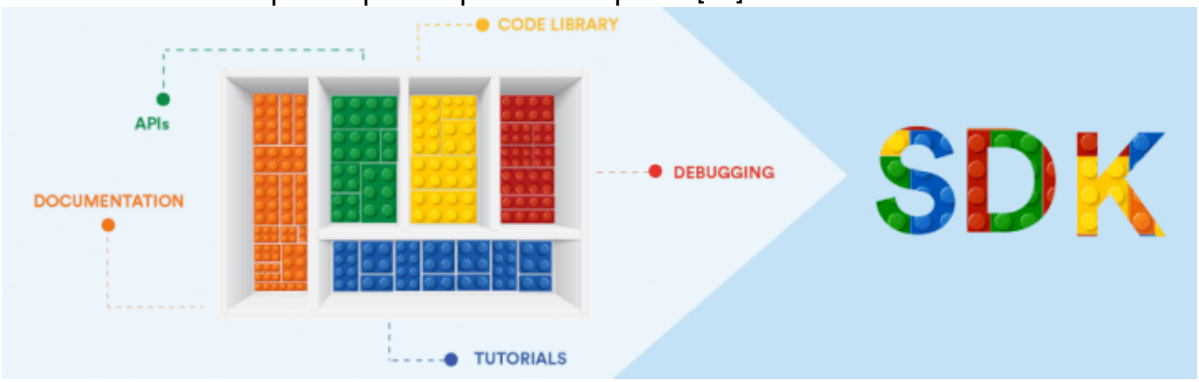
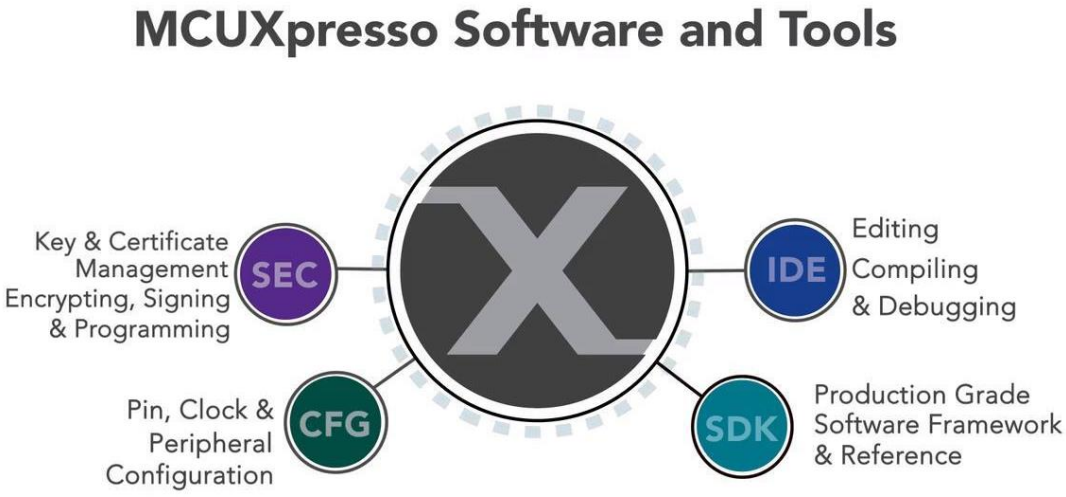


Figura 18 Rappresentazione grafica di ciò che costituisce un SDK (fonte: clevertap.com)

5.2.3 MCUXpresso strumenti e software



5.2.3.1 MCUXpresso IDE

MCUXpresso IDE è l'ambiente di sviluppo integrato di NXP per MCU Arm® Cortex-M. È basato su Eclipse e integra la toolchain **GNU Compiler Collection (GCC) Arm Embedded**, offrendo un'interfaccia unica per editing, build e debug. Tra le funzioni principali:

- **Integrazione nativa con MCUXpresso SDK**: selezione/aggiunta dei pacchetti dall'SDK Builder e import degli esempi direttamente dall'IDE.
- **Debug avanzato**: supporto per MCU-Link, SEGGER J-Link, con SWO trace e profiling, instruction trace (ETB/MTB), viste per registri periferici e supporto a FreeRTOS.
- **MCUXpresso Config Tools** integrati: configurazione grafica ed immediata di **pin**, **clock** e **periferiche** con generazione automatica del codice.
- **New Project Wizard**: creazione rapida del progetto selezionando board, toolchain e componenti SDK.

Nel progetto di questa tesi, MCUXpresso IDE è stato usato esclusivamente per lo sviluppo sul microcontrollore NXP i.MX RT1064, essendo l'ambiente di riferimento ufficiale per questa piattaforma. [21]

5.2.3.2 MCUXpresso SDK

Si tratta di un pacchetto software modulare che fornisce tutto il necessario per sviluppare su una specifica MCU/board NXP. Per la **EVK-MIMXRT1064** mette a disposizione:

- **BSP e driver di periferica**: implementazioni in C per UART, SPI, I²C, **CAN/FlexCAN**, GPIO, timer, watchdog, con pinmux e clock preconfigurati per la scheda.
- **HAL e struttura a layer**: separazione chiara tra accesso all'hardware (driver/HAL), middleware e applicazioni.
- **Middleware e RTOS opzionali**: integrazione con **FreeRTOS** e stack (es. USB,

lwIP, codec audio) selezionabili secondo necessità.
- **Esempi applicativi**: progetti dimostrativi (es. *can_loopback*, *uart_interrupt*, *spi_transfer*) con istruzioni di build ed esecuzione.

Nel contesto di questa tesi, l'SDK ha rappresentato il fondamento dell'intero sviluppo firmware. La disponibilità di esempi ufficiali ha permesso di passare rapidamente dall'idea al prototipo: funzionalità chiave come la gestione CAN sono state verificate in tempi brevi, sfruttando percorsi di inizializzazione e sequenze di test già collaudati. Affidarsi ai driver forniti da NXP ha inoltre ridotto sensibilmente il rischio di errori nei livelli più vicini all'hardware, assicurando un comportamento aderente alle specifiche del silicio e liberando tempo per concentrarsi sugli aspetti applicativi.

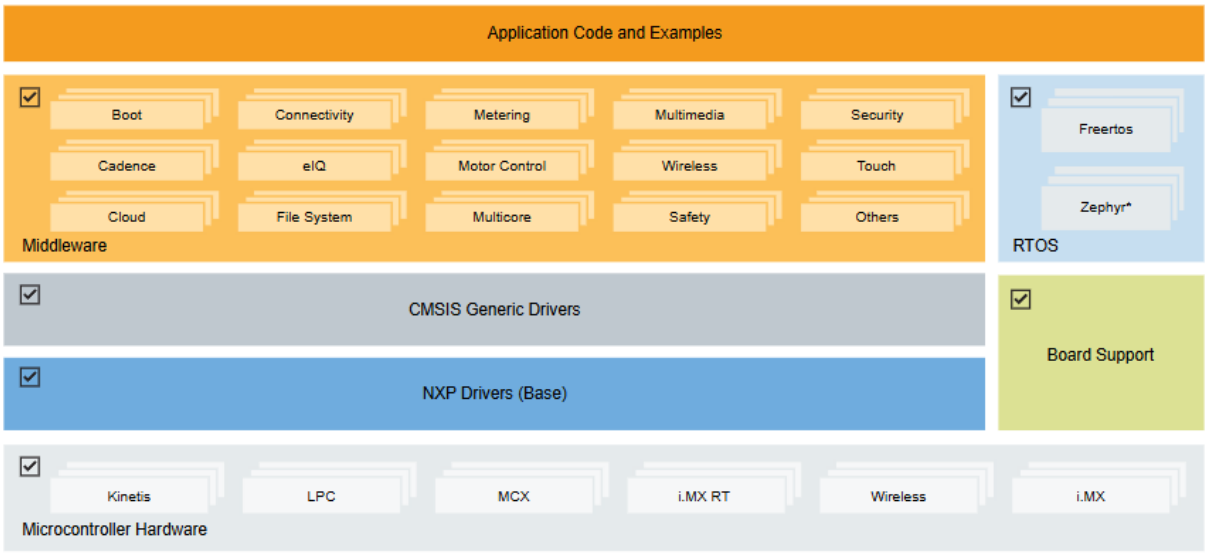


Figura 19 MCUXpresso SDK High-Level Block Diagram [22]

Al tempo stesso, la struttura a strati dell'SDK ha favorito un'architettura centrata sull'HAL: le dipendenze dall'hardware sono rimaste confinate nei driver e nei moduli di astrazione, mentre la logica applicativa ha potuto evolvere in modo più pulito, portabile e tracciabile. Questo approccio ha reso più semplice non solo la manutenzione, ma anche l'eventuale migrazione del codice verso altre piattaforme NXP, in linea con i principi di riusabilità e modularità.

