POLITECNICO DI TORINO

Dipartimento di Ingegneria Meccanica e Aerospaziale

Corso di Laurea in Ingegneria Meccanica

Tesi di Laurea Magistrale

Prove di fatica di un dispositivo park-lock: configurazione del sistema automatizzato di controllo e dell'interfaccia utente

Relatore:
prof. Massimo Sorli

Candidato:
Luca Brachet Cota

Tutor aziendale:
ing. Giorgio Scalici

Anno Accademico 2017 - 2018

Ringraziamenti

Desidero ringraziare con il cuore tutti i famigliari e gli amici che mi hanno ampiamente supportato (e sopportato) nei miei cinque lunghi anni di Politecnico, così come lo stanno ancora facendo nel recente inizio di esperienza lavorativa alla Oerlikon Graziano. Grazie a chi mi ha ascoltato, a chi si è interessato di me e a chi ha cercato di capirmi ed aiutarmi nonostante sia un pessimo comunicatore, tanto dal punto di vista verbale che da quello non verbale. Grazie ancora a tutti quelli che hanno creduto in me molto di più di quanto non ci credessi io e che hanno provato di allontanare quel pessimismo che spesso mi coglie quando devo affrontare contesto nuovo o una prova impegnativa. È grazie a tutti voi che sono qua ora ed è anche per voi che cercherò di migliorare le mie capacità di uomo e di ingegnere. Grazie, vi voglio bene.

Un ringraziamento particolare va a mamma Bruna e papà Mauro, a Sara, a Pierre e Bolo, a Carmine, a Hilde e Giusy, a Giorgio, ad Antonio, ai Merli, a Donato, a Paola, a Mattia, a Claudio e a Manuel.

Abstract

Tra i numerosi dispositivi di sicurezza impiegati nell'ambito automotive, il *park-lock* è certamente tra quelli ingegneristicamente più interessanti ma anche tra i più delicati dal punto di vista della sicurezza (*safety*). Il livello di affidabilità richiesto per il suo funzionamento comporta che nel DVP (*Design Validation Plan*) del prodotto su cui verrà integrato siano inclusi test particolarmente gravosi, da realizzare secondo condizioni operative controllate e ben documentabili. Tra le prove di validazione più impegnative si distinguono quelle di fatica, il cui scopo è simulare in un breve lasso temporale l'effetto sulle funzionalità del dispositivo di anni di utilizzo a bordo veicolo. Questa tesi presenta la strategia che è stata messa in atto per implementare il sistema di controllo e di interfacciamento con l'utente di un *park-lock* progettato dalla *Oerlikon Graziano*; il sistema in questione è stato configurato per realizzare in modo automatizzato alcune prove di fatica previste dal DVP a cui si è accennato in precedenza. La consultazione di questo elaborato è consigliata a chiunque desideri approcciare un'applicazione di controllo *park-lock* assimilabile a quella delineata e sia alla ricerca un riferimento metodologico e pratico da seguire.

Indice generale

Indice delle figure

Indice delle tabelle

Introduzione

Il *park-lock* è un dispositivo di sicurezza molto comune nell'ambito automotive, la cui presenza è obbligatoria a bordo dei veicoli elettrici e di quelli dotati di trasmissione automatica o a doppia frizione. Esso adempie al seguente bisogno: mantenere il veicolo fermo nella posizione di parcheggio qualora il conducente scenda senza inserire il freno a mano o questo risulti inutilizzabile. Siccome è evidente che un malfunzionamento del *park-lock* potrebbe determinare ingenti danni a ciò che circonda il mezzo, il soddisfacimento dei requisiti di *safety* inerenti al suo funzionamento costituisce un punto centrale sia per la progettazione che per la definizione delle strategie di validazione. Il delicato compito del *park-lock* implica che i test previsti dal DVP (*Design Validation Plan*, ossia piano di validazione del design) della trasmissione su cui sarà integrato siano molto severi, sia in termini delle sollecitazioni applicate (meccaniche, termiche, …) che in quelli di controllabilità e documentabilità delle condizioni di prova. In particolare, tale documento prevede sempre per il dispositivo numerose prove di fatica: trattasi di sequenze di cicli di funzionamento eseguiti consecutivamente e in specifiche condizioni operative; il fine principale di tali test è verificare la durabilità degli elementi funzionali del *park-lock* e, soprattutto, la ripetibilità del loro comportamento, da cui dipende l'affidabilità dell'intero meccanismo.

Con questo elaborato si desidera presentare l'iter che è stato seguito per realizzare un sistema di controllo e di interfacciamento con l'utente capace di gestire in modo automatico, affidabile e ripetibile alcune prove di fatica programmate per un *park-lock* progettato da *Oerlikon Graziano* (OG). È opportuno evidenziare fin da subito che le caratteristiche specifiche delle fasi di lavoro descritte successivamente dipendono molto dalle peculiarità elettromeccaniche e sensoristiche del *park-lock Oerlikon Graziano*, dai requisiti imposti come target per il sistema di controllo e, soprattutto, dagli strumenti (sia attrezzature che software)

a disposizione per il lavoro. Ciò nonostante, si reputa che il piano di azione proposto nella tesi integri considerazioni teoriche e soluzioni pratiche interessanti anche al di fuori del contesto particolare considerato; tali elementi, infatti, possono costituire un riferimento importante per chiunque debba cimentarsi nell'implementazione di un sistema completo per la gestione controllata e automatizzata delle prove di fatica di un dispositivo *park-lock*.

Procedendo nella descrizione specifica delle attività svolte, siccome il lavoro che verrà presentato è nato e si è sviluppato come una tesi in azienda (nella sede di Rivoli della *Oerlikon Graziano*), gli obiettivi da perseguire e le fasi operative sono stati concordati insieme al mio tutor aziendale, l'ing. Giorgio Scalici. Inizialmente, mi è stato domandato di analizzare e comprendere (anche mediante la lettura dei codici sorgente disponibili) il funzionamento di due centraline identiche, progettate da una ditta esterna per il controllo di piccoli motori elettrici a spazzole (*brushed*) operanti in corrente continua. In seguito, ho dovuto implementare un'interfaccia grafica adeguata per osservare l'andamento delle grandezze di interesse (posizione e corrente) per entrambi gli attuatori e comandare a distanza le corrispondenti centraline. Il passaggio successivo si è concretizzato nella taratura statica dei sistemi di trasduzione caratteristici dei complessi di interfaccia e di controllo, realizzata mediante gli strumenti di misura di riferimento disponibili. Infine, si è reso necessario riprogrammare le schede elettroniche delle due centraline, così da poterle impiegare per il controllo in parallelo degli attuatori elettrici di due *park-lock*; in particolare, ogni centralina ha dovuto garantire l'esecuzione automatizzata di un significativo numero di cicli di attuazione (centinaia di migliaia), garantendo la ripetibilità di alcune caratteristiche fondamentali nelle leggi di posizione e corrente.

