UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Sviluppo di driver per la comunicazione con la centralina di controllo motore GPEC4LM

Relatore: Fantozzi Carlo

Laureando: Fazzi Riccardo

Anno Accademico: 2022 - 2023

Data di laurea: 21/03/2023

Sommario

Il progetto descritto in questa tesi consiste nello sviluppo del driver per la comunicazione con la centralina di controllo motore GPEC4LM.
Nel capitolo introduttivo viene presentata l'azienda coinvolta nel progetto, inoltre si descrive cos'è e quale scopo ha una centralina di controllo motore.
Nel secondo capitolo vengono illustrati gli obiettivi del progetto e le varie fasi che hanno portato alla realizzazione del driver, per concludersi poi con la fase di verifica di funzionamento del sistema.
Nel capitolo conclusivo del testo, vengono tratte le conclusioni relative al progetto, alle conoscenze acquisite e all'utilizzo possibile di tali conoscenze per eventuali altri progetti correlati. Inoltre viene menzionato il mondo dell'industria automobilistica, con uno sguardo particolare rivolto al settore delle centraline.

Indice

1 Introduzione

1.1 L'azienda

Figura 1: Logo azienda Dfb

dfb technology è un'azienda specializzata nel settore automotive che si occupa dello sviluppo di strumenti per la programmazione di centraline motore e cambio appartenenti ad una vasta gamma di veicoli, tra cui: auto, moto, autocarri, mezzi agricoli e veicoli marini. Il prodotto destinato al cliente consiste in un kit completo per la programmazione della maggior parte delle centraline presenti in commercio, utilizzando il protocollo CAN-bus con differenti modalità di accesso.

Figura 2: Kit Dfox.

In figura viene mostrato il kit prodotto dall'azienda contenente: il cablaggio necessario per operare sulle centraline, le varie periferiche, il manuale d'uso e lo strumento principale: il Dfox pro.

1.2 Il Dfox Pro

Tutte le operazioni disponibili per le centraline sono accessibili unicamente attraverso il Dfox Pro. Questo strumento è sostanzialmente un sistema hardware che si occupa di effettuare la comunicazione seriale, tramite il protocollo CAN-bus (illustrato al capitolo 2.3.2), tra una centralina e un computer. Per poter gestire il sistema è presente un software dedicato sviluppato dall'azienda, contenente i driver relativi ai vari modelli di centraline della maggior parte dei marchi presenti sul mercato.
Sullo strumento sono presenti due interfacce per le 4 modalità di comunicazione.

Figura 3: Dfox Pro.

Le modalità di accesso supportate sono le seguenti:

1.2.1 OBD Mode

Questa modalità consente di leggere e scrivere la centralina attraverso l'ausilio della porta OBD, comunemente utilizzata per le attività di diagnostica del veicolo. Tale approccio si dimostra notevolmente rapido, in quanto non richiede l'estrazione della centralina dal mezzo. Tuttavia ci sono alcune limitazioni rispetto alle modalità che sono elencate in seguito.

Figura 4: OBD mode

1.2.2 BENCH Mode

Attraverso questa modalità, è possibile accedere alla centralina per la lettura e la scrittura completa, senza dover rimuovere la scocca. Ciò evita il rischio di invalidare la garanzia o di danneggiare i componenti della scheda. Tale approccio richiede l'estrazione della centralina dal mezzo.

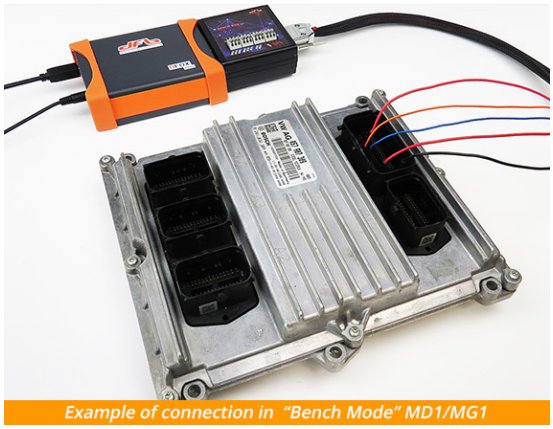

Figura 5: bench mode

1.2.3 BOOT Mode

Attraverso questa modalità, è possibile effettuare una comunicazione a basso livello con il microprocessore della centralina. Risulta utile quando viene bloccato l'accesso in bench mode, oppure quando non è predisposto da parte della casa produttrice. Tale approccio richiede l'estrazione della centralina dal mezzo e la rimozione della scocca, per poter accedere ad alcuni pin sulla scheda madre.

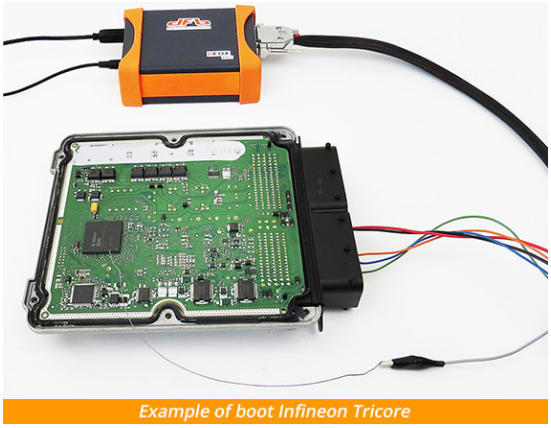

Figura 6: boot mode

1.2.4 JTAG Mode

Questa modalità è molto simile alla precedente, però viene impiegata per un diverso scopo, infatti si utilizza soprattutto per operazioni di debug, in caso di malfunzionamenti hardware o software. Il collegamento avviene tramite flat cable che vengono saldati sulla scheda della centralina, per accedere direttamente al processore e al bus di sistema. Tale approccio richiede l'estrazione della centralina dal mezzo e la rimozione della scocca.

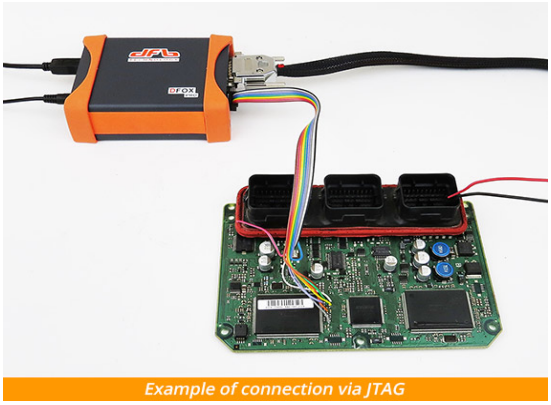

Figura 7: jtag mode

1.3 Come dfb si approccia al mondo automotive

All'interno della maggior parte dei veicoli moderni, sono presenti decine di unità di controllo che gestiscono il funzionamento ad esempio dell'impianto di illuminazione, l'infotainment, l'azionamento dei finestrini, i tergicristalli, e molto altro ancora.
Alcune tra queste unità sono però di fondamentale importanza, in quanto si occupano dei componenti fondamentali e più critici di un veicolo, come ad esempio:

- **ECU** (engine control unit): gestione del motore,
- **TCU** (transmission control unit): gestione della trasmissione,
- **PCU** (powertrain control unit): ECU e TCU unite in un'unica centralina.

Se si necessita di operare su alcune di queste unità di controllo, è essenziale utilizzare uno strumento adeguato, ed è proprio qui che dfb technology si distingue nel settore, offrendo una soluzione innovativa e affidabile per questa tipologia di dispositivi.
Di seguito vengono illustrate soltanto due tra le principali modalità mediante le quali questo strumento può essere impiegato, tuttavia non sono le uniche.

1.3.1 Il tuning

Un settore sviluppato che vede come protagonista un'azienda come dfb technology è quello del *tuning* software. Nei veicoli moderni, le prestazioni e l'efficienza non sono più determinate esclusivamente dalla qualità dei componenti e dalla struttura del veicolo; un contributo è fornito anche dall'utilizzo di sofisticati sistemi elettronici che ne regolano il funzionamento in maniera ottimale. Viene indicato con il termine "*mappa della centralina*" l'insieme di parametri che controllano il funzionamento dei componenti di cui la centralina stessa si occupa. Modificando questi parametri è possibile rimodellare il regime di funzionamento del motore, ad esempio fornendo più potenza, più efficienza, minor consumo di carburante e molto altro ancora. Grazie al kit progettato da dfb technology, è possibile effettuare questa procedura, estraendo "le mappe" dalla centralina per poi modificarle e reinserirle una volta modificate.

1.3.2 La clonazione

Nell'eventualità che alcune centraline subiscano dei danni, risulta spesso difficile e poco conveniente effettuare una riparazione. Questo perché non esiste una standardizzazione nel settore, ma soprattutto è difficile reperire manuali o schemi di progettazione in quanto la

maggior parte delle volte non vengono resi pubblici dalle case produttrici. Molte tra queste centraline non risultano essere economiche, in quanto alcune di esse possono raggiungere un prezzo di diverse migliaia di euro, cifra che rappresenta una somma considerevole in relazione al costo del veicolo medesimo. In situazioni come queste entra in gioco il kit dfox, che permette di effettuare la **clonazione** del contenuto della centralina in un'altra eventualmente di seconda mano, reperita ad un prezzo ragionevole.

1.4 Engine Control Unit (ECU)

Poiché le centraline motore sono state menzionate diverse volte, è necessario approfondire l'argomento per consentire ai lettori di comprendere al meglio i capitoli successivi.
Nell'industria automotive ci si riferisce alle centraline motore con il termine **ECU** (Engine Control Unit), ovvero unità di controllo del motore.

1.4.1 Perchè nascono le ECU?

La gestione del motore a ciclo Otto è fondamentale per poter esprimere tutto il suo potenziale e limitare le emissioni inquinanti.
Prima della diffusione delle ECU, i sistemi in dotazione nei veicoli erano per lo più meccanici, di conseguenza l'efficienza e le prestazioni risultavano molto limitate. Uno dei principali componenti meccanici, per la gestione dell'impianto di alimentazione è il **carburatore**, che si occupa di preparare la miscela di comburente (aria) e carburante (ad esempio benzina) da inviare in camera di scoppio. Questa miscela deve rispettare un **rapporto stechiometrico** ben definito per consentire al motore di esprimere le proprie caratteristiche al meglio. In questi sistemi meccanici la quantità di miscela fornita al motore è determinata da una semplice valvola a farfalla, o da una saracinesca posta all'interno del carburatore, che a seconda della propria apertura può favorirne oppure ostacolarne il passaggio. La principale difficoltà di questo sistema di alimentazione è quella di non riuscire a fornire un corretto rapporto stechiometrico della miscela al variare del carico, in particolare in situazione di utilizzo gravoso del veicolo.
Con lo sviluppo del settore automotive, nascono i sistemi a iniezione elettronica, ovvero dei sistemi di alimentazione basati su componenti denominati **iniettori**, controllati elettronicamente da una centralina, che ne regola il funzionamento.
Esistono due tipi di iniezione elettronica:
diretta: gli iniettori sono posizionati nella testa cilindri, e sono alimentati da una pompa ad alta pressione, per un'elevata efficienza, ma una maggiore complessità di realizzazione.

indiretta:

- single point: un unico iniettore posto sul collettore di aspirazione.
- multi point: un iniettore per ogni cilindro, posizionato a monte di ognuna delle valvole di aspirazione.

Le prime ECU sviluppate includevano un azionamento meccanico regolato dal pedale dell'acceleratore per gestire la quantità di miscela nel motore. Tuttavia, le ECU più moderne considerano l'acceleratore come un sensore e gestiscono una vasta gamma di parametri, tra cui il tempo di iniezione, la fasatura d'accensione, la temperatura del motore e molto altro.

1.4.2 Di cosa si occupa una ECU

Con lo sviluppo tecnologico in ambito automotive sono state progettate centraline sempre più sofisticate, in grado di gestire elettronicamente non solo l'impianto di alimentazione del veicolo, ma anche altri componenti (trasmissione, cambio automatico, l'impianto di illuminazione...), ed è proprio per questo che a bordo dei veicoli più recenti sono collocate svariate unità di controllo.

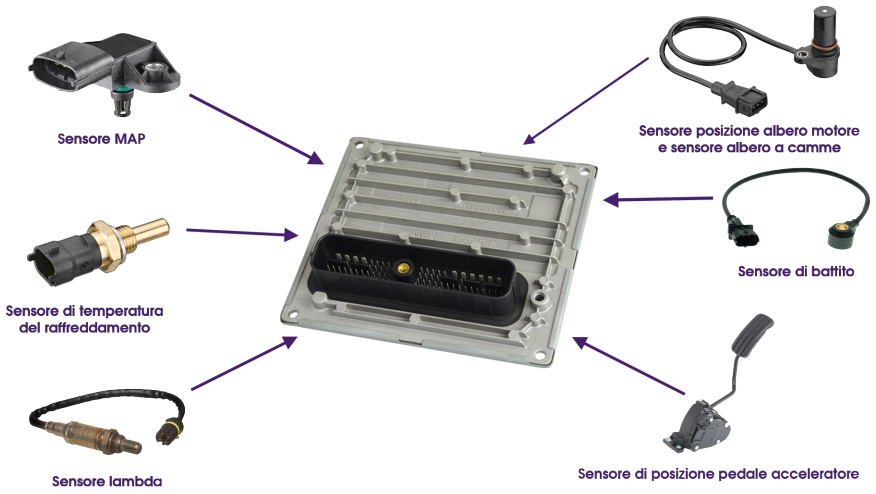

Figura 8: Input in una ECU.