5.2.3.3 I vantaggi dell'ambiente NXP

In sintesi, l'MCUXpresso SDK ha costituito la base software del firmware su RT1064, mentre l'MCUXpresso IDE ha fornito l'ambiente operativo per configurare, compilare e fare debug. Questa combinazione ha reso il flusso di lavoro più efficiente e ha consentito di concentrare gli sforzi sulle parti a maggior valore aggiunto: la logica applicativa (gestione CAN, protocolli, interfaccia SPI verso l'host Raspberry Pi 4) e la definizione dell'architettura complessiva del sistema. In questo modo, la scelta metodologica si è allineata agli obiettivi generali della tesi: realizzare un'architettura modulare, robusta e facilmente estendibile.

5.3 Cross-compilazione

Lo sviluppo di sistemi embedded richiede spesso la **cross-compilazione**, ovvero la generazione di eseguibili per una piattaforma target diversa da quella host su cui gira il compilatore. In questo scenario, il computer host (tipicamente una workstation Linux o Windows) esegue il compilatore, mentre il codice prodotto è destinato a un dispositivo embedded con architettura differente (ad esempio ARM). [23]

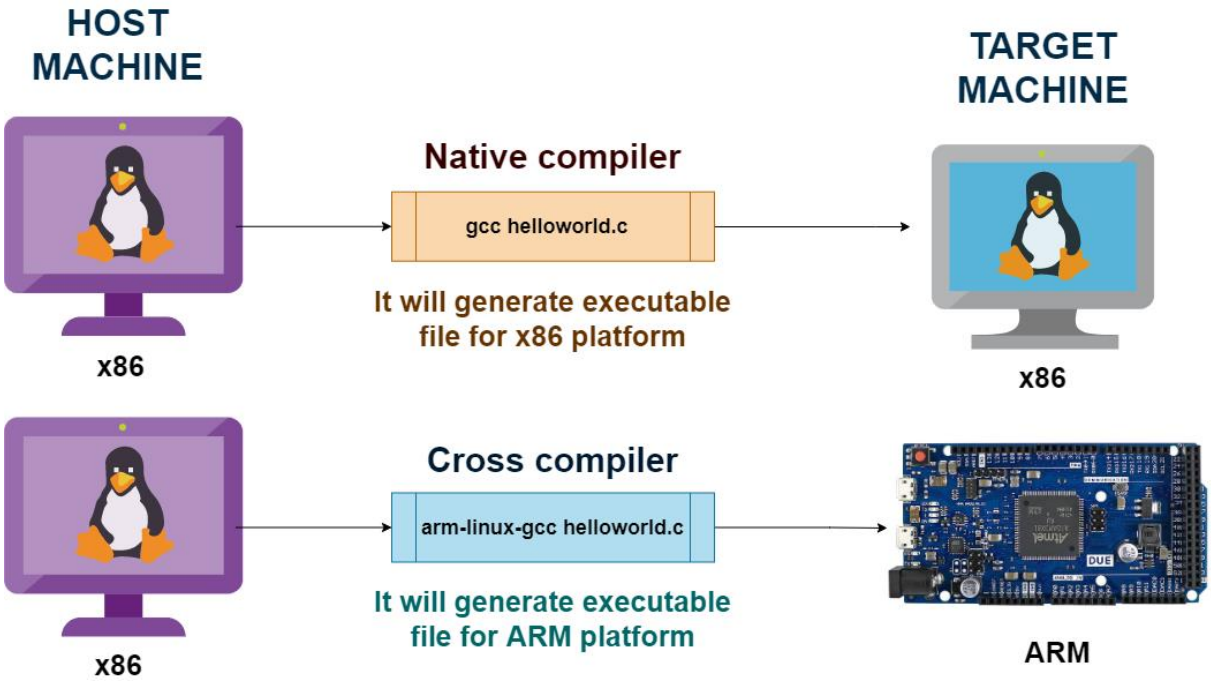


Figura 20 Rappresentazione grafica di compilazione nativa e cross-compilazione [23]

5.3.1 GCC Arm Embedded (arm-none-eabi)

Per il firmware bare-metal/FreeRTOS su NXP i.MX RT1064 (Cortex-M7) e i.MX8qm (Cortex-M4) è stata utilizzata la Arm GNU Toolchain nella variante arm-none-eabi. Si tratta della distribuzione ufficiale open-source di GCC/Binutils/GDB per Arm, con newlib/newlib-nano, pensata per generare codice senza dipendenze da Linux/glibc.
L'uso di GCC ha garantito portabilità, stabilità e la possibilità di sfruttare esempi e template ufficiali forniti da NXP.

5.3.2 QCC / Q++ per QNX

Per i moduli software destinati a girare su **QNX Neutrino RTOS**, è stato invece utilizzato il compilatore **QCC**, fornito all'interno del QNX Software Development Platform (SDP). QCC è un driver di compilazione che integra GCC e Clang con le librerie e i tool specifici di QNX, semplificando la generazione di eseguibili compatibili con il microkernel Neutrino.

5.4 Interrupt: principi, scelte progettuali e uso nel firmware

Gli **interrupt** sono il meccanismo con cui l'hardware segnala eventi asincroni alla CPU (arrivo di un frame CAN, byte su UART/SPI, scadenza di un timer, watchdog imminente). Su MCU **Cortex-M** come l'RT1064, la gestione è affidata al **NVIC** (Nested Vectored Interrupt Controller), che assegna priorità, abilita/disabilita singole sorgenti e supporta annidamento e *tail-chaining* (passaggio rapido tra ISR consecutive).

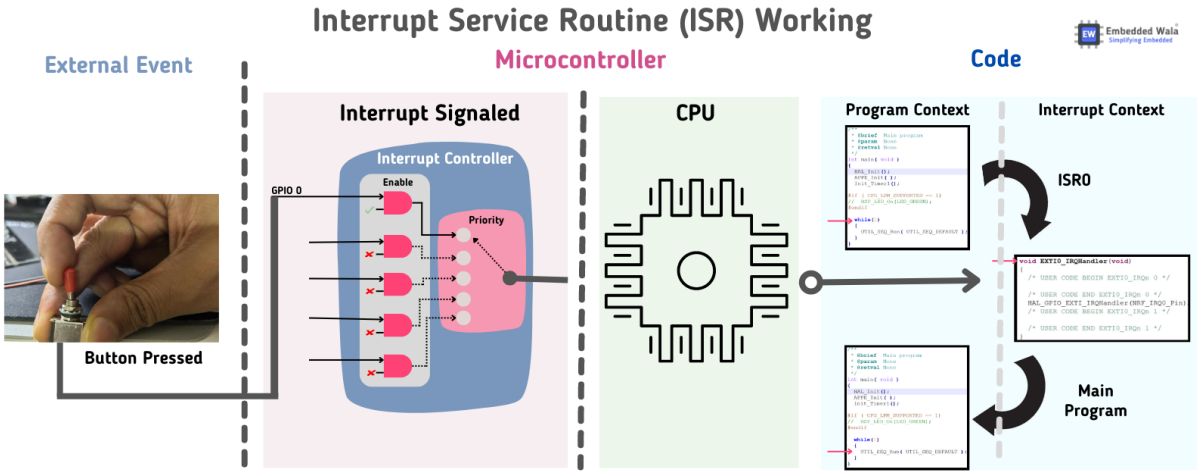


Figura 21 Rappresentazione grafica del funzionamento di un ISR [24]

5.4.1 Concetti chiave

- **ISR (Interrupt Service Routine)**: routine che gestisce l'evento. Deve essere **breve e deterministica**; idealmente si limita a leggere/scrivere qualche dato e/o eseguire operazioni basilari.
- **Priorità e annidamento**: le IRQ con priorità più alta possono interrompere quelle più basse. Una mappa di priorità coerente evita ritardi su eventi *time-critical* (es. CAN RX, SPI).
- **Mascheramento e latenza**: l'uso di *critical section* o la disabilitazione globale delle IRQ va ridotto al minimo; ogni periodo di mascheramento aumenta la latenza percepita da tutte le periferiche può far perdere eventi.
- **Sincronizzazione**: condividendo dati tra ISR e task, vanno usati **buffer lock-free**, variabili volatile, barriere di memoria o primitive RTOS "FromISR" (per code/semafori).
- **Determinismo**: si ottiene tenendo le ISR corte, evitando allocazioni dinamiche, preferendo **DMA** quando disponibile e dimensionando con cura i buffer.