Il primo approccio alle questioni oggetto di tesi è stato analitico e osservativo, finalizzato alla comprensione approfondita della struttura e del funzionamento sia del *park-lock* che delle centraline di controllo. Per la conoscenza degli aspetti meccanici sono stati fondamentali il confronto con il tutor aziendale, lo studio del modello CAD del *park-lock* e l'ispezione di alcuni *datalog* (ossia registrazioni di dati sperimentali) relativi ad attuazioni del dispositivo. L'apprendimento, invece,

delle peculiarità hardware e software della centralina ha richiesto la consultazione dei datasheet dei componenti elettronici, di manuali di programmazione e di siti internet specializzati in applicazioni simili. Per l'implementazione dell'interfaccia è stato necessario documentarsi sulle caratteristiche della comunicazione CAN, per poi leggere manuali d'uso e svolgere tutorial relativi all'utilizzo del software disponibile per realizzare tale scopo. La calibrazione dell'algoritmo di controllo del moto e quella dei sistemi di lettura hanno richiesto la consultazione di volumi didattici e, soprattutto, l'esecuzione di molteplici prove sperimentali, per le quali è stato necessario un accurato lavoro di raccolta e post-processamento dei dati. Infine, le attività di perfezionamento relative al controllo si sono basate sulla comprensione approfondita dei sorgenti della centralina e sulla verifica sperimentale del successo delle modifiche ad essi apportate.

Il contenuto dell'elaborato è strutturato secondo la seguente scaletta:

- **Capitolo 1**

Il primo capitolo si focalizza sulla descrizione del funzionamento del *park-lock* OG, evidenziando l'importanza delle modalità di accoppiamento tra i singoli componenti al fine di ottenere il particolare comportamento del dispositivo. La porzione centrale è riservata alla presentazione del DVP della trasmissione e al resoconto delle prove previste per il *park-lock*. In calce al capitolo, viene proposto l'elenco degli strumenti forniti dall'azienda per lo svolgimento delle attività di tesi.

- **Capitolo 2**

Il secondo capitolo elenca i componenti hardware delle centraline di controllo e per ciascuno di essi vengono indicate le principali caratteristiche tecniche e funzionali; inoltre, sono descritte le connessioni proprie del dispositivo, che gli consentono di scambiare informazioni o potenza elettrica internamente o con l'esterno. Nella seconda parte del capitolo ci si sofferma sulla struttura dei progetti sorgente delle schede elettroniche e ne viene schematizzata la logica di funzionamento ciclico.

- **Capitolo 3**

Il terzo capitolo inizia con la descrizione delle caratteristiche logiche e fisiche della comunicazione CAN e prosegue con la spiegazione del contenuto dei messaggi scambiati tra PC e centralina. La seconda metà del capitolo espone minuziosamente la sequenza di azioni che consente, partendo da pochi elementi iniziali, di creare un'interfaccia utente nell'ambiente software *CANape* adatta alla gestione di un'applicazione analoga a quella considerata.

- **Capitolo 4**

Il quarto capitolo mostra le leggi di spostamento prese come riferimento per l'attuazione del *park-lock* nelle prove di fatica e ne sottolinea le caratteristiche significative che si desidera riprodurre. Quindi, sono rappresentati con schema a blocchi e analizzati due possibili algoritmi di controllo del moto, uno a doppio anello chiuso e uno ad anello aperto. Si prosegue descrivendo la calibrazione dell'algoritmo ad anello aperto e mostrando i risultati della sua implementazione; il capitolo si chiude con il confronto tra le due possibilità di controllo e con la giustificazione della strada intrapresa.

- **Capitolo 5**

Il quinto capitolo propone le modifiche ai sorgenti che hanno permesso di ottenere la lettura ottimale dei valori di posizione e di corrente assorbita tramite l'interfaccia grafica; quindi, vengono riportate le fasi operative che hanno permesso di tarare le letture delle medesime due grandezze. L'ultima porzione di questo capitolo descrive l'implementazione della limitazione di corrente in attuazione, del gradino di corrente a fine corsa e del funzionamento ciclico automatizzato del *park-lock*.

- **Capitolo 6**

Il sesto e ultimo capitolo è riservato alle considerazioni conclusive sul lavoro compiuto; esse derivano principalmente dalla valutazione della capacità del sistema di mantenere le caratteristiche del controllo definite in fase di set-up (legge del moto, ripetibilità, affidabilità, …) nel lungo periodo e in particolare tra l'inizio e la

fine di una prova di fatica. La sezione finale presenta alcuni spunti di studio, legati sia all'ambito del controllo che a quello della modellazione matematica, che potrebbero consentire, in futuro, il completamento e il perfezionamento del sistema automatizzato descritto in questo elaborato.

CAPITOLO 1

Il park-lock e il suo piano di validazione

Questo primo capitolo inizia con la descrizione del contesto meccanico (*park-lock* e trasmissione) a cui fanno riferimento le attività di tesi, la cui comprensione è fondamentale affinché la logica seguita nella presentazione del lavoro sia chiara al lettore. Di seguito, si procederà con la presentazione del piano di validazione del design definitivo della trasmissione su cui il *park-lock* sarà montato, soffermandosi soprattutto su quanto indicato circa il dispositivo di sicurezza. L'ultima sezione mostrerà la postazione di lavoro e l'attrezzatura che è stata utilizzata per l'esecuzione delle attività sperimentali di tesi.

1.1 Il park-lock nel campo automotive

L'intera tesi si sviluppa intorno alla tematica del controllo di un particolare dispositivo di sicurezza che prende il nome di *park-lock* (o *parking pawl*). Si tratta di un congegno molto diffuso in ambito automotive (fig. 1.1) e la cui presenza è obbligatoria per legge su tutti i veicoli dotati di trasmissione elettrica, automatica o a doppia frizione [1-2]. Si tratta, infatti, di un dispositivo indispensabile nei casi in cui non è possibile sfruttare l'azione del freno motore a veicolo spento, la quale tende a mantenerlo fermo anche se il freno a mano non è stato inserito. In commercio si trovano *park-lock* attuati elettricamente, idraulicamente o manualmente: la scelta dipende dalle prestazioni (coppia, velocità, ecc.) domandate al dispositivo di sicurezza.