A seconda della tipologia di motore che va gestito, questa unità di controllo determina la corretta quantità di carburante da iniettare per far si che la miscela bruci completamente. Dal momento che ottenere una combustione perfetta è quasi impossibile dato che i fattori in gioco sono svariati, la centralina interviene per sistemare in tempo reale la miscelazione, per minimizzare la formazione di sostanze inquinanti, nonché il consumo di carburante.

1.5 Sistemi embedded

Le centraline di controllo motore rientrano nella categoria dei **sistemi embedded**.
Con la terminologia *"Sistema embedded"* (sistema incorporato), si indica l'insieme costituito da hardware e software (più spesso indicato come firmware) dedicato a occuparsi di scopi specifici.
La maggior parte delle ECU in commercio si basano su sistemi gestiti da Microcontrollori (**MCU**); sono presenti però anche sistemi basati su **PLC** (*programmable logic controller*) oppure **SoC** (*System on Chip*).

1.5.1 MCU

Già a partire dagli anni 70 si sono distinti i primi microcontrollori, o MCU.
L'idea era quella di fornire un sistema hardware completo all'interno del singolo chip.

Figura 9: MCU

Sin dai primi sviluppi, su alcuni modelli era possibile caricare del software adattato allo scopo designato, su un'apposita memoria EEPROM. Essendo questi componenti dotati di poca memoria, e con performance ridotte, i produttori forniscono ambienti di sviluppo specifici per le singole piattaforme, in grado di fornire gli strumenti fondamentali per la creazione di firmware ad hoc. Rispetto ad una CPU general purpose, gli MCU vedono il loro punto di forza nel ridotto consumo energetico (in molti casi inferiore a 1 Watt), sull'efficienza termica e ovviamente sul prezzo per unità che si riduce notevolmente. Nel vasto mondo dei sistemi embedded, le ECU installate a bordo dei veicoli sono di tipo **Hard Real Time**. Sistemi di questo tipo sono caratterizzati dal fatto che devono garantire un prestabilito tempo di risposta, in quanto il mancato rispetto di una *deadline* corrisponderebbe al totale fallimento del sistema, e ad un potenziale rischio per chi ne fa uso. Basti pensare al sistema che gestisce l'ABS (Anti-lock Block System) nelle autovetture, che come si può intuire, se il tempo di reazione di questo sistema non è pressochè istantaneo, perde completamente la sua funzionalità.
Dal punto di vista hardware, i sistemi embedded real-time vengono progettati per essere compatti ed essenziali, con l'obiettivo di garantire massima affidabilità.

Per quanto riguarda il software, vengono utilizzati sistemi operativi **RTOS** (Real-time operating system), che hanno codice più compatto, con un numero ridotto di funzioni.
Dunque per quanto riguarda le MCU, si tende ad integrare più componenti possibili all'interno del chip.
Ciò nonostante non sempre è sufficiente il singolo chip per svolgere il compito designato, infatti oltre alla gestione energetica (evidentemente necessaria), molti dispositivi necessitano di spazio di storage, di sensoristica accessoria, o di esportare alcune porte di I/O in modo semplice da interfacciare.

1.6 Com'è composta una ECU

Come già accennato in precedenza, le ECU rientrano nella categoria dei sistemi embedded: sono costituite da una serie di componenti elettronici, tra cui un microcontrollore, una memoria programmabile, un circuito di ingresso/uscita, un'interfaccia di comunicazione, un alimentatore e altri componenti ausiliari. Il tutto viene sigillato in una scocca di alluminio per proteggere la scheda elettronica da agenti atmosferici o eventuali urti. Tutto il sistema viene gestito da un firmware che viene caricato all'interno della memoria. Il codice che viene sviluppato per queste componenti è di basso livello (ad esempio Assembly), volutamente semplice e rudimentale, per rispettare le caratteristiche dei sistemi hard real time, garantendo così un'elevata affidabilità.

2 Il progetto GPEC4LM

La costante produzione di nuovi modelli di veicoli conferisce al settore automobilistico un carattere dinamico e in continua evoluzione. Per questo motivo, l'azienda si dedica costantemente all'aggiornamento del software dfox, per consentire ai clienti di utilizzare lo strumento Dfox pro con tutti i nuovi modelli di centraline. La realizzazione dei driver per la comunicazione di questi nuovi modelli è quindi una parte fondamentale del lavoro svolto in azienda.
Questo progetto aveva come obiettivo lo sviluppo del driver per la centralina di controllo motore GPEC4LM in BOOT Mode.
Lo scopo di questa attività consiste nel comprendere: il funzionamento di una centralina di controllo motore, la struttura del protocollo di comunicazione utilizzato a bordo dei veicoli e le procedure da seguire per produrre il driver. Le metodologie e le attività che vengono descritte riguardano questo modello in particolare, ma l'approccio è estendibile ad altri modelli. Naturalmente diverse case automobilistiche utilizzano ECU diverse, che differiscono per architettura, firmware e hardware, dunque non esiste un driver univoco, bensì ogni modello deve averne uno ad hoc.
Il progetto si articola in 6 fasi:

- analisi della ECU e del microprocessore,
- preparazione dell'hardware e cablaggio,
- data Logging e estrazione del boot,
- analisi del Boot,
- scrittura del driver in C++,
- test del driver.

Nelle sezioni successive verranno illustrate queste fasi nel dettaglio.

2.1 Analisi della ECU e del microprocessore

In questa fase iniziale è necessario comprendere i meccanismi di funzionamento e le caratteristiche del microprocessore, prima di poter lavorare alla progettazione del driver, in modo tale avere un quadro generale sul sistema. Tutte le specifiche sono riportate nell'apposito manuale reperibile online. Su questo modello di centralina viene montato l'**MPC5674FMR** prodotto dalla *NXP*.

Figura 10: MPC56764FMR

All'interno del manuale sono illustrate tutte le caratteristiche del microprocessore che è montato su questa centralina. Se ne riportano alcune solo a scopo informativo:

- *Power Architecture* 200z7 core a 264 MHz.
- modulo *SIMD* per *DSP* e operazioni a virgola mobile.
- 4 MB di memoria flash con codici per correzione degli errori (ECC)
- 256 KB data RAM con ECC
- 64-channel dual *ETPU2* con 30K *SRAM*
- 64-channel quad analog-to-digital converter (ADC)

Questo modello trova applicazione nel controllo di svariate tipologie di motori diesel e motori di motoveicoli, inoltre supporta la tecnologia **V2X** (*Veichle-to-Everything*) che consente ai veicoli di comunicare con ciò che si trova nei loro dintorni (come ad esempio altri veicoli, elementi di controllo della circolazione e molto altro) rendendo più sicuro il sistema di guida.

2.1.1 Boot Assist Module

Il Boot Assist Module (BAM) è un blocco di memoria per sola lettura contenente il codice che viene eseguito ogni volta che l'MCU viene messo in funzione, o resettato in modalità normale.
Consente due diverse modalità di boot:

- caricamento del boot dalla memoria flash interna.
- caricamento del boot tramite bus seriale.

Il BAM viene eseguito dall'MCU solo dopo che è stato effettuato un MCU reset. Inoltre effettua una lettura della RCHW (*reset configuration half word*) dalla memoria flash che applica poi al microcontrollore. L'**RCHW** definisce le opzioni di boot e deve essere programmata in delle locazioni predefinite nella memoria flash interna, o all'inizio della flash esterna. Dopo la RCHW vanno impostati i 32 bit che indicano l'inizio dell'applicazione. Il BAM consente quattro modalità di funzionamento:

- **Normal Mode**: il BAM viene eseguito immediatamente a seguito della negazione del RESET.
- **Debug Mode**: in modalità di debug, il programma BAM non viene eseguito dopo un RESET dell'MCU.
- **Internal Boot Mode**: in questa modalità è possibile caricare il boot dalla memoria flash interna che contiene tutto il codice e i dati di configurazione.
- **Serial Boot Mode**: in questa modalità va caricato nella SRAM un programma utente tramite interfaccia seriale eSCI o CAN, per poterlo poi eseguire. Può controllare la cancellazione e la scrittura della memoria flash interna o esterna.

Il BAM occupa 16 KB di spazio in memoria, dall'indirizzo 0xFFFF_C000 a 0xFFFF_FFFF. In realtà la dimensione effettiva del codice è inferiore ai 4 KB e inizia a 0xFFFF_F000 ripetendosi ogni 4 KB nello spazio di indirizzamento dedicato al BAM.

2.1.2 I dispositivi di memorizzazione

La centralina in questione è dotata di due unità di memorizzazione.
La prima unità è interna al microcontrollore, ed è una memoria di tipo non volatile.
Si tratta di una memoria flash di 4Mbyte contenente due chip fisici (Flash_A e Flash_B) che sono organizzati in modo tale da fornire uno spazio contiguo per dati e programmi.
Entrambi gli array di memoria sono suddivisi in tre spazi di indirizzamento che sono: low-address space, mid-addres space e high-address space. Ognuna di queste zone è suddivisa in blocchi identificati da un indirizzo di partenza, come verrà mostrato nella figura 17 in fase di scrittura.

	Flash_A array blocks 128-bits wide	**Flash_B array blocks** 128-bits wide
Low address space 256KB (128 bits wide)	8 x16 KB + 2 x 64 KB	
Mid address space 256KB (128 bits wide)	2 x128 KB	
Low address space 256KB (128 bits wide)		1 x 256 KB
Mid address space 256KB (128 bits wide)		1 x 256 KB
High address space 3MB (256 bits wide)	1 x 256 KB	1 x 256 KB

Figura 11: mappa memoria flash

Questa unità supporta inoltre un bus dati per l'operazione di fetch di istruzioni a 64-bit, e il meccanismo DMA.
La lettura può essere effettuata a byte, halfword, word e doubleword, mentre la scrittura solo a word e doubleword.
La seconda unità è una memoria *EEPROM* ausiliaria di 32 Kbyte, esterna al microprocessore, che viene montata sulla scheda madre della centralina.

2.2 Preparazione dell'hardware e cablaggio

Nella seconda fase, si procede alla preparazione della centralina affinché possa effettuare la comunicazione.

2.2.1 Smontaggio della centralina

Inizialmente la centralina si presenta in questo stato:

Figura 12: ecu originale

È quindi necessario rimuovere la scocca in acciaio, che protegge i componenti elettronici da possibili danni causati da agenti esterni (urti, polvere, umidità, calore eccessivo, etc.). Per rimuovere la scocca è sufficiente scaldarla in modo che il collante si ammorbidisca, consentendo la rimozione dei due gusci.
Una volta rimossa la scocca, la centralina si presenta così:

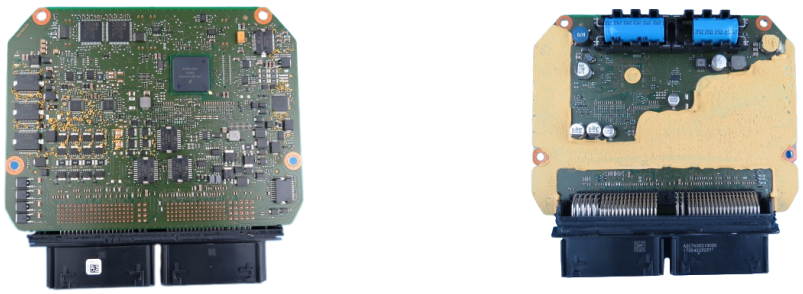

Figura 13: ecu smontata

La rimozione è necessaria per poter saldare un cavo aggiuntivo su un pin della scheda madre, il cui scopo verrà illustrato nella sezione successiva.

2.2.2 Cablaggio della centralina

Una volta che l'ECU è completamente accessibile si procede con il cablaggio per accendere la centralina e far si che comunichi con il data logger (illustrato nella sezione successiva). Il collegamento prevede:

- alimentazione del data logger tramite un alimentatore da banco (13 Volt),
- collegamento del data logger a un computer tramite cavo USB di tipo B,
- collegamento del data logger alla centralina con il cavo multifunzione (Figura 14).