5.4.2 Sorgenti d'interrupt nel progetto

- **CAN**: IRQ di RX/TX/error; usate per drenare i mailbox/buffer hardware e catturare condizioni *error passive* o *bus-off*.

- **SPI**: IRQ di fine trasferimento/errore (o DMA completion) nell'interfaccia verso l'host.
- **UART**: IRQ di RX/TX per console/diagnostica.
- **General Purpose Timers (GPT)**: generazione di **tick** periodici e timestamp.
- **Watchdog**: IRQ/finestra di sorveglianza per azioni di salvataggio minimo prima del reset.

5.4.3 Politica adottata con gli Interrupt

- **ISR piccole, task grandi**: le ISR eseguono solo l'essenziale (lettura/scrittura buffer, aggiornamento contatori, *controllo errori*) e risvegliano uno o più **task** dedicati per l'elaborazione delle informazioni.
- **Codeline "FromISR"**: ogni segnalazione al sistema (es. invio su coda FreeRTOS) usa le API specifiche *FromISR* per evitare blocchi e gestire correttamente la possibile *higher priority task woken*.
- **Priorità NVIC**: assegnate in ordine di criticità temporale;
    1. **CAN RX/TX** (massima)
    2. **SPI/DMA**
    3. **Timer** (timebase)
    4. **UART** (console)
    5. **Watchdog** (funzionamento indipendent dall'interrupt)

    Questa gerarchia limita la jitter sui percorsi dati principali (CAN↔SPI).
- **DMA dove possibile**: SPI (e, se possibile, CAN) usano DMA per ridurre la permanenza in ISR e il carico CPU, vedi Capitolo 7 per ulteriori dettagli.
- **Bufferizzazione**: mai trattare i dati direttamente nelle ISR, inserirli sempre in dei buffer e/o code e farli smaltire separatamente.
- **Misure di latenza**: timestamp in ingresso/uscita ISR (via GPT) per stimare latenza e tempo di servizio; metriche esposte al sistema di log vedi Capitolo 7.

Questa strategia sugli interrupt è alla base dell'organizzazione **HAL**: ciascun modulo periferico gestisce la propria ISR, incapsula i dettagli hardware e offre **API non bloccanti** all'applicativo. Il risultato è un comportamento più **prevedibile** sotto carico, un minor accoppiamento con l'hardware e una migliore **portabilità** del firmware.

5.5 Implementazione degli HAL

Come anticipato nel Capitolo 4.2, il firmware adotta un **Hardware Abstraction Layer (HAL)** per isolare la logica applicativa dai dettagli della piattaforma. In questa sezione vengono presentati i moduli effettivamente sviluppati e la loro integrazione con il sistema.
Ogni modulo espone **API stabili**; incapsula driver, registri e configurazioni dell'SDK. Il disaccoppiamento facilita portabilità, test, modifiche e manutenzione.

I moduli implementati sono:

- **UART**: comunicazione seriale.
- **CAN**: trasmissione/ricezione frame.
- **SPI**: interfaccia verso host.
- **Watchdog timer**: supervisione e reset in caso di fault.
- **General Purpose Timer (GPT)**: generazione di eventi periodici.
- **Codec**: codifica/decodifica dei messaggi CAN.
- **Log**: gestione dei log

5.5.1 HAL UART

Scopo: Configurazione ed uso della seriale ai fini di log/telemetria.

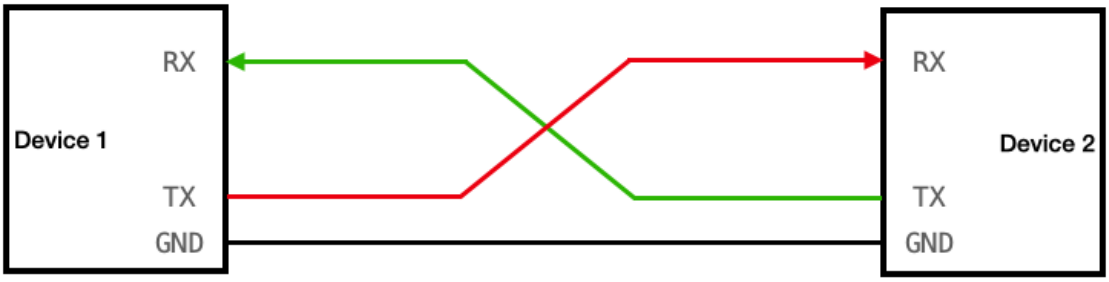


Figura 22 Esempio di connessione UART [25]

Funzioni principali:

- Configurazione periferica (baud rate, bit di dati, parità, stop).
- **RX/TX non bloccante** con **buffer ping-pong** per la ricezione.
- Gestione **ISR leggere**: drenano il dato, aggiornano contatori, notificano i task con API *FromISR*.
- Integrazione con livello **Log** per l'invio dei record.

Flusso operativo:

a. Init clock e pin (Config Tools), creazione handle driver.
b. Allocazione del buffer Ping/Pong.
c. Riceve una richiesta di invio contenente una stringa.
d. Appende la stringa al buffer libero, se si riempie allora avvia una trasmissione non bloccante sulla UART.
e. ISR: alla fine di un invio commuta invia una notifica al task. Inoltre, valuta l'eventuale presenza di errori e li segnala con opportuni flag.
f. Il task monitora lo stato dei buffer, scambiando buffer A e B in base a quale è usato in quel momento per scrivere sulla seriale o per essere usato per appendere. Inoltre, anche se il buffer non è stato riempito, ma è trascorso un tempo di time-out allora invia ugualmente il contenuto del buffer (se presente).

Note di progetto: La soglia di trasmissione (in byte) e di time-out sono state accuratamente scelte per minimizzare il carico sulla CPU ma garantire comunque una bassa latenza sui log.

5.5.2 HAL CAN

Scopo. Gestione completa del **FlexCAN**: configurazione, **mailbox/message buffers (MB)**, filtri, RX/TX frame ed errori.

Funzioni principali:

- Configurazione parametri CAN bus in concordanza con le specifiche IVECO.
- Assegnazione mailbox TX/RX.
- **ISR**: verificano gli stati delle mailbox, marcano timestamp, aggiornano contatori (overflow, errori), segnalano ad i task di "smaltimento" il numero di mailbox da leggere e/o la presenza di errori.
- **Task di elaborazione**: validazione, decodifica via HAL codec, instradamento verso Host e/o Log.

Flusso operativo (esempio generico):

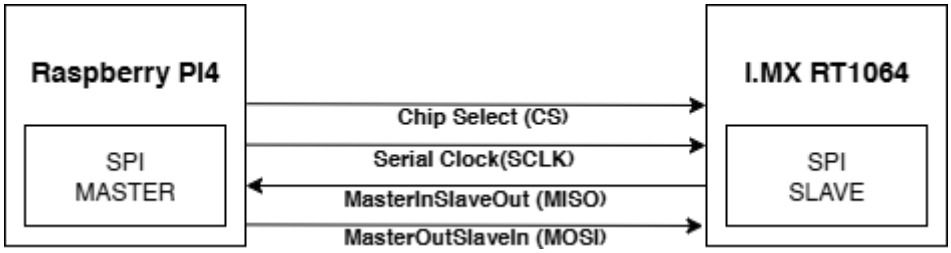
a. Init della periferica FlexCAN e dei filtri
b. Crea un task per la gestione dei messaggi CAN in ingresso che si occuperà di smaltirli
c. Crea un task per il controllo sugli errori ed un eventuale loro correzione
d. Il task per la gestione dei messaggi attende che l'ISR segnali ci siano dei messaggi in ricezione nelle MB allora essi vengono copiati i dati, aggiornati i contatori e riarmate le MB lette.
e. Se si riceve una richiesta di trasmissione allora si scrive sulle mailbox di trasmissione libere, si usano sempre funzioni non bloccanti.

Note di progetto:

Funzionamento reale molto più lungo e complesso omesso, ma in linea di massa coerente con il flusso operativo appena citato.

5.5.3 HAL SPI

Scopo: Collegamento **MCU↔Host** su SPI full-duplex. L'HAL supporta **entrambi i ruoli** (master/slave) via configurazione; in questo caso sarà usato come slave poiché il Raspberry Pi 4 impone il ruolo di *master*.



Funzioni principali:

- Selezione ruolo **master/slave** da configurazione.
- Configurazione parametri concordi fra master e slave: modo (CPOL/CPHA), frequenza target (es. 10 MHz lato master), word size, gestione **CS** (*SSEL*), **MTU**, time-out.
- I/O **non bloccante** con buffer.