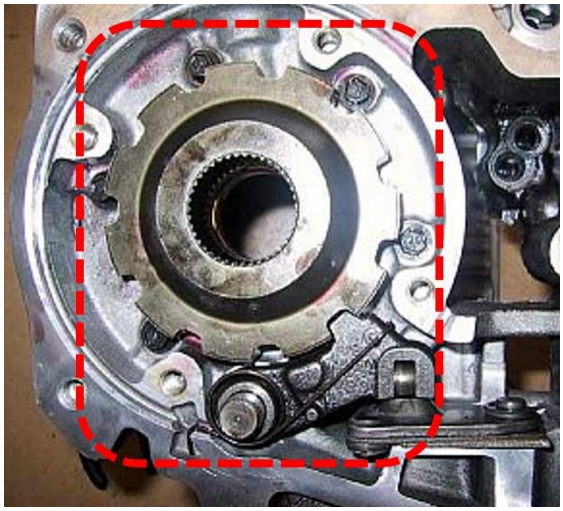


Fig. 1.1: esempio di park-lock (cerchiato in rosso)

In una trasmissione manuale generalmente la frizione è innestata (ossia chiusa) per azione di un meccanismo a molla a tazza e spingidisco: quando il pedale della frizione è alzato spingidisco, disco e volano sono tenuti a pacco dalla molla a tazza e la coppia passa da albero motore a primario grazie all'attrito statico. Il disinnesto (apertura) viene azionato, invece, premendo il pedale della frizione e può avvenire per via idraulica, meccanica o elettromeccanica; in ogni caso, lo spostamento del manicotto con cuscinetto reggispinta determina il disaccoppiamento (a livello di carico e di velocità) dei due alberi interfacciati, consentendo così, ad esempio, di cambiare marcia e di avviare il veicolo senza sovraccaricare o spegnere il motore (fig. 1.2). In questo tipo di trasmissione, anche se il motore è spento e il circuito idraulico di attuazione è scarico, la molla mantiene la frizione innestata e, dunque, il collegamento rigido (fino ad un certo valore limite di coppia) tra ruote e motore [3-5]. Quindi, se nasce una coppia sul primario (per esempio se il veicolo è disposto in pendenza e il freno a mano non è tirato) questa viene riportata all'albero motore, il quale si oppone al movimento grazie alle resistenze passive date dalla circolazione di aria nei pistoni, dagli attriti, dall'inerzia, etc.

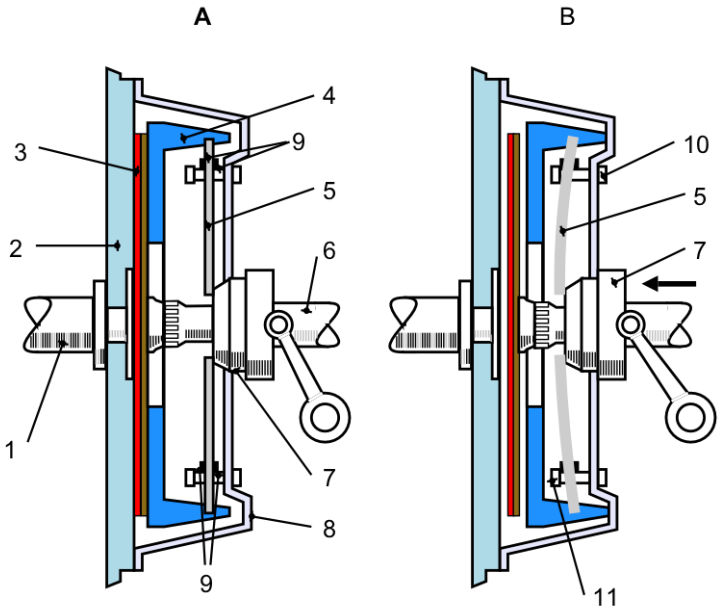


Fig. 1.2: attuazione di una frizione classica per trasmissione manuale

Invece, un veicolo a trasmissione automatica o a doppia frizione è dotato di più frizioni che vengono attuate secondo una logica opposta rispetto a quella appena presentata; infatti, ogni dispositivo di disaccoppiamento presenta delle molle che tengono i dischi normalmente separati, mentre la chiusura viene realizzata idraulicamente con olio in pressione (fig. 1.3). Questa configurazione di lavoro implica che, quando il veicolo è fermo e spento, la depressurizzazione della rete di attuazione idraulica determini l'assenza di collegamento tra le ruote del veicolo e il motore e pertanto l'impossibilità di sfruttare il freno motore. Nel casi di trasmissione automatica e a doppia frizione il *park-lock* agisce sempre su uno degli alberi che si trova verso l'uscita (costituita dai semiassi) del cambio e non sul primario, poiché la presenza delle frizioni ad innesto idraulico impedirebbe all'azione frenante di raggiungere le ruote [3-5].

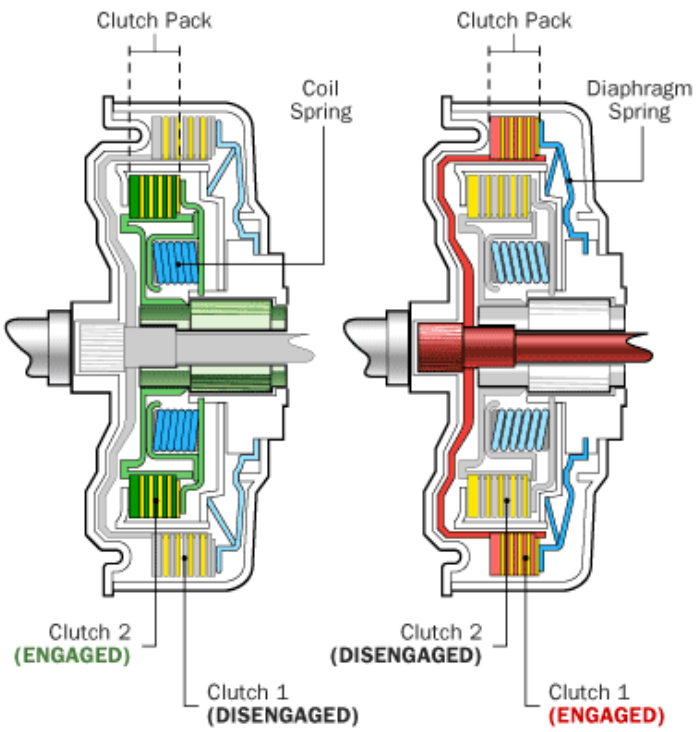


Fig. 1.3: attuazione di una doppia frizione concentrica

Infine, nella maggioranza dei casi (soprattutto quando non sono richieste alte prestazioni) le trasmissioni per veicolo completamente elettrico non prevedono l'utilizzo di frizioni, così come non è richiesta la presenza di un cambio multi-marcia: la trasmissione di potenza dal motore ai semiassi viene realizzata attraverso una cascata di ruote dentate ad uno o più stadi che concretizza un unico rapporto di trasmissione [3-5]. Questa configurazione meccanica è possibile perché i motori elettrici mostrano una legge coppia vs. numero di giri che parte con un valore molto elevato in corrispondenza dello spunto (rotore fermo) per poi diminuire gradualmente all'aumentare della velocità; se il veicolo che di desidera motorizzare elettricamente non ha particolari esigenze di coppia ad alte velocità un solo rapporto di trasmissione è generalmente accettabile (fig.1.4). A veicolo spento il motore elettrico è collegato alle ruote motrici come lo è quello termico in una trasmissione manuale, però l'assenza di valvole, la bassa inerzia del rotore e i ridottissimi attriti impediscono, nel secondo caso, un'azione frenante significativa che si opponga al moto spontaneo del veicolo. In questo caso, a differenza dei due precedenti, grazie

all'assenza di frizioni il *park-lock* può mantenere fermo il veicolo agendo sull'albero primario.