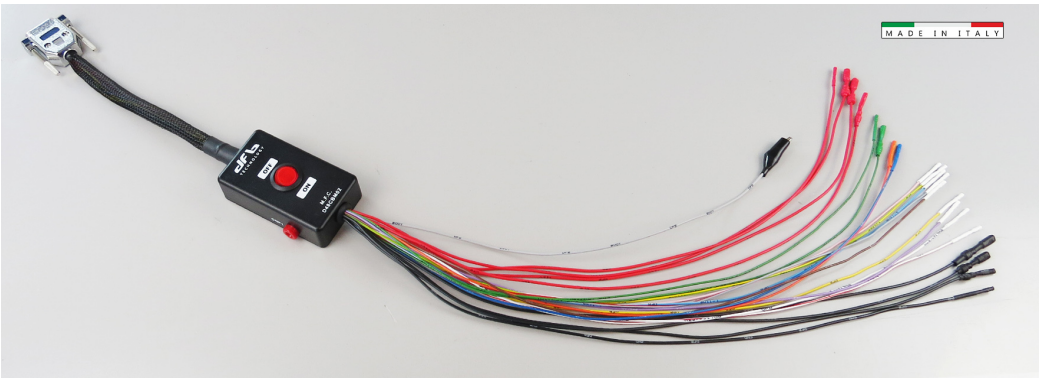

Figura 14: cavo multifunzione

Il cavo multifunzione è costituito da: un interruttore, un connettore seriale per comunicare con il data logger e diversi cavi (identificati da un codice presente sulla guaina) che vengono connessi ai pin dell'interfaccia della centralina. Il collegamento varia per ogni modello, in questo caso va seguito lo schema in Figura 15.

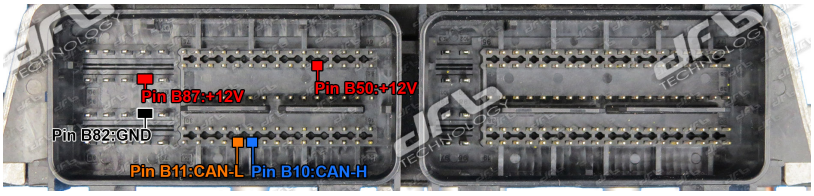

Figura 15: pinout

In aggiunta ai 4 pin appena illustrati, va effettuato un'ulteriore collegamento che prende il nome di **CNF1** (Figura 16). In pratica consiste nell'aggiunta di un cavo che viene saldato su un pin della scheda madre e viene poi connesso al cavo multifunzione. Questo collegamento fornisce al pin sulla scheda un segnale fisso a 3,3 Volt.

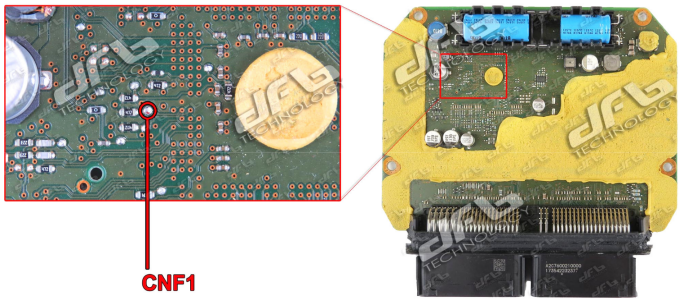

Figura 16: CNF1

Questo ulteriore passaggio è necessario per poter estrarre il boot tramite il data logger, in quanto il segnale fisso pone in condizione di boot la centralina.

2.3 Data Logging e estrazione del Boot

Il Data Logging è il processo di acquisizione, archiviazione e visualizzazione di uno o più set di dati con lo scopo di analizzarne le attività.
Questa operazione ha un ruolo fondamentale nel progetto in quanto le centraline non vengono progettate per interagire con alcun tipo di utente dal momento che sono pensate per interfacciarsi con sensori e eventualmente altri tipi di unità di controllo. I file di log contengono l'acquisizione dei dati che trasmette e riceve la centralina, risultando di fondamentale importanza per due motivi principali: innanzitutto, è possibile estrarre codice a basso livello da questi file (sezione 2.4.1), in aggiunta, se necessario è possibile analizzare questi log, che sono composti da sequenze di byte in esadecimale, per visualizzare ciò che è stato trasmesso.

2.3.1 Il Data Logger

Un data logger è un dispositivo elettronico digitale utile per registrare dati, campionandoli ad intervalli regolari. Dal momento che i dati di interesse per questo progetto riguardano centraline motore, il data logger deve essere compatibile con la comunicazione tramite il protocollo CAN-bus. Questi dispositivi sono facilmente reperibili sul mercato (Figura 17) e sono disponibili in diverse varianti con funzionalità specifiche, ad esempio alcuni possono registrare solo determinati tipi di messaggi CAN, mentre altri possono registrare tutti i messaggi in modo continuo. Per questo progetto è stato utilizzato un dispositivo interamente progettato dall'azienda, sviluppato su misura per effettuare un'analisi completa della maggior parte delle centraline presenti in commercio.

Figura 17: data logger Kvaser

Il data logger per poter funzionare necessita di software ad hoc che provvede all'acquisizione dei dati. In questo caso è stato utilizzato un programma eseguibile sviluppato dall'azienda che si occupa della comunicazione tramite CAN-bus. L'applicativo in questione prende il nome di *"Sniffer"*.

2.3.2 Il protocollo CAN

Il **Controller Area Network** noto anche come (CAN-bus) è uno standard seriale di tipo Multicast, che fu introdotto dalla Bosch nei primi anni 80 per applicazioni automobilistiche. Questo protocollo implementa gli ultimi due livelli della pila **ISO/OSI**, ovvero: il livello di Data Link, e il livello Fisico. Il primo si occupa del filtraggio, della trasmissione dei messaggi e dell'arbitraggio della competizione per la contesa del canale trasmissivo. Il secondo invece specifica il mezzo trasmissivo, il quale deve essere un singolo canale **bidirezionale** che può essere di tipo differenziale o a cavo singolo e terra. Fu progettato per collegare e mettere in comunicazione le diverse unità di controllo presenti a bordo di un veicolo. Tipicamente viene implementato fisicamente seguendo lo standard **rs-485**, ovviamente sono possibili anche altre configurazioni.
Questo protocollo è noto per la sua robustezza in ambienti fortemente disturbati dalle onde elettromagnetiche, e per la *fault tolerance*: infatti, è stato appositamente progettato per garantire il funzionamento anche se uno dei due cavi di trasmissione è interrotto, il che lo rende particolarmente affidabile. Il CAN trasmette dati secondo un modello basato su bit "*dominanti*" (0 logici) e "*recessivi*" (1 logici). Se un nodo trasmette un bit dominante e un altro nodo ne trasmette uno recessivo, tra i due vince il bit dominante, realizzando un AND logico, così da evitare le collisioni. Esistono due versioni del protocollo, costituite da due frame di dimensioni differenti: il **Base Frame** con 11 bit di identificazione (fino a 2048 messaggi diversi), e l'**Extended frame** con 29 bit di identificazione.

2.3.3 EIA RS-485

L'EIA RS-485 è un'interfaccia standard del *livello fisico* di comunicazione della pila **ISO/OSI**, costituita da una connessione seriale a due fili di tipo **half-duplex**. Nella trasmissione di segnali, una delle principali problematiche sono i disturbi di tipo RFI (Radio Frequency Interferences), che vengono però minimizzati grazie alla struttura di questo standard. La connessione seriale rs-485 viene effettuata tramite un collegamento a due o tre fili: uno per il collegamento dati, un'altro per il collegamento dati invertito, e infine il collegamento a massa (0 volt). Il principio di funzionamento è il seguente: i dati vengono trasmessi su uno dei due fili, mentre nell'altro filo viene trasmessa una copia inversa dei dati. I disturbi che si presentano sono praticamente uguali sui due fili, dunque il ricevitore non dovrà fare altro che la differenza tra i due segnali in modo tale da rimuovere quasi completamente il disturbo. Questa tecnica permette di ottenere buoni risultati solamente se su entrambi i cavi è presente lo stesso disturbo, dunque per ottenere una buona approssimazione vengono intrecciati i due cavi (Twisted). Il sistema appena descritto permette di ottenere una gran resistenza alle interferenze di modo comune, e garantisce il funzionamento anche se uno dei due fili viene interrotto. La gestione del segnale è in forma differenziale, cioè la presenza di differenza di tensione tra i due fili costituisce il dato in transito. Una polarità indica il livello logico zero, quella inversa indica il livello 1. La differenza di potenziale deve essere almeno di 0.2 Volt per poter considerare valida l'operazione. Lo standard specifica solo le caratteristiche elettriche del trasmettitore e del ricevitore e permette di ottenere un'elevata velocità di trasmissione di circa 35Mbit/s fino a 10 metri, per questo motivo viene ampiamente utilizzato nel settore automotive.

2.3.4 Estrazione del boot tramite lo sniffer

Una volta completata la configurazione hardware con tutti i collegamenti necessari, si può procedere all'avvio della fase di data logging. Questa operazione, resa possibile dal software "sniffer", consente di ottenere dei file di log contenenti tutte le informazioni relative ai dati che hanno attraversato le linee CAN. Questi file vengono utilizzati principalmente per poter estrarre il boot che viene mandato in esecuzione poi sul microcontrollore, in quanto contiene tutti i comandi del firmware. L'estrazione del boot è possibile grazie al collegamento CNF1, visto al capitolo 2.2.2, che impone alla centralina di inviare il boot tramite le linee CAN. Di norma la ECU andrebbe semplicemente a caricare il boot dalla memoria secondaria alla memoria primaria per poterlo mandare in esecuzione sul microprocessore, dunque non sarebbe visibile al data logger in quanto non attraverserebbe le linee CAN. Oltre

a poter estrarre il boot, i file di log saranno utili per visualizzare le sequenze di comandi in esadecimale, che la centralina invia per effettuare le varie operazioni di lettura e di scrittura. Il software si presenta con la seguente interfaccia:

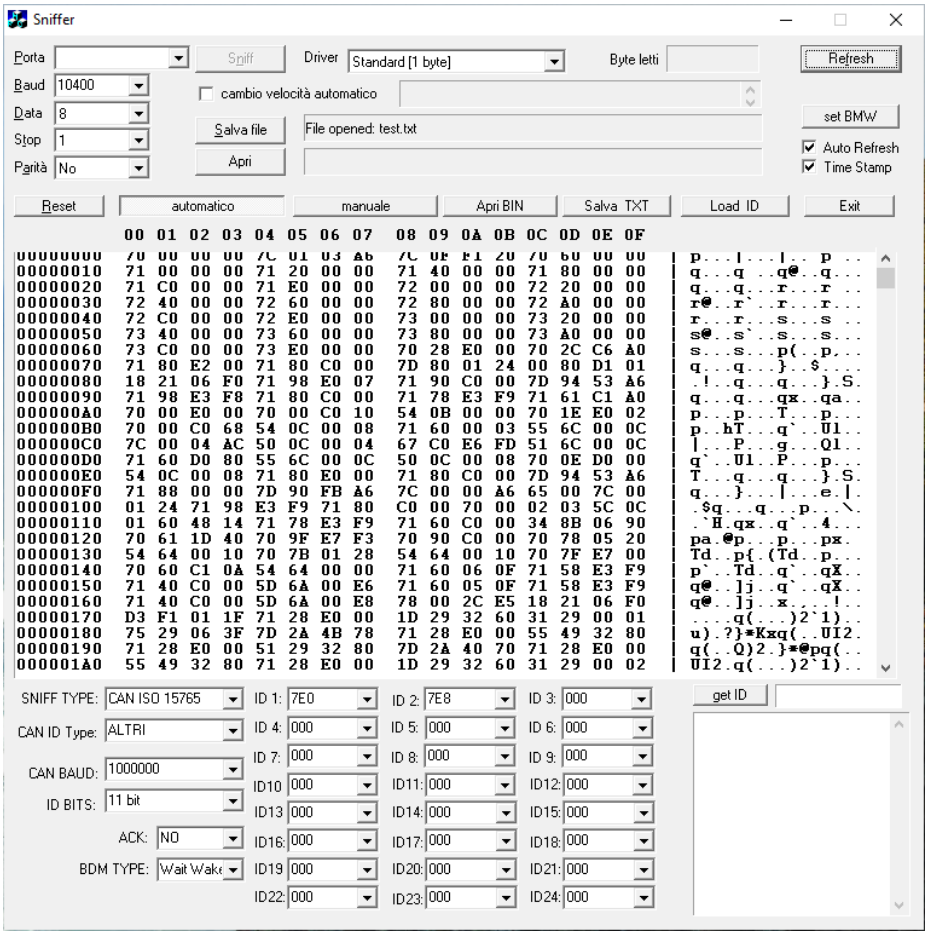

Figura 18: sniffer

Prima di poter avviare la comunicazione vanno impostati alcuni parametri, tra cui:

1) Il **baud rate** che rappresenta in un sistema di trasmissione digitale, il numero di simboli che vengono trasmessi in un secondo.