- Calcolo del CRC sui messaggi in uscita e controllo del CRC sui messaggi in ingresso (opzionale).

5.5.4 HAL Watchdog

Scopo: Supervisione e reset in caso di fault prolungato.
Funzioni: Configurazione (window/standard), start/stop, **refresh** periodico.
Uso: Il task supervisore effettua il refresh entro la finestra di tempo impostata inizialmente nel watchdog; in anomalia si lascia scadere per garantire un reset completo.

5.5.5 HAL GPT (General Purpose Timer)

Scopo: Timebase di sistema e **timestamp**.
Funzioni: Init prescaler/compare, start/stop, callback periodici, API now() (tick→µs).
ISR: Periodica e minimale: incrementa **contatore monotono** e, se richiesto, notifica task.
Uso: Timestamp per Log/Codec, monitoraggio delle prestazioni.

5.5.6 HAL Codec

Scopo: Codifica/decodifica di messaggi e **segnali CAN** (libreria CoderDBC).
Funzioni: Validazione, endianess, versioning; **registrazioni** lato host (sottoscrizioni con periodo *T*, invii *one-shot* o periodici); mapping ID↔segnali.
Flussi:

- **RX:** HAL CAN → decodifica → applica filtri → inoltro a Host (SPI) e/o Log.
- **TX_1:** Host→serializzazione→SPI→deserializzazione→HAL CAN (invio diretto tramite task oneshot)
- **TX_2:** Host→serializzazione→SPI→deserializzazione→HAL CAN (creazione task periodico di invio). (non realmente usata)

5.6 CoderDBC

CoderDBC è un generatore di codice C che, a partire da un database **DBC** della rete CAN, produce funzioni di **pack/unpack** per messaggi e segnali, includendo mapping ID↔segnali, scaling, offset, unità e controlli di validità. La soluzione adottata in azienda deriva da **c-coderdbc** ed è stata adattata internamente. [26]

5.6.1 Che cos'è un file DBC

Il file **DBC** (CAN database) è un formato testuale che descrive i messaggi di una rete CAN: identificatori, lunghezze, segnali con posizione bit, dimensione, endianess, segno, fattore e offset per la conversione *raw→fisico*, unità, limiti e nodi trasmittenti/riceventi. Il formato, introdotto da **Vector Informatik**, è lo standard de facto per analisi, logging e generazione di codice.
In assenza di un DBC, i frame CAN restano sequenze di byte; con il DBC è possibile **decodificare** i dati in grandezze fisiche (es. *EngineSpeed* in rpm) e **codificare** correttamente i segnali in trasmissione. La sintassi supporta anche il **multiplexing** dei segnali all'interno dello stesso messaggio.
Componenti tipici del DBC:

- Messaggi (**BO_**) con ID e DLC;
- Segnali (**SG_**) con bit start/length, endianess, scaling/offset, unità, min/max e destinatari;
- attributi e commenti (**BA_**, **CM_**);

5.6.2 Dal DBC al codice con CoderDBC

Il flusso di lavoro è lineare:

- **Input**: uno o più file DBC validati dal team di sistema.
- **Generazione**: CoderDBC interpreta il DBC e produce header/sorgenti C con API tipizzate per encode/decode, inclusi controlli su range e coerenza.
- **Integrazione**: le API generate vengono richiamate dalla HAL Codec (CAN) per decodificare i frame ricevuti (pipeline RX) e per comporre i frame da trasmettere (pipeline TX), mantenendo l'applicativo isolato dai dettagli del database.

6 Integrazione con sistema host

6.1 Comunicazione tra livelli di processo: applicativo e real-time

Nel contesto dei sistemi embedded automotive, è frequente la necessità di gestire processi su livelli funzionali differenti: da un lato il **livello real-time**, responsabile del controllo diretto delle periferiche e dell'elaborazione deterministica dei segnali; dall'altro il **livello applicativo**, deputato alla logica di alto livello, all'interfaccia utente e alla gestione dei dati. La comunicazione tra questi due domini è un elemento critico per garantire coerenza, reattività e affidabilità del sistema.

6.1.1 Comunicazione interna su SoC eterogenei

Nella prima fase del progetto, il firmware è stato sviluppato sulla piattaforma NXP i.MX8QM, un System-on-Chip eterogeneo che integra core real-time (ARM Cortex-M4) e quelli applicativi (ARM Cortex-A72/A53). In questo tipo di architettura, i diversi core condividono lo stesso die ma operano in ambienti software distinti, con sistemi operativi e modelli di esecuzione differenti.
Come vediamo nella Figura 23, per consentire la comunicazione tra i due livelli, è stato adottato un meccanismo di **Inter-Process Communication (IPC)** basato su **RPMsg (Remote Processor Messaging)**, protocollo standardizzato per lo scambio di messaggi tra core eterogenei.

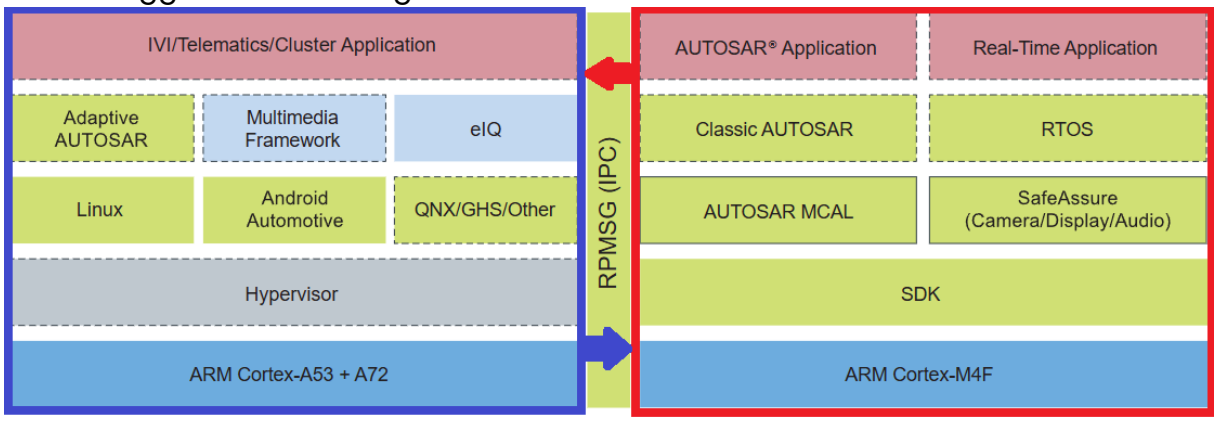


Figura 23 i.MX8QM software architecture [10]

RPMsg consente di trasmettere dati tra il core applicativo (tipicamente Linux/QNX/AUTOSAR su Cortex-A) e il core real-time (Free RTOS/AUTOSAR su Cortex-M) tramite una **memoria condivisa** e una struttura di messaggistica asincrona.
Questa soluzione ha permesso di implementare una comunicazione bidirezionale ro viceversa, senza interferenze dirette sull'esecuzione del firmware.
In teoria, il protocollo garantisce **latenze contenute** e una **buona separazione dei contesti di esecuzione**, consentendo ai due ambienti di operare in modo indipendente e coordinato.

Tuttavia, nella pratica del progetto sono emerse criticità significative, che per motivi di riservatezza non possono essere discusse in dettaglio in questa sede, ma che verranno trattate in termini generali nel Capitolo 7 (

Ottimizzazione e performance).

6.1.2 Transizione verso comunicazione fisica

Con il passaggio a piattaforme **non eterogenee**, in particolare alla **i.MX RT1064** per il livello real-time e al **Raspberry Pi 4** per il livello applicativo, è venuta meno la possibilità di utilizzare memoria condivisa. I due dispositivi sono fisicamente separati e comunicano tramite interfacce hardware. Si è quindi resa necessaria la scelta di un **protocollo di comunicazione fisico** tra MCU e sistema applicativo. I principali candidati considerati sono stati:

- **SPI (Serial Peripheral Interface)**: protocollo sincrono full-duplex, ad alta velocità, semplice da implementare e ampiamente supportato.
- **I²C (Inter-Integrated Circuit)**: protocollo sincrono half-duplex, più lento ma con supporto multi-master e cablaggio semplificato.
- **UART (Universal Asynchronous Receiver-Transmitter)**: protocollo asincrono, semplice e robusto, ma limitato in velocità e gestione dei pacchetti.
- **Automotive Ethernet**: soluzione ad alte prestazioni e scalabilità, ma complessa da configurare e non necessaria per il volume di dati previsto.