Fig. 1.4: esempio di trasmissione per veicolo elettrico a 2 stadi con park-lock

Dunque, i veicoli che presentano una delle tre possibilità di trasmissione presentate devono essere dotati di un dispositivo tipo *park-lock* che impedisca la rotazione del differenziale di output. Generalmente il *park-lock* viene innestato quando la leva del cambio è posta nella posizione indicata con la P di *Park* o comunque quando vengono rimosse le chiavi dal quadro. Questo genere di dispositivo e i suoi componenti attraversano delle fasi di validazione molto severe prima di poter essere prodotti in serie; la normativa SAE (*Society of Automotive Engineers*) J2208 [2] definisce in modo molto approfondito una serie di procedure raccomandate per la validazione di un *park-lock* e fornisce delle indicazioni anche in riferimento al sistema di controllo e alla trasmissione su cui verrà montato.

Sulle automobili e mezzi pesanti in commercio si ritrovano *park-lock* che possono essere differenti per la soluzione costruttiva, il comportamento statico e dinamico, la strategia e il mezzo di attuazione [6-14]; nonostante questo, si ritrovano sempre alcuni elementi comuni che servono a realizzare meccanicamente un principio di funzionamento evidentemente uguale per tutti. La figura 1.5 mostra un dispositivo *park-lock* progettato per una trasmissione automatica, studiato ed

ottimizzato nel comportamento dinamico da due ricercatori della *University of Technology* di Sydney [8].

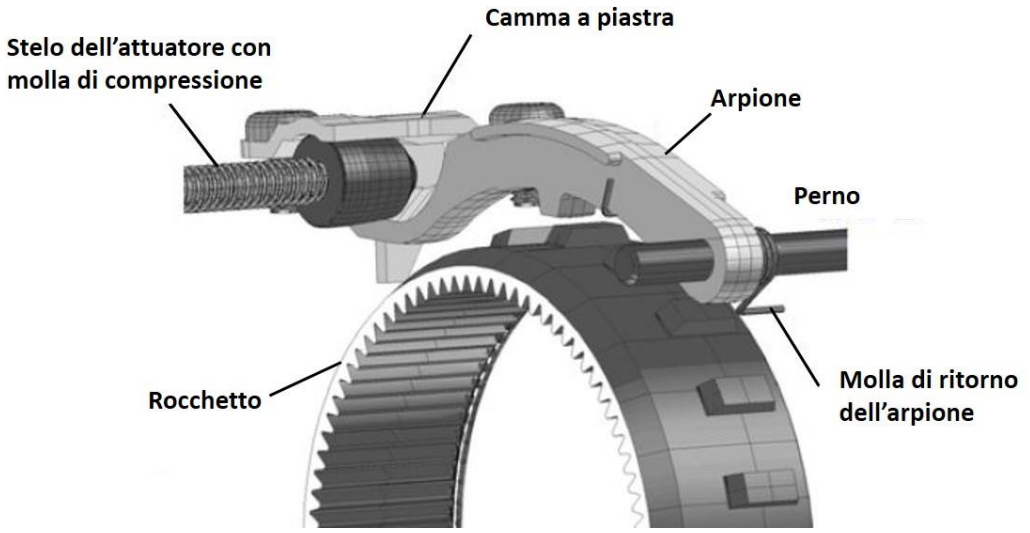


Fig. 1.5: esempio di soluzione costruttiva per un park-lock

In riferimento al dispositivo mostrato nella figura precedente, quando si comanda l'innesto (ossia quando desidera avviarne l'azione di sicurezza) del *park-lock* lo stelo di attuazione viene spinto verso la camma a piastra, ad esempio mediante un motorino elettrico DC accoppiato ad un sistema di trasformazione del moto da rotativo a lineare. L'estremità dello stelo viene così spinta contro due superfici coniugate, una della camma (fissa) e una dell'arpione (mobile); la forza che si sviluppa nel contatto determina la rotazione dell'arpione intorno al perno su cui è incernierato verso il rocchetto, a sua volta accoppiato all'albero di uscita mediante uno scanalato.

Quindi la fase di innesto vede il dente dell'arpione inserirsi nella cava tra due denti del rocchetto, determinando così l'impedimento della rotazione dell'uscita in entrambe le direzioni (a meno di un piccolo gioco). La molla di compressione consente sia all'attuatore di terminare la propria corsa anche se il dente non trova subito la cava, che l'inserimento del *park-lock* quando il veicolo si muove entro un ristrettissimo range di velocità intorno allo zero: la sua importanza verrà esplicitata nella descrizione del dispositivo progettato dalla *Graziano*. Per quanto riguarda il

disinnesto del *park-lock*, lo stelo viene ritirato nella direzione opposta a quella di innesto, allontanando la sua estremità dalla camma e dall'arpione; quindi la molla di ritorno (o di richiamo) riporta l'arpione alla posizione di partenza, rimuovendo così il dente dalla cava e rendendo nuovamente possibile il moto del veicolo.

Gli elementi che accomunano la maggior parte dei *park-lock* sono i seguenti:

- Arpione con dente o testa di innesto.
- Rocchetto calettato solidalmente sull'albero da bloccare.
- Sistema di attuazione, che può prevedere o meno la trasformazione del moto da rotativo a lineare.
- Molla di compressione (o torsionale a seconda della configurazione meccanica).

1.2 Il park-lock Oerlikon Graziano

L'intera tesi si sviluppa intorno ad un particolare dispositivo di sicurezza che costituisce parte integrante di una trasmissione monomarcia per veicolo elettrico, sviluppata dalla *Oerlikon Graziano* negli ultimi anni, i cui più recenti prototipi stanno affrontando gli ultimi test di validazione presso l'area *testing* della sede di Rivoli. La suddetta trasmissione è stata progettata per il mercato cinese e rappresenta uno dei progetti più importanti portati avanti dalla *Oerlikon Graziano* nell'ambito delle applicazioni elettriche in campo automotive; essa è stata dimensionata per l'accoppiamento con un motore elettrico caratterizzato dai seguenti valori di picco: 270 Nm (coppia), 120 kW (potenza), 14000 rpm (velocità).

Le immagini seguenti consentono di osservare la scatola esterna della trasmissione e di intuirne la struttura interna, soprattutto per quanto riguarda la disposizione di alberi e ruote dentate.