2) Il protocollo da utilizzare; in questo caso viene utilizzato il **CAN ISO 15765**, una versione del protocollo specifica per l'utilizzo sui veicoli.

3) Gli identificatori dei nodi che interessano la trasmissione, ad esempio 7E0 per il data logger e 7E8 per la ECU.

Il software si occuperà di filtrare la trasmissione, memorizzando nel file di log solamente i dati che corrispondono agli ID selezionati.
Fatto questo è possibile avviare la procedura di comunicazione tra data logger e centralina. Inizialmente il microcontrollore carica il boot in memoria centrale tramite il BAM, visto nella sezione 2.1.1, per poterlo mandare in esecuzione. Per catturare interamente il boot, il programma viene eseguito per diversi secondi in modalità di ascolto, prima di interromperne l'esecuzione. La centralina genera comandi a intervalli dell'ordine dei millisecondi, per cui è sufficiente un breve periodo di tempo per ottenere il file di log. Una volta interrotta la comunicazione è possibile salvare i dati estratti in formato *.txt*.
In seguito viene mostrato un breve estratto da un file di log:

```
Attesa :000003 ms  07 E0 00 00 03 03 00 00 03 03
Attesa :000003 ms  07 E0 00 00 03 03 00 00 01 00
Attesa :000007 ms  07 E0 00 00 00 5D 00 00 00 61
Attesa :000004 ms  07 E0 00 00 00 5F 00 00 00 5E
Attesa :000004 ms  07 E8 4C 00 00 5F 00 00 00 5E
Attesa :000004 ms  07 E0 4C 40 00 40 C0 80 00 30
Attesa :000005 ms  07 E8 4C 40 00 40 C0 80 00 30
Attesa :000005 ms  07 E0 4B 00 02 F8 00 80 04 00
Attesa :000005 ms  07 E8 8B 80 BA BE 00 00 00 FF
Attesa :000005 ms  07 E0 4B 00 02 FC 00 80 04 00
Attesa :000005 ms  07 E8 8B 00 02 FC 00 80 04 00
Attesa :000005 ms  07 E0 5B 00 00 00 00 00 00 00
```

Figura 19: file di log

Come si può osservare dall'immagine, oltre ai dati veri e propri che attraversano le linee CAN, vengono aggiunte dallo sniffer alcune informazioni utili come:

- il tempo (in millisecondi) che trascorre tra un Data frame e il successivo,
- l'identificatore del nodo che ha trasmesso quel preciso frame di dati (7E0/7E8).

2.3.5 Pulizia della lettura dello sniffer

Dal momento che vogliamo estrarre il boot è necessario rimuovere dal file di log tutte le informazioni aggiuntive, in modo tale da ottenere un file contenente solamente i byte che il microcontrollore ha trasmesso. Per farlo è sufficiente aprire il file in un qualsiasi editor di testo ed effettuare le modifiche.
Nell'immagine viene mostrato un esempio di come deve risultare il file una volta pulito:

```
70 00 00 00 7C 01 03 A6
70 00 00 00 7C 01 03 A6
7C 0F F1 20 70 60 00 00
7C 0F F1 20 70 60 00 00
71 00 00 00 71 20 00 00
71 00 00 00 71 20 00 00
71 40 00 00 71 80 00 00
71 40 00 00 71 80 00 00
71 C0 00 00 71 E0 00 00
71 C0 00 00 71 E0 00 00
72 00 00 00 72 20 00 00
72 00 00 00 72 20 00 00
72 40 00 00 72 60 00 00
72 40 00 00 72 60 00 00
```

Figura 20: file di log ripulito

Per ragioni pratiche, nella foto è riportato solo un estratto di poche righe del file di log, con il solo scopo di mostrare come risulta il file pulito, considerato che quest'ultimo contiene diverse centinaia di righe. Dopo aver pulito il file, è necessario convertirlo in formato binario (.bin) al fine di renderlo utilizzabile nella successiva fase.

2.4 Analisi del boot

La presente fase del progetto prevede l'analisi del boot della centralina elettronica, al fine di identificare le funzioni che sono impiegate per la lettura e la scrittura delle unità di memorizzazione. In questa tesi, il focus principale è sulla scrittura del driver in linguaggio C++, pertanto non verrà fornita un'illustrazione dettagliata di questa fase.
Il boot da esaminare è implementato in linguaggio assembly, il quale viene generato mediante un software apposito, noto come IDA, a partire dal codice macchina che è stato estratto nella fase di Data logging.

2.4.1 IDA: The Interactive Disassembler

L'IDA (https://hex-rays.com/ida-pro/) è un software che rientra nella categoria dei **disassemblatori**, in pratica si occupa di generare codice sorgente in linguaggio Assembly a partire da codice macchina eseguibile.
Questo applicativo verrà utilizzato in questa fase per convertire il file binario (che contiene il boot), in codice Assembly. Il codice che verrà generato, contiene una serie di istruzioni opportunamente organizzate all'interno di diverse subroutine, che coordinano le operazioni del microcontrollore. Solo alcune di queste sono di interesse per questo progetto, nello specifico quelle che riguardano la lettura e la scrittura dei dispositivi di memorizzazione.
Nella sezione successiva sarà fornita una breve descrizione di una subroutine, al fine di aiutare il lettore a comprendere meglio l'argomento trattato.

2.4.2 Illustrazione comando per la scrittura della memoria flash

In questo capitolo viene illustrata una breve sezione della subroutine che si occupa della scrittura della memoria flash, per dare al lettore un'idea di come venga progettato il firmware della centralina.

```
lis       r9, 0x400032C0@ha
add16i    r9, r9, 0x400032C0@l
lbz       r9, (0x400032C0 - 0x400032C0)(r9)
clrlwi    r9, r9, 24
cmpl16i   r9, 0x47 #'G' // verifica che il comando ricevuto è 0x47
mcrf      cr7, cr0
mfcr      r9
slwi      r9, r9, 28
mtcrf     0x80, r9
bne       loc_2FE0
bl        sub_1916 // subroutine per iniziare la scrittura
b         loc_3260
```

In questa prima fase vengono fatte delle semplici operazioni sul registro r9 in cui è presente il byte contenente il comando che è stato richiesto alla centralina. Come si può notare viene effettuato un confronto tra il contenuto del registro r9 e il byte esadecimale 0x47 che corrisponde al comando per la scrittura della memoria flash. Ovviamente la struttura è ben più complessa e si dirama in altre subroutine che si occupano di diverse operazioni.

Nella seguente immagine è illustrata la gerarchia con cui le subroutine della scrittura sono organizzate.

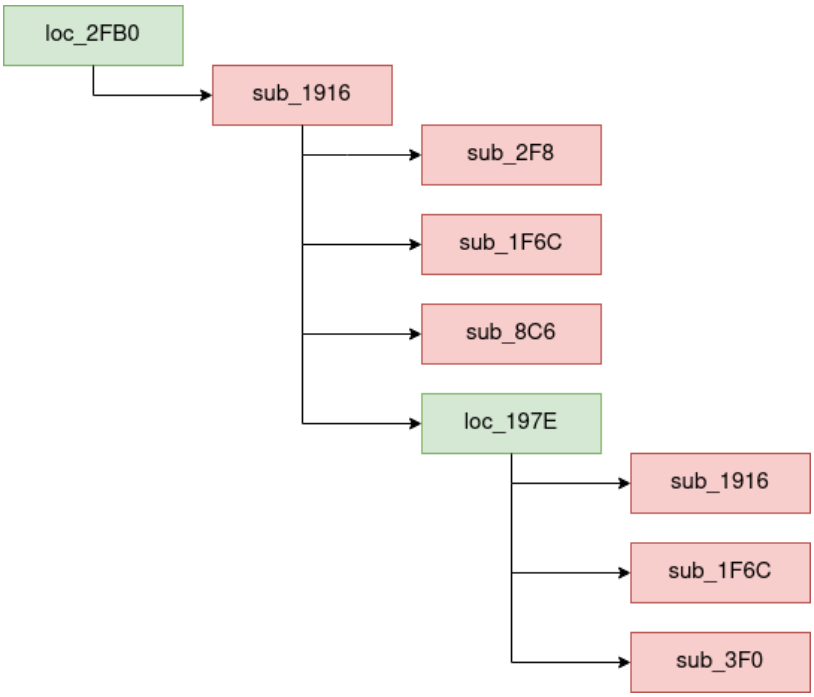

Figura 21: gerarchia subroutine

Si possono distinguere due differenti tipi di entità, che sono rispettivamente subroutine e location.
Per capire bene la differenza tra questi due elementi è necessario osservare che l'intero boot viene mappato in una zona di memoria i cui indirizzi sono contigui.
Quando si parla di subroutine si intende una funzione vera e propria, con un indirizzo di inizio e uno di fine. Le location rappresentano invece dei semplici indirizzi di memoria, per cui quando si effettua ad esempio l'operazione **b loc_3260** si sta semplicemente effettuando un salto all'indirizzo indicato (00003260).

Alcune delle subroutine in figura si occupano della lettura del buffer contenente il comando che riceve la centralina, altre invece si occupano di effettuare la scrittura vera e propria, altre ancora di inviare la risposta.

L'analisi di queste funzioni e dei file di log estratti ci permettono di avere un quadro generale riguardo le funzioni del firmware che servono per le varie operazioni di lettura e scrittura.

2.5 Scrittura del driver in C++

In questo capitolo viene illustrato il processo di sviluppo delle funzioni per effettuare la lettura e la scrittura della memoria flash interna, e della memoria EEPROM esterna all'MCU. In pratica tramite il linguaggio ad alto livello C++ si vanno a replicare i comandi che la centralina normalmente impiega per effettuare queste operazioni. Verranno composte all'interno di alcune funzioni le varie sequenze di byte che tramite il dfox pro vengono inviate al microcontrollore per essere interpretate ed eseguite.
Essenzialmente si va a ricreare ciò che si è potuto osservare dai file ottenuti nella fase di data logging, con la differenza che i frame di dati vengono modellati in base alle esigenze. Ecco un esempio che ci aiuterà a comprendere meglio di cosa si tratta: supponiamo che un cliente che possiede il kit voglia apportare alcune modifiche ai byte della mappa della centralina, che è salvata in memoria. In tal caso, sarà necessario estrarre l'intero contenuto della memoria (effettuando una lettura), apportare le modifiche desiderate, e infine riscrivere i dati modificati sulla centralina (effettuando una scrittura).
L'obiettivo di questa fase consiste nel riuscire a controllare la centralina tramite i comandi che vengono inviati dal software, attraverso il CAN bus.
La difficoltà maggiore in questa fase sta nel comporre correttamente le sequenze di byte, in quanto un errato invio di un comando può anche compromettere il firmware della centralina stessa.
Il codice prodotto in questa fase costituirà il driver vero e proprio che verrà introdotto nel software dfox, per poter essere utilizzato dai clienti per operare su questo modello specifico di centralina.