Dopo un'analisi comparativa basata su criteri di semplicità, velocità, affidabilità e compatibilità, è stato scelto il protocollo **SPI** come canale di comunicazione tra i due livelli. La scelta è stata motivata dalla sua efficienza in ambienti embedded, dalla disponibilità di periferiche SPI su entrambe le piattaforme e dalla facilità di implementazione lato firmware vista la presenza nella documentazione di qualche esempio.

6.1.3 Implementazione e problematiche SPI

L'implementazione del canale SPI ha richiesto:

- la configurazione dei registri SPI su entrambe le piattaforme,
- la definizione di un protocollo di scambio dati con header, payload e checksum,
- la gestione di buffer e interrupt per garantire la reattività del sistema.

Durante la fase di sviluppo sono emerse alcune problematiche come l'**assenza di driver SPI completi** su QNX (Raspberry Pi), che ha richiesto interventi di adattamento e test manuali.

Durante la fase di porting del firmware sulla **MIMXRT1064-EVK**, è stato necessario affrontare alcune problematiche legate alla **connessione fisica dell'interfaccia SPI**.

Nelle prime prove di comunicazione tra la scheda RT1064 e il **Raspberry Pi 4**, nonostante la corretta configurazione software dei driver SPI forniti dall'SDK NXP, non si otteneva alcuno scambio di dati. Per verificare l'origine del problema, sono stati condotti diversi test comparativi:

- **RT1064 come master e Arduino UNO come slave**: non si osservava alcuna

comunicazione.

- **ESP32 come master e Arduino UNO come slave**: la comunicazione SPI funzionava regolarmente, confermando che l'Arduino UNO operava correttamente come dispositivo slave.
- **Raspberry Pi come master e Arduino UNO come slave**: la comunicazione funzionava, confermando la corretta configurazione lato Raspberry.

Questi test hanno permesso di isolare il problema sulla scheda RT1064.

Per approfondire, la RT1064 è stata configurata come **master SPI** con un programma di prova che scriveva continuamente sul bus. A questo punto è stato utilizzato un **oscilloscopio** per verificare la presenza del segnale di clock sul pin **SCK (Serial Clock)**. Tuttavia, non si rilevava alcun segnale: il pin risultava **flottante**.

Come ulteriore verifica, i pin dell'interfaccia SPI sono stati forzati manualmente a livello logico alto direttamente dal firmware, ma anche in questo caso l'oscilloscopio mostrava un valore flottante, segno che i pin non erano fisicamente collegati ai connettori esterni.

Un'analisi più approfondita dello **schematico della scheda** ha permesso di individuare la causa: le linee dell'SPI verso gli header erano interrotte da **resistenze da 0Ω marcate come DNP (Do Not Place)**, ovvero componenti non montati di fabbrica.

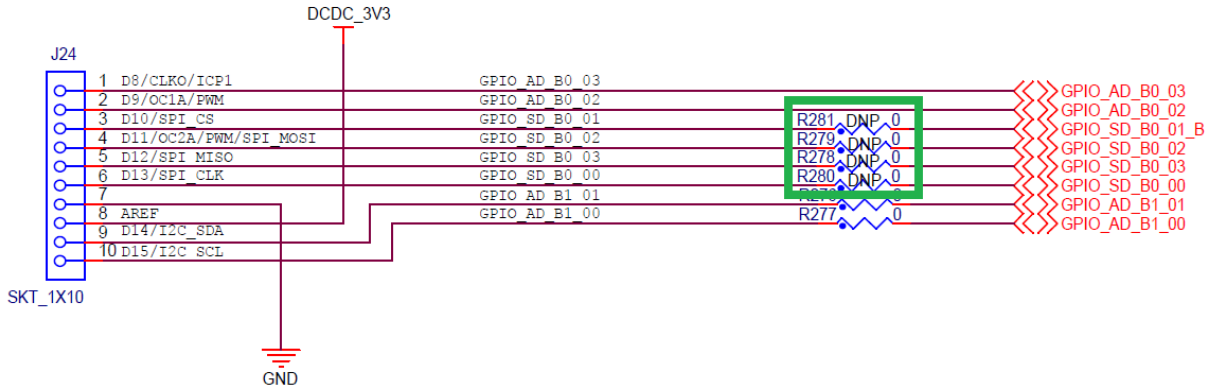


Figura 24 i.MXRT1064 SPI pins connector schematics

Di conseguenza, i pin del microcontrollore non erano effettivamente connessi ai connettori disponibili sulla scheda.

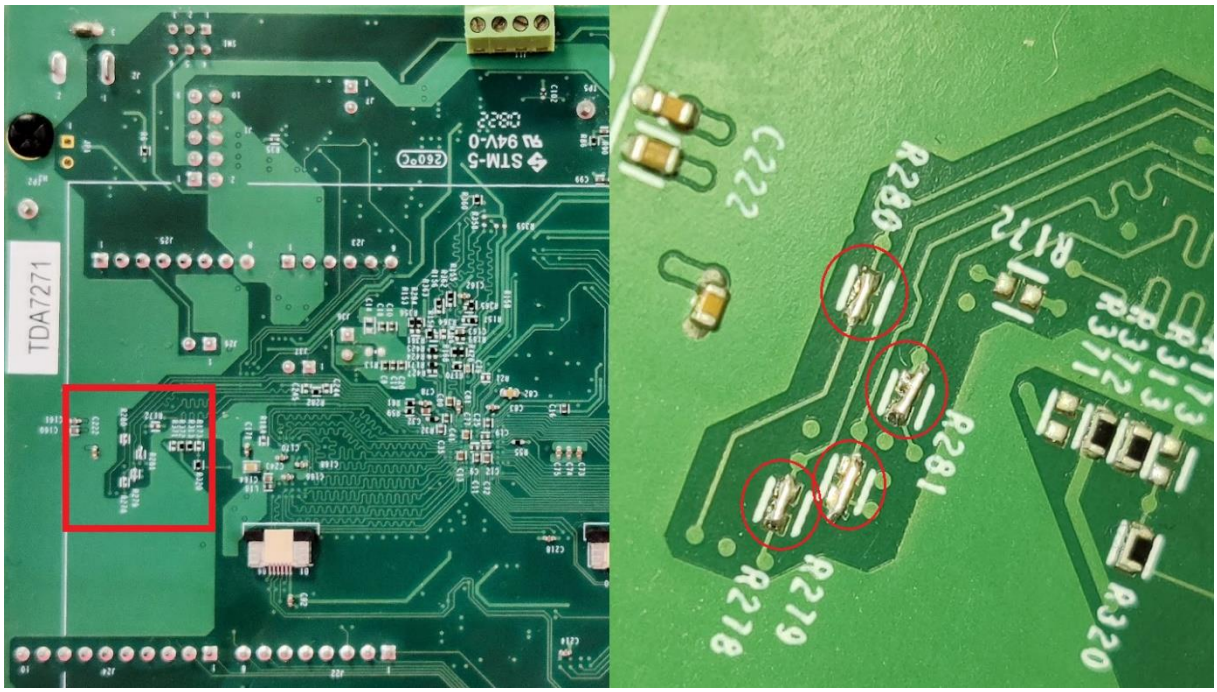


Figura 25 Immagine del PCB con i ponti in rame – Sulla destra un ingrandimento del PCB

Non avendo a disposizione resistenze da 0Ω nel formato 0402 (1x0.5mm) la soluzione è stata quindi quella di realizzare **ponti fisici sulla PCB**, saldando manualmente dei ponti ricavati da singoli pezzetti di rame, spessi poco più di un capello, per ristabilire i collegamenti necessari per abilitare correttamente l'interfaccia SPI.
Seppur si tratti di una soluzione semplice, risalire al problema è stato tutt'altro che banale. Alla fine, la comunicazione tra **RT1064** e **Arduino Uno** è stata finalmente stabilita con successo, consentendo di proseguire con l'integrazione del firmware.

6.2 Serializzazione dati per SPI

Per consentire lo scambio di informazioni tra il microcontrollore dedicato alla gestione della rete CAN e l'applicazione di alto livello in esecuzione sull'host, è stato necessario definire un meccanismo di **serializzazione dei dati**. Tale passaggio è fondamentale poiché la comunicazione avviene attraverso il bus **SPI**, che trasferisce sequenze di byte prive di semantica: spetta quindi al livello software stabilire un formato condiviso che permetta di codificare e decodificare i messaggi in modo coerente.
In letteratura e nella pratica industriale sono disponibili diversi approcci e librerie per la serializzazione ma i principali risultano essere due:

- **FlatBuffers** (Google): progettato per efficienza e accesso diretto ai dati senza necessità di deserializzazione completa.
- **Protocol Buffers (Protobuf)**: ampiamente diffuso, con forte supporto multi-linguaggio e schema evolutivo.