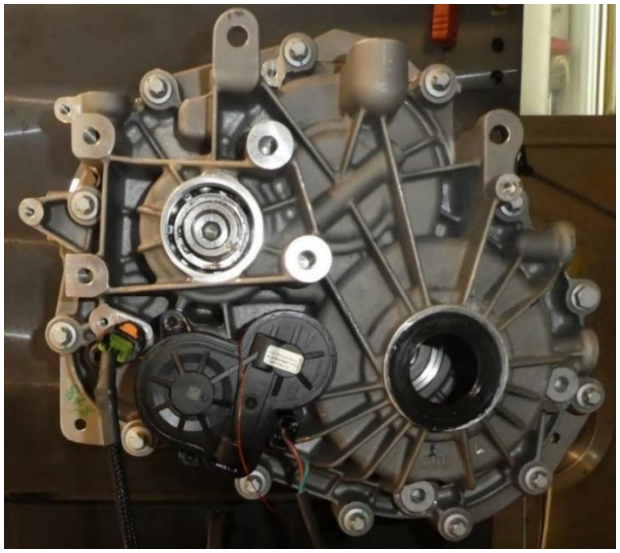


Fig. 1.6: vista dell'esterno della trasmissione OG

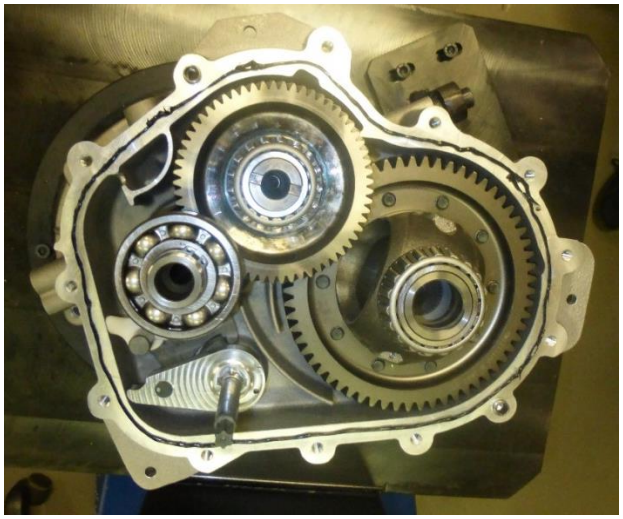


Fig. 1.7: vista dell'interno della trasmissione OG

La figura 1.7 consente di fare qualche precisazione sul funzionamento della trasmissione e sulla sua meccanica. In alto a sinistra è montato l'albero primario con il pignone di ingresso; tale albero riceve coppia direttamente dal motore elettrico, elemento assente dalla fotografia e che sostituisce il motore termico proprio dei veicoli convenzionali. In alto a destra è presente l'albero di rinvio, che porta due ruote dentate; quella di raggio superiore ingrana con il pignone precedentemente nominato a completare il primo step di riduzione (la velocità

diminuisce mentre la coppia aumenta), mentre quella di raggio inferiore costituisce il pignone che, ingranando con la corona del differenziale di uscita, completa il secondo passaggio di riduzione. Il differenziale, di tipo tradizionale a due satelliti, guida i due semiassi di uscita e tale assieme si può osservare in basso a destra; il rapporto di trasmissione (coppia di uscita su coppia di ingresso) complessivo tra pignone e scatola del differenziale è circa pari a 9.3.

Le figure 1.6 e 1.7 permettono, inoltre, di scorgere il *park-lock*, montato in basso a sinistra sulla trasmissione ed agente sul primario. In particolare, nella fig. 1.6 emergono il *case* nero della MGU (*Motor Gear Unit*, ossia unità integrante sia motore elettrico che riduttore) e il sensore di posizione ad effetto Hall, corredati dai corrispettivi cablaggi; invece, nell'immagine 1.7 si possono scorgere alcuni dei componenti che costituiscono il cuore cinematico e meccanico del dispositivo in questione. Le immagini seguenti mostrano il *park-lock* della *Oerlikon Graziano* (in forma CAD) sotto varie angolazioni, permettendo così di osservare la sua struttura e soprattutto di comprendere come i suoi elementi siano accoppiati ed interagiscano tra di loro; inoltre, all'interno delle suddette figure verranno indicati i nomi dei componenti presenti per aiutare la comprensione.