2.5.1 Lettura memoria flash interna

La prima funzione che viene illustrata implementa la procedura e i comandi per effettuare la lettura della memoria flash interna, producendo in uscita un file binario con il contenuto della memoria. In pratica attraverso l'invio del comando che verrà illustrato a breve, impostato con opportuni parametri, è possibile imporre alla centralina di trasmettere l'intero contenuto della memoria flash, che tramite il software dfox verrà scritto su un file binario. In seguito viene mostrato il codice necessario per effettuare questa procedura:

```
IniTab1 = 0x0000;   EndTab1 = 0x400000;

BYTE ADD[] = {0x4B,0x00,0x00,0x00,0x00,0x80,0x02,0x00};
BYTE READ[] = {0x23,0x00,0x00,0x41,0x00,0x00,0x00,0x00};

for(i=IniTab1; i<EndTab1;i+=0x200) {
  ADD[2] = BYTE(i>>16);
  ADD[4] = BYTE(i>>8);
  e = SendComCAN(ADD, sizeof(ADD),3);
  if(e < 0){CommandMsg("ERROR ADD", -1); return false;}
  if(ReadBuf[21]==0x80 && ReadBuf[22]==0xBA && ReadBuf[23]==0xBE)
  {  for(j=0; j<0x200; j++) M[y++] = ReadBuffer[27]; }
  else // se la zona non è vuota
  {
    e = SendComCAN(READ, sizeof(READ),7);
    if(e<0 && e!=-4){CommandMsg("ERROR READ", -1); return false;}
    Pack = (ReadBuf[15]<<8)+ReadBuf[16];
    if(Pack<0x20A) // verifica correttezza pacchetto inviato
    {
      e = SendComCAN(ADD, sizeof(ADD),3);
      if(e<0 && e!=-4){CommandMsg("ERROR ADD", -1); return false;}
      e = SendComCAN(READ, sizeof(READ),7); // rimando la lettura
      Pack = (ReadBuf[15]<<8)+ReadBuf[16];
      if(Pack<0x20A) {CommandMsg("ERROR READ", -1); return false;}
    }
      PulisciBuffer_Tricore_BOOT();
    for(j=0x8; j<0x208; j++) M[y++] = ReadBuf[j];
  }
 }
```

Schematicamente i passaggi per l'operazione di lettura possono essere rappresentati con il seguente diagramma:

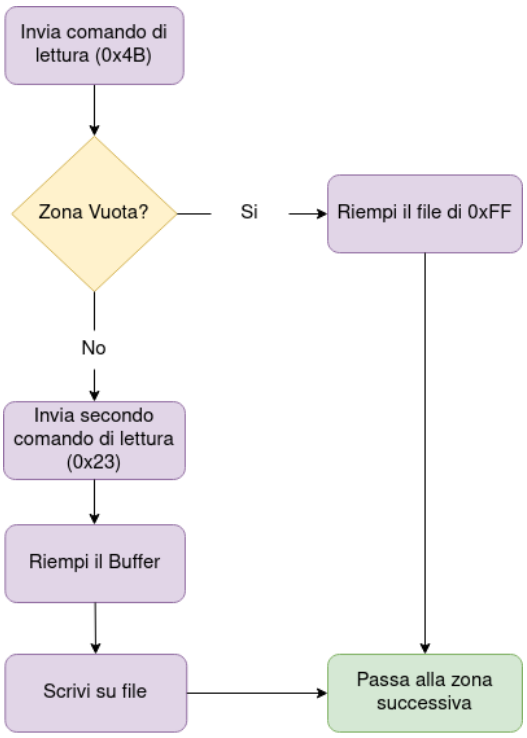

Figura 22: lettura flash

Per questioni pratiche e di ottimizzazione del codice, i byte letti dalla memoria flash vengono inseriti in un array temporaneo che simula la memoria (indicato con M), che viene poi scritto in un file su disco.
Come supporto si utilizza un altro buffer temporaneo (*ReadBuffer*) che facilita la gestione della lettura in quanto quest'ultima viene effettuata a blocchi.
I comandi che verranno utilizzati per questa operazione sono: 0x4B e 0x23.
Questi comandi opportunamente impostati verranno inviati alla centralina tramite una funzione (*sendComCAN*) che permette la trasmissione tramite CAN-bus.
Per prima cosa vengono definiti gli estremi di lettura della memoria:

- IniTab1 = 0x0000
- EndTab1 = $0x400000_{16}$ che corrisponde a 4194304 bytes (4kbyte).

Il ciclo for più esterno permette l'attraversamento di tutta memoria flash.
Si noti che la lettura viene effettuata a blocchi per questioni dovute al firmware del microprocessore.

I blocchi sono di dimensione $0x200_{16}$, ovvero 512 byte.
Ad ogni iterazione viene inviato sempre lo stesso comando a cui viene applicato uno shift logico a destra, che modifica il secondo, terzo e quarto byte, per poter di volta in volta accedere alla zona successiva.
Una volta inviato il primo comando ADD, si verifica l'eventuale presenza di errori tramite il valore di ritorno, e si attende la risposta da parte della ECU.
Un esempio di invio è il seguente: 07 E0 **4B** **00** **02 00 00** **80** **02 00**
Possiamo distinguere i seguenti 4 elementi:

- Identificatore del nodo responsabile dell'invio del frame di dati (in questo caso il software dfox): 07 E0.
- ID del comando per la lettura: 4B.
- Parametro del comando n.1, rappresenta l'indirizzo di partenza della zona che va letta: 02 00 00.
- Parametro del comando n.2, indica il numero di byte da leggere: 02 00.

A meno di eventuali errori le risposte che interessano per la lettura sono di due tipi.

- **07 E8 8B 80 BA BE 00 00 00 FF**.
- **07 E8 8B 00 02 00 00 80 02 00**.

Nel primo caso la centralina ci comunica che quella particolare zona di cui abbiamo richiesto la lettura è vuota, ovvero è costituita solamente da byte uguali a 0xFF. Per una questione di ottimizzazione, il microprocessore si limita a comunicare questa assenza di dati, evitando così di inviare 512 byte uguali a 0xFF. Dal momento che il driver deve produrre in uscita un file con il contenuto della memoria, non è possibile omettere l'inserimento di questo blocco, per cui bisogna provvedere all'inserimento di questi 512 byte "nulli". Dunque come si può osservare, se la condizione di if è verificata, un successivo ciclo for si occupa di inserire nell'array ReadBuf tutti i byte uguali a 0xFF.
Nel secondo caso la risposta dell'MCU consiste in una copia del comando da noi inviato. Questo sta ad indicare che la zona non è vuota. A seguito di questa risposta inizierà la trasmissione dei dati presenti nella memoria a gruppi di 8 byte alla volta. La gestione di questo caso viene effettuata all'interno del ramo *else* che prevede innanzitutto l'invio di un nuovo comando identificato da *READ*. Questo ulteriore comando indica al microprocessore che si è pronti per la ricezione dei dati. Una volta inviato il comando vengono fatti alcuni controlli per verificare la lunghezza del pacchetto ricevuto.

Viene impostato un secondo tentativo di invio in caso di errore.
Una volta che la comunicazione è andata a buon fine bisogna trasferire le informazioni lette sull'array M che simula la memoria flash, per poi scrivere il tutto sul file binario di output. Il metodo *PulisciBuffer_Tricore_Boot()* si occupa di ripulire il buffer di lettura dagli ID di comunicazione del protocollo CAN-bus in modo tale da avere nel buffer solamente il payload.
L'ultimo ciclo for si occupa di copiare la zona appena letta all'interno dell'array che mappa la memoria M.

Come ultima fase vengono effettuate le operazioni per trasferire il contenuto della memoria che è stato letto, nel file binario.
Vengono distinti due casi in quanto nel software, è possibile effettuare le letture separate delle varie unità di memorizzazione presenti sulla centralina, oppure è possibile avviare una procedura che legge in modo sequenziale tutte le unità. Il metodo *WriteFile* si occupa di gestire correttamente la scrittura.

```
if(m_fileName == "" || m_fileName.IsEmpty())
{CommandMsg(lingua.GetAt(28), 0); return false;}
if(UseA)  F = fopen(m_fileName+"_IntFl", "wb");// lettura completa
if(!UseA) F = fopen(m_fileName, "wb"); // lettura singola
if(WriteFile(M, EndTab1-IniTab1, 1, F) != 1)
{
  CommandMsg(lingua.GetAt(27), -10); // errore scrittura file
  fclose(F);
  return false;
  }
fclose(F);
```

2.5.2 Lettura memoria EEPROM

In questo capitolo viene illustrata la procedura analoga a quella vista nel capitolo precedente, per la lettura della memoria EEPROM esterna.

```
IniTab1 = 0x0000;
EndTab1 = 0x8000;
y=0;

BYTE READ[] = {0x30,0x00,0x00,0x00,0x00,0x00,0x01,0x00,
        0x30,0x00,0x00,0x00,0x00,0x00,0x01,0x00};

for(i=IniTab1; i<EndTab1;i+=0x100)
{
  READ[2] = BYTE(i>>16);
  READ[4] = BYTE(i>>8);
  READ[10] = BYTE(i>>16);
  READ[11] = BYTE(i>>8);

  e = SendComCAN(READ, sizeof(READ),7);
  if(e<0 && e!=-4){CommandMsg("ERROR READ", -1); return false;}
  Pack = (ReadBuffer[23]<<8)+ReadBuffer[24];
  if(Pack<0x102)
  {
    e = SendComCAN(READ, sizeof(READ),7);
    Pack = (ReadBuffer[23]<<8)+ReadBuffer[24];
    if(Pack<0x102) {CommandMsg("ERROR READ", -1); return false;}
  }
  PulisciBuffer_Tricore_BOOT();
  for(j=0x0; j<0x100; j++) M[y++] = ReadBuffer[j];
}
```

Ecco una rappresentazione schematica dell'operazione di lettura della memoria EEPROM:

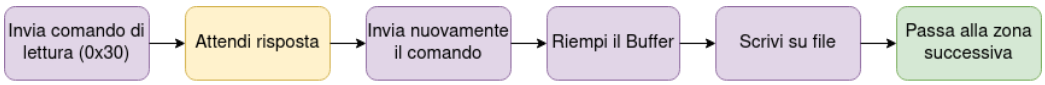

Figura 23: lettura EEPROM

L'implementazione di questo metodo è molto simile a quella vista nel capitolo precedente. In questo caso il comando per effettuare la lettura è solo uno, ed è contenuto nell'array di byte READ. Il comando è sempre costituito da 8 distinti byte, però la procedura di lettura prevede che venga inviato due volte. Per comodità quindi è stato inserito nell'array *READ*

doppio. Quando verrà inviato tramite la funzione *sendComCan*, il parametro 7 permetterà di inviare solo i primi 8 byte e i restanti 8 solo dopo aver ricevuto la risposta.
Viene effettuata una lettura a blocchi, di dimensione $0x100_{16}$ (256 byte ognuno). A differenza della procedura attuata nel caso della flash interna, essendo la memoria EEPROM di dimensioni molto più ridotte ($0x8000_{16}$ cioè 32 kbyte), non viene fatta distinzione tra zone vuote e zone contenenti dati.
Il ciclo for si occupa dell'invio del comando di lettura ogni 256 byte, fino ad esaurire la memoria.
All'interno del ciclo vengono effettuati appositi shift logici a destra per poter modificare l'array contenente il comando da inviare, per far si che punti alle varie zone della memoria, una dopo l'altra.
In questo caso viene inviato il comando, si attende la risposta e si invia nuovamente il comando. Dopodiché inizierà la trasmissione dei dati presenti nella memoria EEPROM a gruppi di 8 byte alla volta, e il Buffer viene riempito.
Viene mostrato in figura un esempio di questa procedura:

```
07 E0 30 00 00 00 00 00 01 00
07 E8 86 00 00 00 00 00 01 00
07 E0 30 00 00 00 00 00 01 00
07 E8 FF FF FF FF FF FF FF FF
07 E8 FF FF FF FF FF FF FF FF
07 E8 FF FF FF FF FF FF FF FF
07 E8 FF 00 FF FF FF FF FF FF
07 E8 FF FF FF FF FF FF FF FF
07 E8 FF FF FF FF 03 FF 03 FF
07 E8 FF FF FF FF 03 FF 03 FF
07 E8 FF FF 02 00 01 00 02 00
07 E8 01 00 00 BF 00 00 00 00
07 E8 FF FF FF FF FF FF FF 17
```

Figura 24: lettura flash

Dove si possono notare in giallo l'indirizzo della zona che sarà 0x0000, 0x0100 e così via fino alla fine della memoria, e in verde il numero di byte da leggere. Dopo l'invio dei due comandi la centralina inizia a trasmettere pacchetti di 8 byte uno dopo l'altro.
Infine come per la memoria flash, il contenuto del buffer viene scritto in un file binario.

2.5.3 Scrittura memoria Flash interna

In questa sezione viene illustrata la procedura per effettuare la scrittura di un file binario, nella memoria flash interna al microprocessore.
I vari passaggi che verranno effettuati si possono rappresentare con il seguente diagramma di flusso:

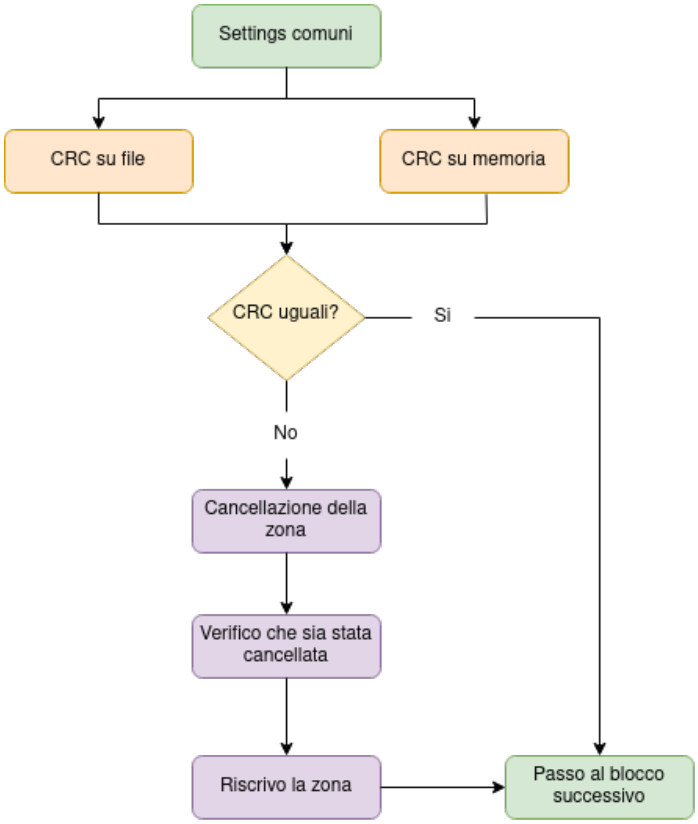

Figura 25: diagramma scrittura memoria flash

Come si può notare dall'immagine la procedura comprende diverse fasi che verranno analizzate singolarmente.
Anche l'operazione di scrittura viene effettuata a blocchi di $0x200_{16}$ (ovvero 512 byte ognuno) per questioni di praticità, in quanto la memoria fisica è suddivisa in zone.
Per prima cosa vengono effettuati dei controlli per verificare la dimensione del file che l'utente ha intenzione di scrivere, che deve corrispondere alla dimensione del file originariamente presente nella flash.
Dopodichè si procede con il calcolo del CRC.