Nel progetto è stato adottato FlatBuffers, che ha garantito un buon compromesso tra efficienza, portabilità e semplicità di integrazione. La definizione di uno schema

ha permesso di descrivere in maniera formale la struttura dei messaggi CAN e di generare automaticamente il codice per encoder e decoder, riducendo il rischio di errori e mantenendo la coerenza tra MCU e applicazione host.

In questo modo, il flusso dati dal microcontrollore all'applicazione di alto livello è risultato deterministico, compatto e facilmente estendibile, assicurando la corretta interpretazione dei messaggi CAN anche in presenza di evoluzioni future del protocollo o dell'architettura software.

6.2.1 Redundant checks

Per rendere robusto il trasporto su SPI, il messaggio **FlatBuffers** viene incapsulato con due metadati: **lunghezza** del payload e **CRC a 16 bit** del payload. Il framing resta leggero e indipendente dal contenuto serializzato.

Formato (header + payload):

LEN (16 bit, little-endian) | CRC16 (16 bit, little-endian) | PAYLOAD (FlatBuffers)

- **LEN**: numero di byte del solo payload FlatBuffers (0…65535).
- **CRC16:** calcolato sul solo payload FlatBuffers (polinomio standard IEEE 0x1021).
- **PAYLOAD**: buffer FlatBuffers generato dallo schema (frame CAN, log, cfg, ecc.).

Segue un breve esempio di pseudo-codice C per CRC16-CCITT (polinomio 0x1021)

Tabella 2 CRC16 classico

```
uint16_t crc16_ccitt(uint8_t *data, uint32_t length) {
    uint16_t crc = 0xFFFF;         // valore iniziale
    uint16_t polynomial = 0x1021; // polinomio generatore

    for (size_t i = 0; i < length; i++) {
        crc ^= (uint16_t)data[i] << 8; // XOR con il MSB

        for (int bit = 0; bit < 8; bit++) {
            if (crc & 0x8000) {   // se il bit più significativo è 1
                crc = (crc << 1) ^ polynomial;
            } else {
                crc = crc << 1;
            }
        }
    }

    return crc & 0xFFFF; // risultato finale a 16 bit
}
```

7 Ottimizzazione e performance

Questo capitolo racconta come il firmware sia stato raffinato per essere rapido, stabile e prevedibile sotto carico. L'obiettivo non è solo "andare veloce", ma farlo in modo costante: tempi regolari, percorsi critici snelli, consumi di risorse sotto controllo.

Il lavoro si è mosso per passi: prima la misura (strumenti semplici ma affidabili per osservare ciò che accade davvero), poi la lettura dei dati per individuare i colli di bottiglia, infine la messa a punto del codice e dell'organizzazione interna. Ne è derivata una catena più lineare: le ISR fanno il minimo indispensabile, i task assorbono il lavoro pesante, i buffer sono dimensionati sul comportamento reale e il logging non intralcia il tempo reale.

Il risultato è un sistema più "elastico": regge meglio i picchi, reagisce in modo prevedibile e resta facile da mantenere. Le sezioni che seguono presentano gli strumenti usati, le metriche osservate e le principali scelte che hanno portato a questo equilibrio.

7.1 Strumenti di Misura

Le misure si sono basate su quattro pilastri:

- Il **DWT/CYCCNT** del Cortex-M7(RT1064) e del Cortex-M4 (i.MX8) ha fornito i cicli di CPU, così da stimare con precisione il tempo consumato da task e sezioni critiche (pack/unpack, ISR CAN, serializzazione SPI).
- Il **GPT** ha offerto una timebase stabile per timestamp e finestre di osservazione, utile per calcolare latenze e jitter senza dipendere dal carico istantaneo.
- I **contatori interni** hanno tracciato stato di mailbox e buffer, retry su SPI, CRC falliti, pacchetti decodificati, watermark massimi.
- Infine, il **Log** ha raccolto eventi e metriche con timestamp, in modo non bloccante, e li ha inviati periodicamente ed un computer per la loro registrazione.

Per l'analisi post-run è stato impiegato uno script Python che acquisiva in tempo reale i record (log) via seriale e generava un file Excel con tabelle e fogli preimpostati per grafici. Questo ha permesso di confrontare rapidamente build e scenari (idle, 100% bus, host >80% CPU, ecc) e di evidenziare trend e correlazioni.

7.2 Metriche Principali

Le metriche chiave hanno riguardato l'intera catena di elaborazione. La **latenza end-to-end MCU→Host** (dall'ingresso in ISR RX alla consegna su SPI) è stata analizzata su P50/P95 e correlata con il carico dell'host. La **latenza delle ISR** e il relativo jitter sono stati misurati con timestamp GPT e, quando necessario, in cicli tramite DWT. I **CPU cycles** hanno quantificato il costo delle zone più critiche (pack/unpack, serializzazione SPI, ISR CAN). Il throughput e i tassi di errore/retry (SPI, CAN), insieme a overflow e drop, hanno validato la robustezza sotto saturazione bus. Per la parte RTOS, sono stati monitorati lo stack high watermark dei task e **quota di uso CPU** per ciascun task, così da calibrare dimensionamento e priorità.

Tabella 3 Metriche e sorgenti dati

Metrica	Definizione sintetica	Sorgente/Strumento
Latenza MCU→Host	RX ISR → invio completato su SPI	Timestamp GPT
Latenza ISR / Jitter	t_out - t_in e varianza	GPT (tempo), DWT (cicli)
CPU cycles (hot paths)	Costo pack/unpack, SPI, ISR CAN	DWT (cicli di clock)
Throughput / Errori	Frame/s, retry, CRC fail, overflow	Contatori interni
Stack per task	Picco di stack utilizzato	Telemetria RTOS
% CPU per task	Tempo CPU normalizzato su finestra	DWT (sampling) + aggregazione

7.3 Individuazione dei problemi

L'analisi ha evidenziato quattro criticità ricorrenti, osservate in modo trasversale alle piattaforme:

- **Accoppiamento tra livelli su piattaforme multi-core.**
  I meccanismi di IPC possono introdurre dipendenze temporali forti tra logica applicativa e MCU. Il risultato è una catena meno prevedibile, sensibile a ritardi e back-pressure.
- **Indipendenza limitata del core di controllo.**
  Senza un partizionamento accurato del sistema, tramite l'SCU, il core real-

time non risulta pienamente autonomo. In caso di fault, il ripristino richiede reset selettivi di periferiche e riallineamenti di stato, con tempi di recupero meno lineari o addirittura impossibile uscire da situazioni critiche gravi.

- **Moltiplicazione del payload verso l'applicazione.**
  La decodifica dei messaggi in numerosi segnali elementari e la loro rappresentazione uniforme (es. valori floating-point per grandezze booleane o intere piccole) amplificano il traffico sul link MCU←→applicazione, con impatto su latenza e jitter. Un singolo messaggio CAN può contenere 8byte, di conseguenza ci potrebbero essere fino a 64 singoli segnali (booleani) per ogni messaggio CAN. Se rappresentati da Double 64bit(8B) significa passare da un ingresso di 8byte ad un uscita di 64*8B = 512B ovvero di quasi 2 ordini di grandezza in più.
- **Operazioni bloccanti nelle zone critiche.**
  Scritture sincrone su seriale/log e chiamate bloccanti in sezioni critiche generano rallentamenti episodici, anche quando l'utilizzo medio di CPU rimane basso. L'effetto è una variabilità della risposta, difficile da controllare sotto carico.

7.4 Interventi di ottimizzazione

Gli interventi sono stati organizzati in tre direttrici:

- **architettura**
- **codec/serializzazione**
- **zone di codice critiche e risorse**

7.4.1 Architettura: eliminazione dell'IPC e autonomia del controllo

Problema: Accoppiamento tra livelli dovuto all'IPC e limitata indipendenza del core di controllo.
Intervento: L'IPC è stato rimosso dall'equazione: è stata infatti adottata una MCU completamente separata per la parte real-time (vedi Cap. 3.2.3). Questo ha eliminato le dipendenze temporali con l'applicazione di alto livello e ha reso efficace il watchdog di sistema (ripristino rapido senza dover resettare le singole periferiche).
Effetto atteso: Catena temporale più pulita, latenza più prevedibile, recovery semplice e lineare.

7.4.2 Codec e serializzazione: generazione mirata e formati compatti

Problema: Traffico e lavoro superfluo dovuti alla decodifica/serializzazione dei segnali e a strutture decisionali poco efficienti.