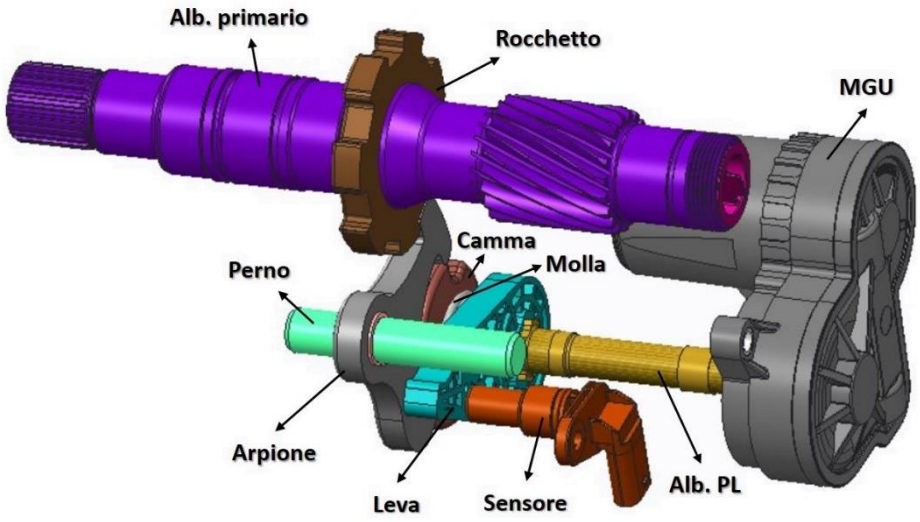


Fig. 1.8: vista CAD n.1 dell'assieme park-lock con primario, MGU e sensore

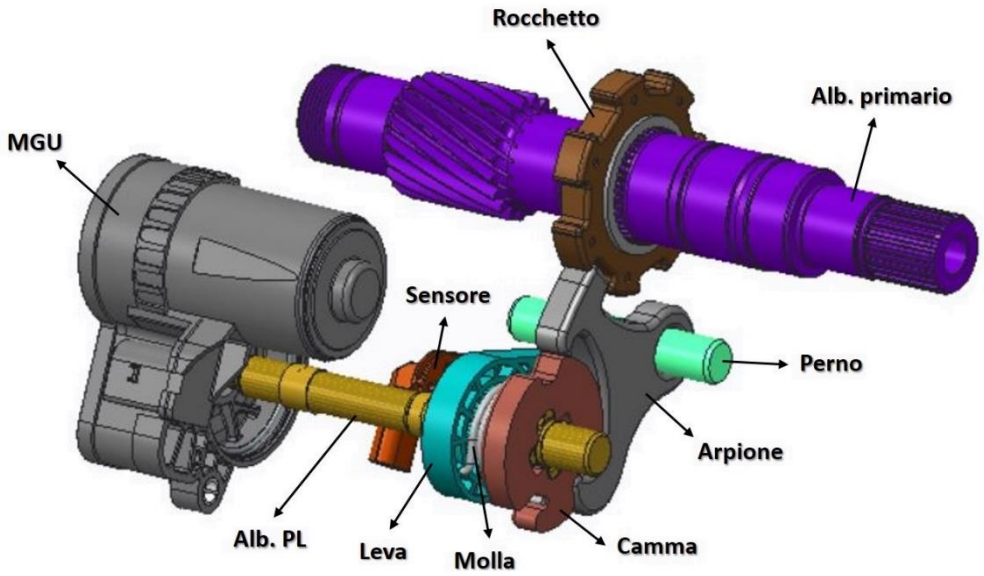


Fig. 1.9: vista CAD n.2 dell'assieme park-lock con primario, MGU e sensore

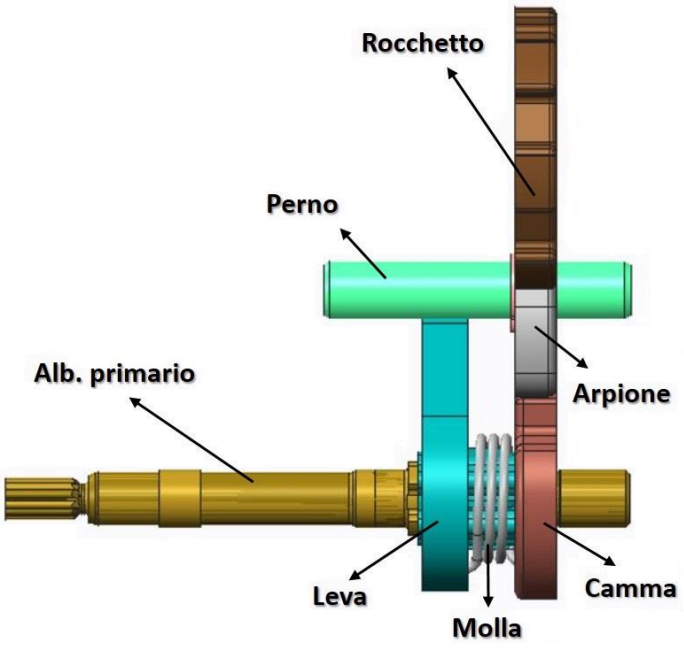


Fig. 1.10: vista CAD n.3 dell'assieme park-lock

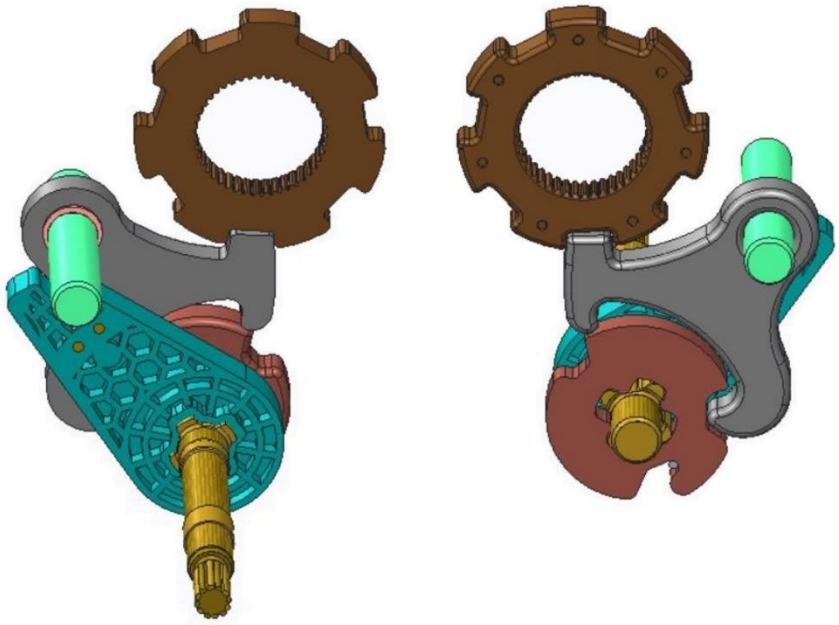


Fig. 1.11: vista CAD n.4 dell'assieme park-lock da lato primario (sx) e da lato MGU (dx)

L'albero di input è accoppiato ad un estremo con la MGU mediante uno scanalato e regge la camma, la molla e la leva porta-magente. La camma è montata sull'alberino attraverso un secondo profilo scanalato con gioco angolare di circa 30° (fig. 1.12), mentre la leva è calettata sull'albero con uno scanalato senza gioco. La molla, elemento fondamentale del dispositivo, è posta concentricamente all'alberino e i suoi due estremi sono agganciati rispettivamente alla camma e alla leva; l'azione della molla comporta che, in condizioni di assenza di carico, i profili scanalati femmina della camma siano schiacciati in un verso contro i fianchi dei denti dello scanalato maschio dell'albero, annullando quindi la presenza di gioco.

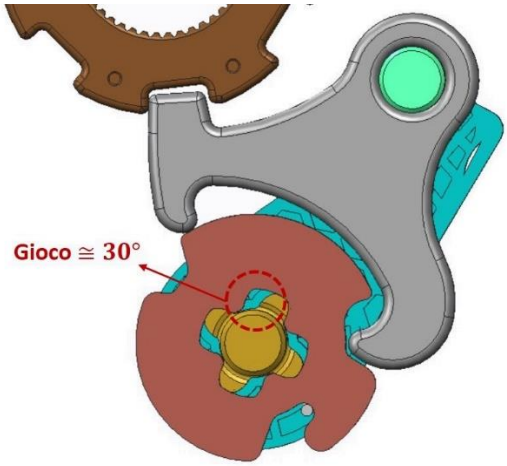


Fig. 1.12: gioco angolare tra camma e scanalato

La camma è desmodromica, ossia progettata per condurre l'elemento ad essa coniugato sia nella fase di rotazione in senso orario che in quella di rotazione in senso antiorario (naturalmente entro certi limiti di corsa), senza l'ausilio di una molla di richiamo e, quindi, unicamente grazie alla peculiare forma dei profili a contatto. L'arpione, elemento cruciale del *park-lock*, ruota intorno al perno ed è caratterizzato da tre estremità utili: due servono a completare l'accoppiamento di forma con la camma e una a realizzare l'azione di blocco del moto propria del dispositivo di sicurezza. Infatti, un rocchetto dentato è calettato saldamente sul primario della trasmissione (sempre mediante scanalato): quando viene fornito il comando, l'inserimento della testa dell'arpione tra due denti impedisce il moto dell'albero primario e di quanto si trova a valle, realizzando l'azione di sicurezza desiderata.

Nella figura successiva è possibile osservare i componenti del *park-lock* (esclusi MGU e sensore) l'alberino di ingresso (1), la camma desmodromica (2), la molla di torsione (3), la ralla anti-usura (4), l'arpione (5), il perno dell'arpione con anello *seeger* (6) e la leva porta-magnete (7).

Fig. 1.13: componenti determinanti la cinematica del park-lock

Bisogna sottolineare che la leva porta-magnete assolve a due distinte funzioni di grande importanza:

1. Come si può intuire dal nome, ingloba un piccolo magnete che produce il campo magnetico necessario per il funzionamento del sensore Hall di posizione.
2. Sbattendo contro un finecorsa solidale alla carcassa, costituito dallo stesso perno intorno a cui ruota l'arpione, limita la rotazione dell'alberino di ingresso del *park-lock* nella fase di innesto.

Le due immagini seguenti mostrano da sinistra verso destra la sequenza dei movimenti di camma e arpione che caratterizzano la fase di innesto (*engagement* o anche solo *engage*) del *park-lock* e poi quelli relativi alla fase di disinnesto (*disengagement* o *disengage*).