2.5.3.1 Il calcolo del CRC

Il firmware del microprocessore oltre ai vari comandi, ha una funzione per il calcolo del CRC su blocchi di memoria. Il CRC consiste in una stringa testuale che viene prodotta utilizzando un algoritmo, a partire dal contenuto stesso del blocco della memoria.
Lo scopo di questa procedura consiste nel poter effettuare un confronto tra due CRC.

Per prima cosa viene calcolato il CRC di un blocco del file (che l'utente ha intenzione di scrivere all'interno della memoria), tramite una funzione integrata nel driver. Dopodiché tramite il comando **0x4A** la centralina effettua il calcolo del CRC sui dati che sono presenti nel corrispondente blocco della memoria flash.
Dato che l'algoritmo utilizzato per il calcolo è lo stesso, si evince che se i due CRC calcolati risultano **uguali**, allora i dati presenti nel file da scrivere (in un particolare blocco) e quelli presenti nel corrispettivo blocco della memoria interna, sono gli stessi, dunque non è necessario andare a effettuare la scrittura.
Analizziamo ora la procedura per il calcolo dei CRC:

```
// comando per calcolo del CRC
BYTE CRC[] = {0x4A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00};
// buffer che conterrà i dati da scrivere in memoria
BYTE BUFFER[0x200];

// doubleword contenente la parte alta/bassa (4 bytes)
// del CRC calcolato su file
DWORD CRC_Alta;
DWORD CRC_Bassa;

// indirizzi zone memoria flash
DWORD Flash_Address[]= {0x00,0x4000, ... ,0x380000,0x400000};

// verifica della lunghezza del file da scrivere
if(writeFileLen < 0x408000 || writeFileLen > 0x408000 && !UseA)
{
  CommandMsg("ERROR : wrong file size", -1);
  return -1;
}
```

```
// calcolo il crc sul file da scrivere nel micro
for (i = 0; i < 20; i++)
{
    // CALCOLO DEL CRC DELLA ZONA
  for (j = Flash_Address[i]; j < Flash_Address[i + 1]; j += 0x800)
  {

    CRC_Alta = CRC_Calc(writeFile + j, 0x800, 0); // CRC su file
    CRC_Bassa = CRC_Calc(writeFile + j, 0x800, 1); // CRC su file

    CRC[2] = (BYTE)(j >> 16);
    CRC[3] = (BYTE)(j >> 8);
    CRC[4] = (BYTE)j;

    // comando per ottenere il crc del blocco della flash
    e = SendComCAN(CRC, sizeof(CRC), 3);
    if (e < 0 && e!= -4) CommandMsg("ERROR data Micro 1", -1);

        // attesa altra risposta senza invio comando
    e = SendComCAN(CRC, sizeof(CRC), 5);
    if (e < 0 && e!= -4) CommandMsg("ERROR data Micro 2", -1);

    e = PulisciBuffer_Tricore_BOOT();
    if (e < 0) { CommandMsg("ERROR data Micro 3", -1); return -1;}

    DWORD Alta = ReadBuffer[3] + (ReadBuffer[2] << 8) + // CRC su flash
                 (ReadBuffer[1] << 16) + (ReadBuffer[0] << 24);
    DWORD Bassa = ReadBuffer[7] + (ReadBuffer[6] << 8) + // CRC su flash
                 (ReadBuffer[5] << 16) + (ReadBuffer[4] << 24);

    if (Alta != CRC_Alta && Bassa != CRC_Bassa) // confronto i CRC
    {
      uguali = false;
      break;
    }
    }
...
```

2.5.3.2 Cancellazione della zona

Una volta confrontati i CRC, se sono diversi è necessario riscrivere il corrispettivo blocco di memoria, però è prima necessario cancellare la zona.

Per effettuare la cancellazione viene invocato un metodo apposito che se ne occupa.
Questa operazione infatti richiede che vengano sbloccati alcuni registri tramite dei codici di sblocco che possono essere reperiti nel manuale del microprocessore.
La procedura prevede che vengano sbloccate tutte le zone di memoria ad ogni cancellazione, per poter poi effettuare la scrittura.
La memoria flash interna è così suddivisa:

Table 11-1. Memory Map

Offset from FLASH_BASE (0x0000_0000)	Use	Flash A Block	Flash B Block	Partition	Flash A Block Size	Flash B Block Size	Data Width
0x0000_0000	Low-address space (Flash A)	L0	—	1	16K	—	128
0x0000_4000		L1	—		16K	—	128
0x0000_8000		L2	—		16K	—	128
0x0000_C000		L3	—		16K	—	128
0x0001_0000		L4	—	2	16K	—	128
0x0001_4000		L5	—		16K	—	128
0x0001_8000		L6	—		16K	—	128
0x0001_C000		L7	—		16K	—	128
0x0002_0000		L8	—	3	64K	—	128
0x0003_0000		L9	—		64K	—	128
0x0004_0000	Mid-address space (Flash A)	M0	—	4	128K	—	128
0x0006_0000		M1	—		128K	—	128
0x0008_0000	Low-address space (Flash B)	—	L0	5	—	256K	128
0x000C_0000	Mid-address space (Flash B)	—	M0		—	256K	128
0x0010_0000	High-address space[1]	H0	H0	6	256K	256K	256
0x0018_0000		H1	H1		256K	256K	256
0x0020_0000		H2	H2	7	256K	256K	256
0x0028_0000		H3	H3		256K	256K	256
0x0030_0000		H4	H4	8	256K	256K	256
0x0038_0000		H5	H5		256K	256K	256
0x0040_0000	Reserved						
0x00EF_C000	Shadow Block (Flash B space) (see Table 11-2)	—	S0	All[2]	—	16K	128
0x00FF_C000	Shadow Block (Flash A space) (see Table 11-2)	S0	—		16K	—	128
0x0100_C000	Reserved						

Figura 26: flash memory

I comandi che vengono utilizzati per lo sblocco delle varie zone sono:

- **0x49** per l'accesso ai vari registri.
- **0x48** per la cancellazione.

Viene replicata in C++ la struttura della memoria, suddivisa nelle varie zone, in modo tale che al variare del parametro *Address* che rappresenta l'indirizzo della zona che si desidera cancellare, vengano opportunamente impostati i parametri del comando per la cancellazione attraverso uno *switch-case*.

```
DWORD BLOCK, REG;

switch(Address)
{
  // LOW address Flash A
  case      0x0: BLOCK = 0x00001; REG=0x8004; break; // L0
  case   0x4000: BLOCK = 0x00002; REG=0x8004; break; // L1
  case   0x8000: BLOCK = 0x00004; REG=0x8004; break; // L2
  case   0xC000: BLOCK = 0x00008; REG=0x8004; break; // L3
  case  0x10000: BLOCK = 0x00010; REG=0x8004; break; // L4
  case  0x14000: BLOCK = 0x00020; REG=0x8004; break; // L5
  case  0x18000: BLOCK = 0x00040; REG=0x8004; break; // L6
  case  0x1C000: BLOCK = 0x00080; REG=0x8004; break; // L7
  case  0x20000: BLOCK = 0x00100; REG=0x8004; break; // L8
  case  0x30000: BLOCK = 0x00200; REG=0x8004; break; // L9
  // MID address Flash A
  case  0x40000: BLOCK = 0x10000; REG=0x8004; break; // M0
  case  0x60000: BLOCK = 0x20000; REG=0x8004; break; // M1
  // LOW address Flash B
  case  0x80000: BLOCK = 0x00001; REG=0xC004; break; // L0
  // MID address Flash B
  case  0xC0000: BLOCK = 0x10000; REG=0xC004; break; // M0
  // HIGH address Flash A/B
  case 0x100000: BLOCK = 0x00001; REG=0x8008; break; // H0
  case 0x180000: BLOCK = 0x00002; REG=0x8008; break; // H1
  case 0x200000: BLOCK = 0x00004; REG=0x8008; break; // H2
  case 0x280000: BLOCK = 0x00008; REG=0x8008; break; // H3
  case 0x300000: BLOCK = 0x00010; REG=0x8008; break; // H4
  case 0x380000: BLOCK = 0x00020; REG=0x8008; break; // H5
}
```

Una volta definiti i vari casi si procede con lo sblocco dei vari registri.

```
BYTE UNLOCK[] = {0x49,0x00,0x00,0x00,0xC3,0xF8,0x80,0x04,
0xA1,0xA1,0x11,0x11,0x00,0x00,0x00,0x00};
// FLASH-A
// Low-/Mid-address space block locking register
 e = SendComCAN(UNLOCK, sizeof(UNLOCK),3);
if(e < 0){CommandMsg("ERROR UNLOCK", -1); return -1;}

// Secondary low-/mid-address space block locking register
UNLOCK[6]=0x80;UNLOCK[7]=0x0C;UNLOCK[8]=0xC3;
UNLOCK[9]=0xC3;UNLOCK[10]=0x33;UNLOCK[11]=0x33;
e = SendComCAN(UNLOCK, sizeof(UNLOCK),3);
if(e < 0){CommandMsg("ERROR UNLOCK", -1); return -1;}

// High-address space block locking register
UNLOCK[6]=0x80;UNLOCK[7]=0x08;UNLOCK[8]=0xB2;
UNLOCK[9]=0xB2;UNLOCK[10]=0x22;UNLOCK[11]=0x22;
e = SendComCAN(UNLOCK, sizeof(UNLOCK),3);
if(e < 0){CommandMsg("ERROR UNLOCK", -1); return -1;}

// FLASH-B
// Low/Mid-address space block locking register
UNLOCK[6]=0xC0;UNLOCK[7]=0x04;UNLOCK[8]=0xA1;
UNLOCK[9]=0xA1;UNLOCK[10]=0x11;UNLOCK[11]=0x11;
e = SendComCAN(UNLOCK, sizeof(UNLOCK),3);
if(e < 0){CommandMsg("ERROR UNLOCK", -1); return -1;}

// Secondary low/mid-address space block locking register
UNLOCK[6]=0xC0;UNLOCK[7]=0x0C;UNLOCK[8]=0xC3;
UNLOCK[9]=0xC3;UNLOCK[10]=0x33;UNLOCK[11]=0x33;
e = SendComCAN(UNLOCK, sizeof(UNLOCK),3);
if(e < 0){CommandMsg("ERROR UNLOCK", -1); return -1;}

// High-address space block locking register
UNLOCK[6]=0xC0;UNLOCK[7]=0x08;UNLOCK[8]=0xB2;
UNLOCK[9]=0xB2;UNLOCK[10]=0x22;UNLOCK[11]=0x22;
e = SendComCAN(UNLOCK, sizeof(UNLOCK),3);
if(e < 0){CommandMsg("ERROR UNLOCK", -1); return -1;}
```

Il comando 0x49 viene inviato 6 volte, opportunamente modificato per agire sui diversi registri che sono:

- FLASH_A_LMLR: *Low/Mid-address space block locking register* (C3 F8 80 04)
- FLASH_A_SLMLR: *Secondary low/mid-address space block locking register* (C3 F8 80 0C)

- FLASH_A_HLR: *High-address space block locking register* (C3 F8 80 08)
- FLASH_B_LMLR: ”” ”” (C3 F8 C0 04)
- FLASH_B_SLMLR: ”” ”” (C3 F8 C0 0C)
- FLASH_B_HLR: ”” ”” (C3 F8 C0 08)

Dopo questi 6 invii del comando sono state sbloccate tutte le zone della memoria.
Come si può osservare in Figura 26, la memoria è suddivisa in zone: low, mid e high che a loro volta sono suddivise in blocchi identificati dal loro indirizzo di partenza.
È necessario dopo aver sbloccato le zone, intervenire anche sui singoli blocchi che riguardano la cancellazione. Essi sono numerati con potenze di due, come si può vedere nello switch-case in cui la variabile *BLOCK* identifica il blocco.
Ad esempio se la zona da cancellare corrisponde all'indirizzo 0x00067800 (Figura 26), sappiamo che il blocco si trova nella zona MID della memoria flash A (identificato dal valore *BLOCK* = 0x20000), la quale è suddivisa secondo la tabella della memoria, in due blocchi da 128 Kilobyte l'uno.
È dunque necessario sbloccare il LMLR_A (*Low/Mid Address space block locking register*) e il SLMLR_A (*Secondary Low/Mid Address space block locking register*) tramite i seguenti comandi:

```
07 E0 49 00 00 00 C3 F8 80 04
07 E0 A1 A1 11 11 00 02 00 00

07 E8 89 00 11 11 00 02 00 00

07 E0 49 00 00 00 C3 F8 80 0C
07 E0 C3 C3 33 33 00 02 00 00

07 E8 89 00 33 33 00 02 00 00
```

Figura 27

Si possono notare gli indirizzi dei due registri in azzurro, l'indice del blocco (0x20000) e la chiave di sblocco della zona.
Infine prima di poter effettuare la cancellazione è necessario agire su altri due di un ulteriore insieme di 4 registri che sono:

- FLASH_A_LMSR: *Low/mid address space block select register* (C3 F8 80 10)
- FLASH_A_HSR: *High address space block select register* (C3 F8 80 14)
- FLASH_B_LMSR: ”” ”” (C3 F8 C0 10)
- FLASH_B_HSR: ”” ”” (C3 F8 C0 14)

Per comprendere meglio questa operazione viene mostrato in Figura 28 un esempio di sblocco di questi registri. In questo caso si vuole effettuare la cancellazione a partire dall'indirizzo 102000, che si trova nell'High address space.