Intervento:

- **CoderDBC ottimizzato:** generazione di **switch-case** al posto di alberi di **if-else**; minor branching e percorso più deterministico.
- **Pre-filtri di generazione:** il codice prodotto include **solo** messaggi e segnali **di interesse**; eliminati pack/unpack inutilizzati.
- **Riduzione del payload verso l'host:** rappresentazioni tipizzate coerenti (booleani/interi dove basta), invio **selettivo** e/o **aggregazione** quando opportuno. (non completamente implementata, vedi 7.5)
- **Framing robusto su SPI:** header con **lunghezza** e **CRC**, retry limitati.

Effetto atteso: Meno banda occupata, minore lavoro di decodifica, latenza e jitter più stabili.

7.4.3 Zone di codice critiche e gestione risorse

Problema: Variabilità dovuta a lavoro e I/O bloccante in sezioni sensibili; dimensionamento non ottimale dei buffer.

Intervento:

- **ISR leggere e costanti:** solo drenaggio buffer/mailbox, contatori e notifica ai task; nessuna allocazione.
- **Batch nei task** per ridurre risvegli frequenti e sfruttare i tempi morti.
- **Priorità NVIC** confermate: sorgenti tempo-critiche sopra quelle non critiche (UART/log).
- **Log non bloccante** con buffer Ping/Pong e se sotto stress scarta semplicemente i messaggi per dare priorità agli altri task;
- **Buffer sizing** sulla base dei **watermark P99**; abolito l'uso di **heap** nelle sezioni real-time, introdotti **pool a blocchi fissi** dove necessario. **Effetto atteso.** Maggior determinismo, assenza di stalli da I/O, robustezza ai picchi con footprint sotto controllo.

Tabella 4 CRC16 con lookup table

```
static const uint16_t crc16_table[256] = {
    // Tabella precalcolata per il polinomio 0x1021
    // (valori generati una volta e riutilizzati)
    // ...
};

uint16_t crc16_ccitt(uint8_t *data, uint32_t length) {
    uint16_t crc = 0xFFFF;          // valore iniziale

    for (size_t i = 0; i < length; i++) {
        pos = (uint8_t)((crc >> 8) ^ data[i]);
        crc = (crc << 8) ^ crc16_table[pos];
    }

    return crc & 0xFFFF; // risultato finale a 16 bit
}
```

In *Tabella 4* possiamo vedere una revisione del codice della *Tabella 2*, in questo

caso l'ottimizzazione è presente usando una lookup table da 256B invece che calcolando i valori ogni volta, questo porta un miglioramento in particolare sui calcoli ripetuti, come per l'appunto sui dati trasmessi via SPI.

7.5 Inoltro del frame CAN su SPI

L'esperimento è stato concepito per affrontare un problema non tanto di carico computazionale sulla MCU, quanto di **efficienza del bus SPI**. Nella configurazione classica, infatti, la MCU spacchetta i frame CAN in singoli segnali e li serializza verso l'host. Questa strategia, pur funzionando, comporta un aumento significativo del traffico sul link SPI, con conseguente rischio di saturazione e jitter nei burst di comunicazione.

Per ridurre questo overhead, è stata testata una modalità alternativa: l'inoltro "grezzo" del frame CAN, comprensivo di ID, DLC, payload e timestamp, direttamente su SPI. In questo schema la MCU (i.MX RT1064) si limita a trasferire il frame così come ricevuto, aggiungendo soltanto un **framing leggero** con campi di lunghezza (LEN) e controllo d'integrità (CRC16), oltre ad alcuni campi opzionali (TYPE, SEQ, VER) utili per tracciabilità e versioning.

La decodifica dei segnali viene quindi demandata al sistema host, basato su QNX. Qui è stato possibile riutilizzare lo stesso **CoderDBC** già impiegato sul microcontrollore: essendo scritto in C/C++, l'adattamento al Raspberry Pi è risultato immediato e senza particolari difficoltà. In questo modo, l'host si occupa di interpretare i frame secondo il DBC, applicare filtri e instradare i dati verso l'applicazione finale, liberando la MCU da questo compito e riducendo il traffico sul bus.

Le prime prove hanno evidenziato diversi vantaggi:

- **Riduzione della banda SPI occupata**, poiché si trasferiscono solo gli 8 byte del payload (più header) invece di una molteplicità di segnali già serializzati.
- **Latenza più regolare nei burst**, con picchi ridotti e jitter contenuto.
- **Maggiore flessibilità lato host**, dove eventuali aggiornamenti del DBC possono essere gestiti direttamente senza ricompilare il firmware.

In questo contesto, l'uso di un ulteriore livello di serializzazione come **FlatBuffers non è strettamente necessario**: il frame CAN è già nativamente strutturato, e l'integrità è garantita dal meccanismo LEN + CRC16. FlatBuffers rimane utile solo se si desidera aggiungere metadati applicativi ulteriori.

Naturalmente, l'esperimento ha avuto durata limitata e non ha coperto tutti i corner case.
Restano da approfondire l'integrazione con la pipeline di telemetria e logging lato host, la gestione dei fault in presenza di messaggi corrotti o multiplexing

complesso, e la definizione di politiche di QoS. Tuttavia, i risultati preliminari suggeriscono che questa soluzione possa migliorare il determinismo della MCU, ridurre il carico sul bus SPI e semplificare la manutenzione grazie alla centralizzazione della decodifica su QNX.

8 Testing e validazione

Questo capitolo descrive metodologia, strumenti e risultati della validazione. I dati numerici riportati sono esemplificativi e hanno finalità illustrative (non rappresentano misure reali provenienti dai test).

8.1 Metodologia di test

Approccio. Validazione progressiva: **banco** → **simulazioni** → **veicolo**. Ogni fase misura latenze, throughput, affidabilità del link MCU↔host e resilienza ai fault.

8.1.1 Banco e simulazioni: CANcase + CANalyzer

Le prove **su banco** e in **simulazione** sono state condotte con **Vector CANcase** e **CANalyzer**. Il setup ha permesso di:

- **Generare traffico CAN** controllato (singoli messaggi, burst, sequenze periodiche) con timing e bitrate configurabili.
- **Riprodurre tracce registrate a veicolo**, replicando cicli guida ed eventi rari in ambiente di laboratorio.
- **Iniettare condizioni di errore** (ID inattesi, DLC variabile, fluttuazioni di carico) e verificarne la gestione lato firmware.
- **Filtrare e monitorare** frame e segnali con logging sincronizzato, oltre ad **automatizzare** campagne ripetibili per il confronto pre/post.
- **Essere ripetibile**, infatti si possono creare script e lanciare le stesse simulazioni più volte così da poter avere risultati comparabili.

Figura 26 Vector CANcase VN4610

8.1.2 Strumentazione e raccolta dati

- **Timestamp e cicli**: GPT per misure temporali e DWT/CYCCNT per cicli CPU nelle zone di codice critiche.
- **Telemetria interna**: contatori overflow/retry/CRC, watermark, stack peak, quota CPU per task.
- **Log strutturato**: eventi con timestamp (fonte GPT) su canale non bloccante.
- **Script Python**: acquisizione via seriale e generazione di un **file Excel** con distribuzioni (P50/P95) e trend.

Approfondimenti al capitolo 7.

8.1.3 In veicolo

Test di integrazione a bordo dell'IVECO S-WAY: coerenza dei segnali decodificati, stabilità del link MCU↔host, assenza di perdite percepibili. Logging a campionamento ridotto, focalizzato su eventi e metriche principali.
Le finestre di prova su veicolo sono state limitate poiché prioritarie per il team di sviluppo di altre componenti software/hardware; in tale contesto è stato comunque fornito supporto operativo ai loro test.

8.2 Risultati sintetici

Esempio di miglioramenti ottenuti dopo le ottimizzazioni del Cap. 7: latenza più stretta, throughput stabile a pieno carico, meno errori su link e risorse più sotto controllo.

KPI (condizioni note)	Pre ottimizzazione	Post ottimizzazione
Latenza MCU→Host P50 (ms)	1,6	1,2
Latenza MCU→Host P95 (ms)	3000,8	2,2
Average RX MB used	1	1
Jitter ISR CAN (ms, σ)	0,35	0,18
Throughput @100% bus (drop %)	1,2%	0,0%
Retry SPI su 1 h (%)	NA	0,06
CRC16 fail su 1 h (#)	NA	1
Stack peak task CAN_RX (KiB)	7,5	5,0
Average %CPU task CAN_DECODE	22%	21%
Average %CPU task total	40%	25%
Max. %CPU task total	60%	90%

9 Conclusioni

Il lavoro ha portato a un firmware più prevedibile, stabile e osservabile.
L'architettura con MCU separata, le ISR leggere, i buffer, il framing LEN+CRC16 su SPI invece che su shared memory e l'ottimizzazione del CoderDBC (switch-case e pre-filtri) hanno ridotto latenza, jitter e traffico superfluo.