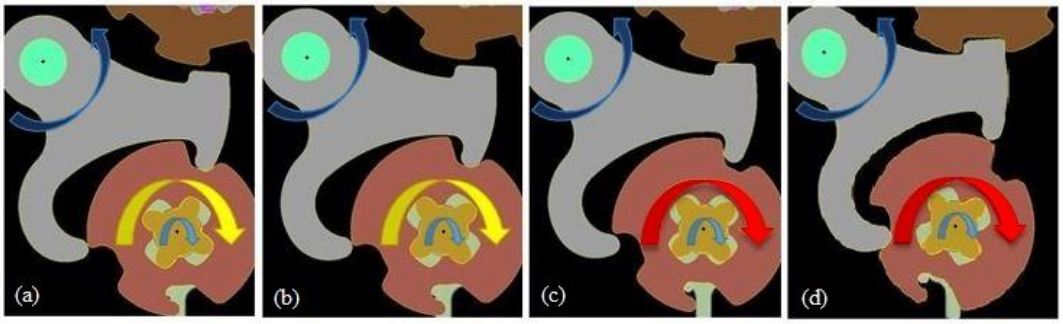


Fig. 1.14: movimenti di alberino, camma e arpione in engagement

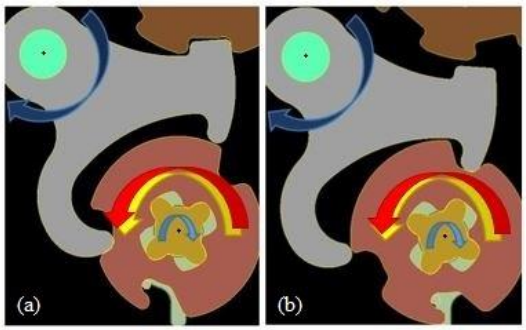


Fig. 1.15: movimenti di alberino, camma e arpione in disengagement

In entrambe le figure precedenti, il colore delle frecce viene impiegato rispettando le seguenti corrispondenze:

- Giallo: rotazione dell'alberino di ingresso.
- Rosso: rotazione della camma desmodromica.
- Blu: rotazione dell'arpione intorno al perno.
- Azzurro: azione del precarico della molla.

È opportuno sottolineare che, in realtà, i frame di movimento indicati come (b) e (c) in figura 1.14 sono stati separati solo per comodità di descrizione, per far comprendere al meglio i movimenti dei singoli componenti in gioco. Infatti, la fase di *engagement* del *park-lock* inizia con la messa in moto da parte dell'unità di attuazione dell'alberino di ingresso, il quale si muove seguendo una certa legge del moto fino a quando la leva non incontra il finecorsa/perno (b), per una rotazione di circa 27°. Nel frattempo, la rigidezza e il precarico della molla di torsione spingono anche la camma a ruotare con l'alberino, in modo non perfettamente sincrono a causa dei ritardi creati da inerzie ed attriti (c); il movimento prosegue fino a quando

l'arpione incontra la testa di un dente del rocchetto e quindi sia lui che la camma sono obbligati a fermarsi, mentre l'alberino può ancora ruotare fino al finecorsa grazie alla presenza del gioco angolare di circa 30°. La rotazione, presumibilmente lenta, del primario e la coppia che la molla (la cui torsione è aumentata) esercita sulla camma porta l'arpione ad infilarsi tra due denti e a completare così la fase di innesto (d).

Si ritiene opportuno soffermarsi sulla grande importanza della presenza della molla torsionale nel dispositivo di sicurezza. Tra i vari requisiti funzionali che deve soddisfare il *park-lock* ve ne è uno relativo al fenomeno del *ratcheting*: con questo termine si indica l'azione del carico intermittente che agisce quando si prova ad innestare il *park-lock* quando il veicolo non è completamente fermo. Se il *park-lock* fosse privo della molla (detta spesso 'di compressione') e l'accoppiamento tra la camma e l'albero fosse rigido e senza giochi elevati, il tentativo di innesto dell'arpione sul rocchetto in rotazione porterebbe con ogni probabilità ad un innesto incompleto ed incerto, quindi non accettabile in termine di sicurezza; inoltre, l'azione di carichi elevati ed impulsivi determinerebbe un sicuro danneggiamento degli elementi coinvolti, con possibile perdita di funzionalità da parte del *park-lock.* La molla di torsione evita, dunque, la situazione appena descritta e consente all'arpione di 'saltellare' sui denti del rocchetto in moto fino a quando questo non raggiunge una velocità sufficientemente bassa per un innesto privo di rischi secondo i due punti di vista evidenziati.

Per quanto riguarda la fase di *disengagement*, rappresentata in figura 1.15, l'alberino e la camma si muovono in modo solidale ed in senso opposto rispetto a quanto fatto in *engagement*; in questo caso la corsa dei due elementi viene limitata dalla geometria delle superfici a contatto camma - arpione e non è presente un finecorsa classico come quello sfruttato in *engagement*. Terminato il disinnesto, gli elementi del *park-lock* tornano ad essere nella stessa configurazione che presentavano prima dell'innesto.

La movimentazione dell'albero di ingresso del *park-lock* viene realizzata mediante un attuatore elettromeccanico. Si tratta di un attuatore già in uso su

applicazioni EPB (*Electric Park Braking*) ed è stato scelto perché in possesso di caratteristiche consone per azionare il *park-lock* progettato; inoltre, essendo prodotto in milioni di unità l'anno, ha *piece-price* (costo al pezzo) e costi di sviluppo molto ridotti rispetto ad un nuovo sviluppo. Gli attuatori come questo vengono frequentemente chiamati con il termine MGU, che corrisponde a *Motor Gear Unit*, ad indicare che non è presente unicamente, ad esempio, un motore elettrico quale fonte primaria di potenza meccanica ma sono inclusi anche elementi meccanici volti a garantire l'opportuna trasmissione del moto e della coppia fino all'uscita, oltre che, eventualmente, ad assolvere a funzionalità specifiche. Le figure 1.16, 1.17 e 1.18 presentano l'esterno e l'interno della MGU in questione.