Essendo la parte alta condivisa dalla flash A e dalla flash B, bisogna intervenire sui registri FLASH_A_HSR e FLASH_B_HSR.

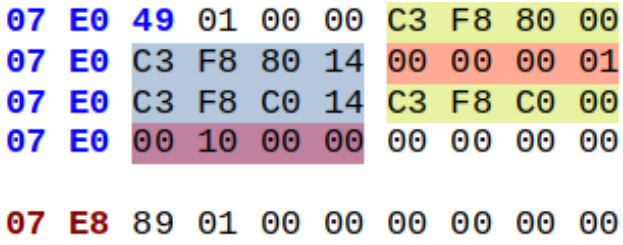

Figura 28

Gli altri elementi evidenziati sono rispettivamente: l'indirizzo di partenza della zona da cancellare (00 10 00 00), l'indice del blocco (00 00 00 01), e gli indirizzi degli MCR (Module configuration register).
Il codice per l'invio di quest'ultimo comando è il seguente:

```
BYTE COM[] = {0x49,0x01,0x00,0x00,0xC3,0xF8,0x80,0x00,
              0xC3,0xF8,0x80,0x10,0x00,0x00,0x00,0x01,
              0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
              0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

COM[6] =BYTE(REG>>8); // per selezionare la parte alta dal valore di REG
COM[10]=BYTE(REG>>8); // (es. REG = 8004 -> 80), (es. REG = C004 -> C0)

if(Address>0xC0000){  // solo per zone High
  COM[11]=0x14; COM[16]=0xC3;
  COM[17]=0xF8; COM[18]=0xC0;
  COM[19]=0x14; COM[20]=0xC3;
  COM[21]=0xF8; COM[22]=0xC0;
}
// questi shift invece selezionano il numero del blocco
COM[13]=BYTE(BLOCK>>16); // (es. BLOCK = 00001 -> 00)
COM[14]=BYTE(BLOCK>>8);  // (es. BLOCK = 00001 -> 00)
COM[15]=BYTE(BLOCK); //  // (es. BLOCK = 00001 -> 01)
// questi selezionano invece l'indirizzo della zona
COM[25]=BYTE(Address>>16); // (es. Address = 100000 -> 10)
COM[26]=BYTE(Address>>8);  // (es. Address = 100000 -> 00)
COM[27]=BYTE(Address);     // (es. Address = 100000 -> 00)

e = SendComCAN(COM, sizeof(COM),3);
if(e < 0){CommandMsg("ERROR COM", -1); return -1;}
```

Una volta sbloccati i registri si può procedere con la cancellazione vera e propria, che avviene tramite il comando 0x48.

```
BYTE LOOP[] = {0x48,0x00,0x00,0x00,0xC3,0xF8,0x80,0x00,
               0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

LOOP[6] =BYTE(REG>>8);
LOOP[9] = BYTE(Add>>16);
LOOP[10] = BYTE(Add>>8);
LOOP[11] = BYTE(Add);

if(Add>0xC0000)  // solo per zone High
{
  LOOP[12] = 0xC3;
  LOOP[13] = 0xF8;
  LOOP[14] = 0xC0;
}
int k=0;
do
{
  freeze(200);
  e = SendComCAN(LOOP, sizeof(LOOP),3);
  if(e < 0){CommandMsg("ERROR LOOP", -1); return -1;}
}while(ReadBuffer[29]!=1 && k++<20);
```

Questo comando, una volta che è stato opportunamente impostato in base al blocco e all'indirizzo, viene inviato ripetutamente in attesa di risposta da parte della centralina.
Ad ogni invio la centralina risponde nel seguente modo, mentre provvede alla cancellazione dei dati:

```
07 E0 48 00 00 00 C3 F8 C0 00
07 E0 00 0C 00 00 00 00 00 00
07 E8 40 00 00 00 00 00 00 00
```

Figura 29

Una volta che la cancellazione è terminata restituisce una risposta diversa:

```
07 E0 48 00 00 00 C3 F8 C0 00
07 E0 00 0C 00 00 00 00 00 00
07 E8 40 01 00 00 00 00 00 00
```

Figura 30

Quel byte posto a 0x01 è di fatto la condizione di uscita dal ciclo, assieme ad un contatore per evitare un'eventuale mancata terminazione. Infine per concludere la procedura di cancellazione si invia l'ultimo comando che verifica che la zona sia stata cancellata:

```
BYTE CLOSE[] = {0x48,0x01,0x00,0x00,0x00,0x00,0x00,0x00};
// varie operazioni di shift per gestire le varie zone
CLOSE[2]=BYTE(Add>>16);
CLOSE[3]=BYTE(Add>>8);
CLOSE[4]=BYTE(Add);
CLOSE[5]=BYTE(Len>>16);
CLOSE[6]=BYTE(Len>>8);
CLOSE[7]=BYTE(Len);

e = SendComCAN(CLOSE, sizeof(CLOSE),3);
if(e < 0){CommandMsg("ERROR CLOSE", -1); return -1;}
// non viene inviato due volte,
// il parametro 5 indica l'attesa di un ulteriore risposta
e = SendComCAN(CLOSE, sizeof(CLOSE),5);
```

Il comando che viene inviato è il seguente:

```
07 E0 48 01 06 00 00 02 00 00
```

Figura 31

A cui la centralina risponde con:

```
07 E8 88 00 06 00 00 02 00 00
07 E8 00 08 00 00 FF FF FF FF
```

Figura 32

Indicando che la zona è vuota (FF FF FF FF). Si può ulteriormente verificare che la procedura sia andata a buon fine verificando i CRC calcolati sulla memoria flash. È sufficiente verificare che il CRC corrisponda alla stringa: **45 51 0D 39 3F 55 D1 7F** che equivale al calcolo del CRC tramite l'apposito algoritmo, su una zona completamente vuota. Questo passaggio è opzionale nella procedura di scrittura e può essere utile come ulteriore verifica.

2.5.3.3 Scrittura della zona

Una volta completata la cancellazione è possibile passare alla fase di scrittura dei dati. Questa fase prevede l'utilizzo di un solo comando: **0x47**.

```
BYTE COM47[24] = {0x47,0x00,0x01,0x00,0xC3,0xF8,0x80,0x00,
                  0x00,0x00,0x00,0x00,0xC3,0xF8,0xC0,0x00,
                  0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00};

// Low-Mid della Flash B
if(Flash_Address[i] >= 0x80000 && Flash_Address[i] < 0x100000){
  COM47[6] = 0xC0;
  COM47[12] = 0x00;
  COM47[13] = 0x00;
  COM47[14] = 0x00;
  COM47[15] = 0x00;
}
// Low-Mid della Flash A
if(Flash_Address[i] < 0x80000){
  COM47[12] = 0x00;
  COM47[13] = 0x00;
  COM47[14] = 0x00;
  COM47[15] = 0x00;
}
for (f = Flash_Address[i]; f < Flash_Address[i + 1]; f+=0x200)
{
  for(k = 0; k < 0x200; k++){ // verifica che la zona non sia "vuota"
    if(writeFile[f+k] != 0xFF)
    {empty = false; break;}
  }
  if(!empty){ // se non è vuota devo procedere alla scrittura
    COM47[17] = (BYTE)(f >> 16);  // imposto indirizzo di partenza
    COM47[18] = (BYTE)(f >> 8);   // ""                          ""
    COM47[19] = (BYTE)(f);        // ""                          ""
    freeze(50);

    e = SendComCAN(COM47, sizeof(COM47), 3); // invio comando
    if (e < 0 && e!= -4) {CommandMsg("ERROR", -1); return -1;}
    for(s = 0; s < 0x200; s++)
          BUFFER[s] = writeFile[f+s];

        // invio dei 512 byte di dati
    e = SendComCAN(BUFFER, sizeof(BUFFER), 3);
    if (e < 0 && e!= -4){ CommandMsg("ERROR", -1); return -1;}
  }
  empty = true;
}
```

Come prima operazione si impostano i byte per i Module Configuration register, in base alla zona da scrivere.
Successivamente si verifica che la zona del file da scrivere non sia vuota, cioè con tutti i byte posti a 0xFF.
Questa verifica si fa per una questione di efficienza, infatti, dato che la zona nella memoria flash è stata cancellata, tutti i byte sono già impostati a 0xFF e in tal caso si può evitare di effettuare la scrittura. Se invece la zona nel file che si desidera scrivere, non è vuota devo procedere alla scrittura.
Dunque si impostano i tre byte contenenti l'indirizzo di partenza di scrittura, ottenendo il seguente comando:

```
07 E0 47 00 01 00 C3 F8 80 00
07 E0 00 00 00 00 C3 F8 C0 00
07 E0 00 10 20 00 00 00 02 00
```

Figura 33

In cui possiamo distinguere gli indirizzi dei Module Configuration Register,
l' indirizzo di partenza della scrittura e il numero di byte da scrivere.
Dopo aver ricevuto la risposta da parte della centralina si procede a riempire un buffer, che conterrà il contenuto del file che l'utente vuole scrivere. Una volta riempito si procede con l'invio dei 512 byte letti dal file dell'utente. Si ripete questa operazione per tutto il blocco di memoria.

2.5.4 Scrittura memoria EEPROM

In questo capitolo viene analizzata la procedura di scrittura della memoria EEPROM esterna al microcontrollore, che può essere schematizzata nel modo seguente:

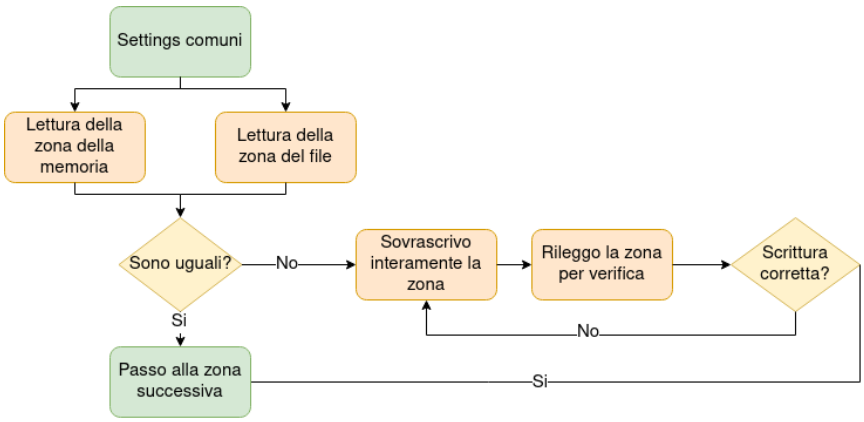

Figura 34

Viene mostrato ora il codice per implementare questa funzione,

```
// verifica lunghezza file
if(writeFileLen != 0x8000)
{   CommandMsg("ERROR : wrong file size", -1); return -1; }

  BYTE ReadEEP[] = {0x30 ,0x00 ,0x00 ,0x20 ,0x00 ,0x00 ,0x01 ,0x00 ,
                    0x30 ,0x00 ,0x00 ,0x00 ,0x00 ,0x00 ,0x01 ,0 x00};
  BYTE WRITE[0x108] = {0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00};
  corretto = false;
  DWORD i = 0;
  for(i = 0; i < 0x8000; i+=0x100){
    sovrascrivi = false;
    ReadEEP[3] = (BYTE)(i >> 8);

    e = SendComCAN(ReadEEP, sizeof(ReadEEP),7);  // lettura zona
    if(e < 0){CommandMsg("ERROR data EEP 1", -1); return -177;}
    e = PulisciBuffer_Tricore_BOOT(); // pulizia buffer
    if(e < 0){CommandMsg("ERROR data EEP 3", -1); return -1;}

    for(j = 0; j < 0x100; j++){
      if(ReadBuffer[j] != writeFile[i+j])
      { sovrascrivi = true; break; }
  }
```

In questa prima parte di codice vengono effettuate 3 operazioni.
Per prima cosa si verifica la dimensione del file che l'utente ha intenzione di scrivere, in maniera analoga a quanto è stato fatto per la memoria flash.
Il primo ciclo for serve per far avanzare ogni volta la zona della scrittura in quanto viene effettuata a blocchi di $0x100_{16}$, cioè 256 byte per volta, fino ad arrivare a 32 Kbyte.
Dopodiché viene inviato il comando che richiede al microcontrollore di effettuare la lettura della memoria EEPROM. Dopo la lettura, il contenuto della memoria viene inserito all'interno del buffer che veniva utilizzato anche per la memoria flash.
Un successivo ciclo for verifica che il contenuto del buffer sia diverso da quello del file da scrivere.
Se anche un singolo byte differisce, si deve procedere con la sovrascrittura dell'intera zona, che viene mostrata nella pagina seguente.