Contributo dell'esperimento 7.5, l'inoltro "grezzo" del frame CAN su SPI, con decodifica lato host, ha mostrato riduzione della banda sul link superiore al 70%, latenza più regolare nei burst e maggiore flessibilità nell'evoluzione del DBC.
L'approccio rende opzionale un'ulteriore serializzazione applicativa (es. FlatBuffers) essendo i messaggi CAN già serializzati.

9.1 Valutazione critica

Sebbene ci siano tanti punti di forza, come:

- Architettura semplificata: MCU indipendente, watchdog efficace, pipeline deterministica.
- HAL coerente con real-time: ISR leggere, task di smaltimento, priorità NVIC adeguate.
- Codec più efficiente: switch-case, pre-filtri, meno branching; formati compatti sul link.
- Osservabilità: metriche e log strutturati facilitano diagnosi e regressioni.

Sono presenti altrettanti punti critici da migliorare:

- Copertura su veicolo limitata; necessarie campagne più ampie e condizioni ambientali variabili.
- Modalità "forward grezzo" validata su finestra breve: da estendere a multiplexing complesso, scenari lunghi.
- Sicurezza del canale MCU←→host oggi centrata solo su integrità (**CRC16**)
- Assenza di **CAN FD** e J1939 nel prototipo: payload e bitrate vincolati a CAN 2.0
- Presenza di BUG e funzionalità non del tutto testate

Le scelte architetturali e le ottimizzazioni introdotte hanno posto basi solide per un firmware real-time affidabile e mantenibile. L'esperimento di decodifica lato host indica una direzione promettente per ridurre il carico su SPI e migliorare il determinismo, preservando al contempo la flessibilità necessaria all'evoluzione del sistema. Resta tuttavia il vincolo di piattaforme e schede **non certificate**, con implicazioni sul perimetro di impiego.
Alla luce degli studi condotti, risulta inoltre più semplice e rapido, a fronte di **una curva di apprendimento iniziale più ripida**, sviluppare un applicativo con requisiti analoghi adottando **AUTOSAR**, grazie a componenti standardizzati, processi maturi e percorsi di certificazione più lineari.

10 Bibliografia

[1] «CAN in Automation (CiA) "History of CAN Technology.",» [Online]. Available: https://www.can-cia.org/can-knowledge/history-of-can-technology. [Consultato il giorno 19 agosto 2025].

[2] «Vector Informatik GmbH. "Automotive Ethernet.",» [Online]. Available: https://support.vector.com/kb?id=kb_article_view&sysparm_article=KB0013990&sys_kb_id=c0dd16cf3ba122509a9c6a34c3e45a61&spa=1#Automotive. [Consultato il giorno 19 agosto 2025].

[3] «Porsche AG. "Porsche – Future of Code.",» Medium, 2021. [Online]. Available: https://medium.com/next-level-german-engineering/porsche-future-of-code-526eb3de3bbe. [Consultato il giorno 19 agosto 2025].

[4] «Cars are Made of Code,» NXP, 2017. [Online]. Available: https://www.nxp.com/company/about-nxp/smarter-world-blog/BL-CARS-MADE-CODE. [Consultato il giorno 21 agosto 2025].

[5] NXP, «101: Controller Area Network (CAN) standard,» 2021. [Online]. Available: https://community.nxp.com/t5/NXP-Tech-Blog/101-Controller-Area-Network-CAN-standard/ba-p/1217054. [Consultato il giorno 28 settembre 2025].

[6] A. Alfardus e D. Rawat, «Evaluation of CAN Bus Security Vulnerabilities and Potential Solutions,» 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10145267. [Consultato il giorno 2 ottobre 2025].

[7] M. Falch, «A simple intro to CAN FD,» 2022. [Online]. Available: https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro. [Consultato il giorno 26 settembre 2025].

[8] J. Schneider e I. Smalley, «What is a microcontroller?,» [Online]. Available: https://www.ibm.com/think/topics/microcontroller. [Consultato il giorno 2 Ottobre 2025].

[9] NXP, «NXP i.MX8QM MEK,» [Online]. Available: https://www.nxp.com/design/design-center/development-boards-and-designs/i-mx-evaluation-and-development-boards/i-mx-8quadmax-multisensory-enablement-kit-mek:MCIMX8QM-CPU. [Consultato il giorno 1 giugno 2025].

[10] NXP, «NXP i.MX8,» [Online]. Available: https://www.nxp.com/products/i.MX8. [Consultato il giorno 1 giugno 2025].

[11] M. Rodriguez, «Introduction to the System Controller Firmware on i.MX8 Processor,» ottobre 2019. [Online]. Available: https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/conn

ects/206/1/AMF-AUT-T3889_Introduction%20to%20the%20System%20Controller%20Firmware%20on%20i.MX%208%20Application%20Processor_rev.pdf. [Consultato il giorno 1 dicembre 2024].

[12] NXP, «S32k312 evaluation board,» [Online]. Available: https://www.nxp.com/design/design-center/development-boards-and-designs/S32K312EVB-Q172. [Consultato il giorno 16 febbraio 2025].

[13] NXP, «i.MX RT1064 Evaluation Kit,» [Online]. Available: https://www.nxp.com/design/design-center/development-boards-and-designs/MIMXRT1064-EVK. [Consultato il giorno 10 maggio 2025].

[14] wikipedia, «Hexagonal architecture (software),» 30 luglio 2025. [Online]. Available: https://en.wikipedia.org/wiki/Hexagonal_architecture_(software). [Consultato il giorno 4 ottobre 2025].

[15] freertos, «freertos,» [Online]. Available: https://www.freertos.org/. [Consultato il giorno 10 novembre 2024].

[16] freertos, «freertos fundamentals,» [Online]. Available: https://www.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/01-RTOS-fundamentals. [Consultato il giorno 7 ottobre 2025].

[17] freertos, «freertos scheduling,» [Online]. Available: https://freertos.org/Documentation/02-Kernel/02-Kernel-features/01-Tasks-and-co-routines/04-Task-scheduling. [Consultato il giorno 7 ottobre 2025].

[18] qnx, «blackberry qnx,» [Online]. Available: https://blackberry.qnx.com/en. [Consultato il giorno 18 settembre 2025].

[19] wikipedia, «wikipedia QNX,» [Online]. Available: https://it.wikipedia.org/wiki/QNX. [Consultato il giorno 18 settembre 2025].

[20] J. Schneider e I. Smalley, «What's an integrated development environment(IDE) ?,» ibm, [Online]. Available: https://www.ibm.com/think/topics/integrated-development-environment. [Consultato il giorno 6 ottobre 2025].

[21] NXP, «Getting started with MCUXpresso SDK for EVK-MIMXRT1064,» NXP, 2024. [Online]. Available: https://mcuxpresso.nxp.com/mcuxsdk/24.12.00/html/boards/evkmimxrt1064/gettingStarted/topics/overview.html. [Consultato il giorno 20 settembre 2025].

[22] «MCUXpresso Software Development Kit,» [Online]. Available: https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-software-development-kit-sdk:MCUXpresso-SDK. [Consultato il giorno 8 ottobre 2025].

[23] LinuxBaya, «Cross-compilation,» [Online]. Available: https://www.linuxbaya.in/2020/09/cross-platform-development-toolchain.html. [Consultato il giorno 28 settembre 2025].

[24] embeddedwala, «What is ISR?,» 11 aprile 2024. [Online]. Available: https://embeddedwala.com/EmbeddedSystems/embedded-c/what-is-isr. [Consultato il giorno 3 ottobre 2025].

[25] h. Adams, «UART,» [Online]. Available: https://vanhunteradams.com/Protocols/UART/UART.html. [Consultato il giorno 2025].

[26] astand, «c-coderdbc,» [Online]. Available: https://github.com/astand/c-coderdbc. [Consultato il giorno 15 novembre 2024].

[27] NXP, «S32K1xx reference manual,» [Online]. Available: https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/S32K/32215/1/S32K-RM.pdf. [Consultato il giorno 1 giugno 2025].

[28] wikipedia, «freertos wikipedia,» [Online]. Available: https://it.wikipedia.org/wiki/FreeRTOS. [Consultato il giorno 20 settembre 2025].