Questa parte della funzione è sempre interna al for che itera tutta la memoria.

```
if(sovrascrivi)

 WRITE[3] = (BYTE)(i >> 8); // indirizzo di partenza

// invio comando scrittura
 for(int k = 0; k < 0x100; k++) WRITE[k+8] = writeFile[(i+k)+offset];
 e = SendComCAN(WRITE, sizeof(WRITE), 3);
 if (e < 0) { CommandMsg("ERROR WRITE", -1); return -1; }

 // Rileggo la zona per verificare che sia stata sovrascritta
 e = SendComCAN(ReadEEP, sizeof(ReadEEP), 3);
 if (e < 0) CommandMsg("ERROR READ", -1);

 e = PulisciBuffer_Tricore_BOOT();
 if(e < 0){CommandMsg("ERROR pulizia Buffer", -1); return -1;}

 for (j = 0; j <= 0x100; j++) {
   if (ReadBuffer[j] != writeFile[i + j])
   {
     // ERRORE FILE NON SOVRASCRITTO CORRETTAMENTE
     if(e < 0){CommandMsg("ERROR scrittura file", -1); return -1;}
   }
 }
```

Nel codice illustrato sopra si utilizza il comando 0x45 per effettuare l'operazione di scrittura.

Questo comando è definito nell'array *WRITE* che ha dimensione 108_{16} (ovvero 256 + 8 byte), per poter contenere i byte da scrivere nella EEPROM ($0x100_{16}$) e il comando per avviare la procedura ($0x8_{16}$).

Per prima cosa viene riempito l'array *WRITE*, con i byte letti dal file che l'utente ha intenzione di scrivere. Successivamente si invia il comando per permettere al microcontrollore di effettuare la scrittura.

Per verificare che la scrittura sia andata a buon fine si va a leggere la zona appena scritta, confrontandola con il file di input.

Se le due parti confrontate risultano uguali allora la scrittura è andata a buon fine, ed è possibile proseguire con la successiva zona, altrimenti viene interrotta l'operazione e viene generato un messaggio di errore.

2.6 Test del driver

L'ultima fase del progetto consiste nel verificare il funzionamento del driver. Per prima cosa viene compilato e aggiunto nel software il codice sorgente illustrato nel capitolo precedente. Dopodiché viene effettuato il collegamento della centralina con il dfox pro, seguendo gli stessi collegamenti visti nella sezione 2.2.2, come mostrato in Figura 35.

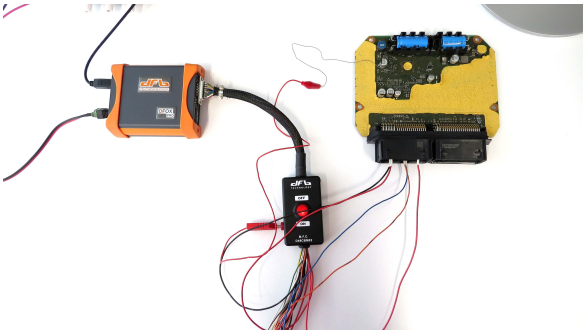

Figura 35: collegamento completo

Una volta collegato il dfox pro con la ECU, si avvia il software che si presenta con la seguente interfaccia:

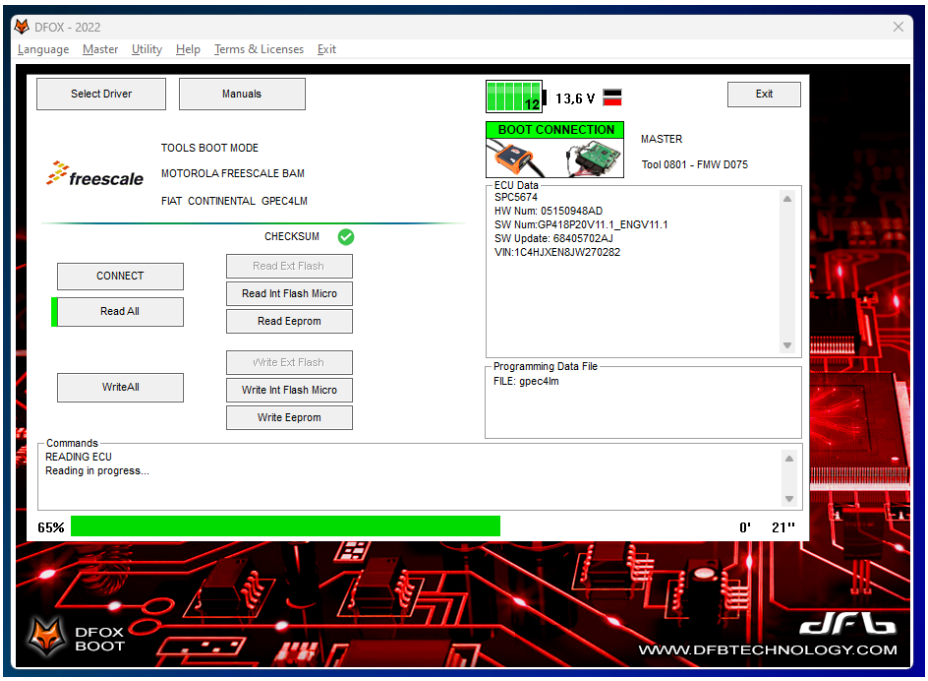

Figura 36: software dfox

Sono presenti diversi elementi con cui interagire per effettuare le varie operazioni.
Si seleziona per prima cosa il nuovo driver mediante l'apposito pulsante in alto a sinistra.
Per verificare che il driver funzioni correttamente, si estrae il contenuto di entrambe le memorie, dopodiché si vanno a effettuare alcune modifiche nei file estratti.
Infine si riscrivono entrambe le unità di memorizzazione con i file modificati, per cui leggendo nuovamente entrambe le memorie devono essere presenti nei file estratti anche le modifiche da noi effettuate.

3 Conclusioni

In questa tesi, vengono esposti solamente i punti essenziali del processo relativo ad un progetto ben più complesso e articolato. Alcune fasi, infatti, non vengono descritte nel dettaglio a causa della necessità di un'analisi più approfondita e di una conoscenza vasta non solo in campo informatico, ma anche in ambito elettronico, trattandosi di un settore altamente specializzato. A titolo di esempio, basti pensare al complesso progetto che sta alla base dello sviluppo della parte hardware del dfox pro, oppure alle complesse funzioni software che permettono di gestire la comunicazione tramite CAN-bus, impiegabili su centraline di qualsiasi tipologia e sviluppate da diverse case automobilistiche. Questo prodotto è il frutto di anni di studio ed esperienza nel settore automotive.
I passaggi che sono stati descritti in questa tesi sono a grandi linee applicabili a diversi tipi di centraline motore, ovviamente poi nello specifico ci sono degli aspetti che differiscono in ogni modello (come ad esempio il boot, i componenti hardware, ecc...).

Questo settore ha un futuro promettente grazie alla continua evoluzione dei veicoli, soprattutto con l'introduzione sul mercato delle auto elettriche.
La tecnologia delle centraline ha subito una svolta radicale, passando dalla gestione della miscela nei motori a combustione interna, alla gestione di motori elettrici all'avanguardia.
È importante considerare che i motori a combustione interna possono essere migliorati soprattutto grazie alle tecnologie di progettazione del motore stesso, mentre i motori elettrici dipendono maggiormente dalla loro gestione elettronica.
In pratica, mentre la gestione delle ECU nei motori a combustione interna controlla principalmente l'apporto di carburante e aria, la gestione elettronica dei motori elettrici regola la velocità, la coppia e il flusso di corrente tra la batteria e il motore elettrico.

4 Bibliografia/Sitografia

- Nxp Semiconductors, *"MPC5674F Microcontroller Reference Manual"*, www.nxp.com, 2015.
- Nxp Semiconductors, *"Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture"*, www.nxp.com, 2005.
- Alexjan Carraturo, Andrea Trentini, *"Sistemi Embedded: teoria e pratica"*, Ledizioni, Milano, 2017.
- Marco Di Natale, Haibo Zeng, Paolo Giusto, Arkadeb Ghosal, *"Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice"*, Springer, 2012.
- Infineon Technologies AG, *"Tricore microcontroller user manual vol.2"*, Germania, 2012.
- *"Cos'è una ECU"*, www.ecutesting.com/categories/ecu-explained
- *"I microcontrollori"*, www.techtarget.com/iotagenda/definition/microcontroller
- *"Evoluzione della ECU"*, www.motori.it/glossario/ecu-engine-control-unit
- *"Guida alla comunicazione rs-485"*, www.eltima.com/it/article/rs485-communication-guide
- *"La tecnologia V2X"*, www.autoweek.com/news/technology/a36190311/v2x-technology

Glossario

CRC (*cyclic redundancy check*) è un metodo per il calcolo di somme di controllo (checksum), in cui i dati di uscita sono ottenuti elaborando i dati in ingresso, tramite operazioni elementari basilari, quali somme, sottrazioni, ecc.. . 31

DMA (*Direct Memory Access*) è quel meccanismo che permette ad altri sottosistemi, come ad esempio le periferiche, di accedere direttamente alla memoria interna per effettuare operazioni di I/O, senza coinvolgere la CPU che può procedere ad eseguire altre operazioni. 12

driver è l'insieme di procedure software che si occupano della gestione dell'hardware. 9

DSP (*Digital Signal Processor*) è un particolare processore dedicato e ottimizzato per garantire un'elevata efficienza di esecuzione per semplici operazioni ricorrenti (ad esempio somme, sottrazioni, moltiplicazioni). 10

EEPROM (*Electrically Erasable Programmable Read-Only Memory*) è un dispositivo di memorizzazione non volatile che viene impiegato per memorizzare un piccola quantità di dati che necessitano di essere conservati anche in assenza di alimentazione elettrica, spesso contiene dati di configurazione. Le operazioni di scrittura e lettura vengono effettuate elettricamente. 7, 12

eSCI (*Serial Communication Interface*) è una particolare versione sviluppata dalla NXP, della normale interfaccia SCI che si occupa di effettuare la trasmissione seriale di bit tra il microcontrollore e le periferiche. 11

eTPU2 (*Enhanced Timing Processor Unit*) è essenzialmente una MCU indipendente che si occupa della gestione di I/O, comunicazioni seriali e controllo del tempo. 10

ISO/OSI (*Open System Interconnection*) è uno standard introdotto nel 1984 dall'International Organization for Standardization, per l'interconnessione di sistemi di computer. Tale modello stabilisce per l'architettura logica di rete, una struttura a strati composta da una pila di protocolli di comunicazione di rete suddivisa in 7 livelli. 16

NXP (*Next eXPerience SemiConductors*) è un'azienda di semiconduttori fondata dalla Philips nel 2006, con attualmente più di 40 sedi in tutto il mondo. 10

payload (*Carico utile*), indica la parte di dati trasmessi effettivamente di interesse per l'utilizzatore. 27

Power Architecture un gruppo di specifiche che riuniscono un ampio insieme di set di istruzioni per microprocessori RISC, cioè dei particolari tipi di processori con instruction set ridotti per ottenere prestazioni più elevate. 10

SIMD (*Single Instruction stream, Multiple Data Stream*), consiste in un elevato numero di processori identici che eseguono la stessa sequenza di istruzioni su insiemi di dati differenti, costituito da un'unica Control Unit che gestisce più ALU che operano in maniera sincrona. 10

SRAM (*Static Random Access Memory*), è un tipo di ram volatile che non necessita di memory refresh. 10, 11