Fig. 1.16: MGU per il park-lock

Fig. 1.17: MGU smontata per esporne i componenti interni

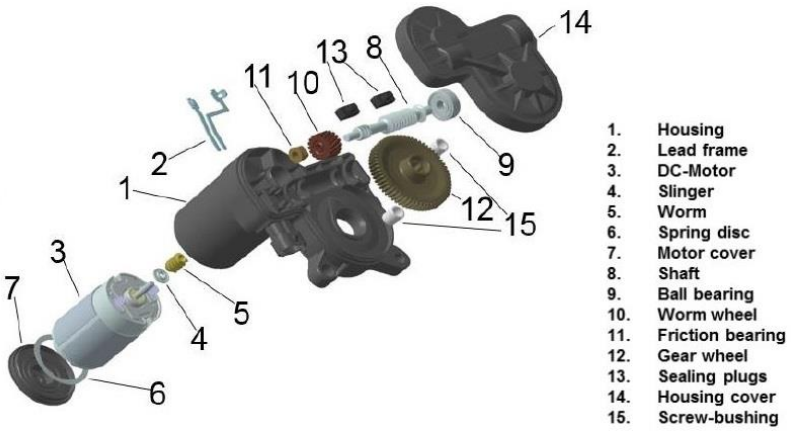


Fig. 1.18: esploso CAD della MGU del park-lock

Nell'esploso di figura 1.18 si osserva la presenza di un piccolo motore elettrico DC a spazzole (*brushed*); i suoi avvolgimenti sono alimentati mediante i due pin del connettore femmina presente sull'*housing*. La potenza meccanica prodotta dal motorino viene trasferita all'uscita mediante un ingranaggio vite senza fine – ruota a denti elicoidali (*worm gear*) a due stadi: il primo corrisponde all'ingranamento tra il pignone/vite posto all'estremo dell'alberino del rotore e la ruota dell'albero intermedio, il secondo all'accoppiamento tra pignone/vite dell'albero intermedio e la ruota condotta di output, che a sua volta si unisce mediante scanalato all'alberino di ingresso del *park-lock.*

La scelta progettuale di impiegare un doppio *worm gear* nella MGU non è casuale ma deriva da due caratteristiche [15-16] che distinguono questo ingranaggio da uno tradizionale e che sono molto utili per questa applicazione:

1. La possibilità di realizzare senza problemi legati a sollecitazioni ed ingombri rapporti di riduzione input/output molto elevati, dell'ordine delle decine sul singolo stadio. Nel caso della MGU considerata, il rapporto tra la velocità di ingresso (del rotore del motore *brushed*) e quella di uscita (della ruota condotta con scanalato femmina) è pari a 351; siccome il rendimento meccanico si aggira intorno al 30%, la coppia disponibile in uscita è $351 * 0.3 \cong 105$ volte quella in ingesso data dal motore. Si sottolinea che il rendimento molto basso,

provocato dall'elevato attrito tra i denti in presa, è tipico di questa tipologia di ingranaggio e ne costituisce uno dei limiti principali.

2. L'irreversibilità della direzione di trasmissione del moto quando si impiegano elevati rapporti di riduzione: il notevole attrito che si sviluppa tra il profilo della vite e quello dei denti della ruota elicoidale rende impossibile per la seconda mettere in rotazione la prima (entro un certo limite di carico). I vantaggi introdotti dall'applicazione di questo meccanismo auto-bloccante alla movimentazione del *park-lock* sono evidenti; è infatti possibile rimuovere l'alimentazione al motore (e quindi non usare energia elettrica) quando il *park-lock* ha raggiunto la configurazione desiderata (innesto o disinnesto), con la sicurezza che il dispositivo non si muoverà fino a quando non verrà fornito volutamente un nuovo comando.

Per controllare il moto del *park-lock* e soprattutto per implementare sistemi diagnostici in grado di riconoscere *fault* potenzialmente dannosi per gli utenti del veicolo, soprattutto in relazione all'analisi FMEA (*Failure Mode and Effect Analysis*), si richiede la presenza di un adeguato sensore di posizione. Nell'applicazione considerata è stato utilizzato un sensore ad effetto Hall: trattasi di un trasduttore in grado di rilevare un campo magnetico (generato da un magnete o da un elettromagnete) e che produce un segnale di uscita con caratteristiche che dipendono dal modulo di una delle componenti del campo magnetico stesso.

Il sensore Hall di posizione utilizzato (fig. 1.19) è dotato di un connettore integrato a tre pin, ha un foro liscio per il passaggio del bullone di collegamento alla carcassa della trasmissione elettrica e, infine, presenta un *o-ring* per evitare il trafilamento di olio.

Fig. 1.19: sensore e vista del connettore integrato

Il sensore è montato sulla carcassa dallo stesso lato della MGU e ha una testa che si estende nella scatola per intercettare il campo magnetico prodotto dal magnete della leva porta-magnete; l'output del trasduttore è costituito da un segnale in PWM (*Pulse Width Modulation*) caratterizzato da frequenza portante di 1000 Hz e il cui *duty cycle* (che sarà spesso abbreviato *dc*) varia linearmente (teoricamente e in prima approssimazione) con l'angolo di rotazione della leva, entro una certa corsa dell'albero. Infine, i tre pin del connettore integrato servono per l'alimentazione a 5V, per la massa e per il PWM di uscita.

1.3 Il DVP della trasmissione per veicolo elettrico

DVP è l'acronimo di *Design Validation Plan* e indica il piano di validazione definito per il design di un prototipo. Nell'ambito automotive, l'avvio dell'industrializzazione e della produzione di una nuova trasmissione meccanica è sempre preceduto da diverse fasi di validazione prototipale. Infatti, è praticamente impossibile che il primo design del prodotto, definito in fase progettuale, sia in grado di rispettare tutti i requisiti concordati con il cliente (in termini di resistenza statica, fatica, lubrificazione, rumorosità, …) e, in ogni caso, sarebbero necessari dei test sperimentali per provarlo. Per questo motivo, la soluzione iniziale viene concretizzata in una serie di prototipi uguali, i quali affrontano una serie di prove con lo scopo di evidenziare gli aspetti da correggere; una volta definite le modifiche al design da implementare, esse vengono convogliate nel design di una nuova fase prototipale, la quale sarà caratterizzata con test più o meno analoghi a quelli della

fase precedente. Si tratta di un processo iterativo che termina quando viene individuato un design che supera tutta la procedura di validazione: a questo punto la soluzione tecnica viene 'congelata' (almeno momentaneamente) e inoltrata per l'industrializzazione.

Le indicazioni operative per l'esecuzione delle prove di fatica del *park-lock* sono state tratte dal documento che presenta il DVP dell'ultima fase prototipale del riduttore per trasmissione elettrica che è stato presentato in precedenza. La maggior parte delle prove di validazione descritte nel documento riguardano la trasmissione in sé; però, un intero capitolo è riservato alla presentazione dei test che il *park-lock* deve affrontare: questo fatto evidenzia quanto il funzionamento del dispositivo di sicurezza venga considerato importante per la validazione dell'intera trasmissione. Tra le principali prove a cui il design del riduttore deve essere sottoposto si ritrovano le seguenti:

- Test di lubrifica, per verificare che tutte le utenze di lubrifica (cuscinetti, ruote dentate e guarnizioni) ricevano una sufficiente quantità di olio a diversi numeri di giri in ingresso.
- Misura del gioco angolare e della rigidezza torsionale, da confrontare con i requisiti cliente o, in loro mancanza, con valori di archivio della *Oerlikon Graziano*.
- Rilevazione dell'impronta di contatto tra le ruote dentate in diverse condizioni di carico, così da condividere i risultati con il team di calcolo e valutare eventuali modifiche della microgeometria.
- Misura delle coppie di trascinamento a varie velocità, da confrontare con valori di riferimento per prodotti simili.
- Test di rottura statica, al fine di valutare la massima coppia che può essere fornita in ingresso e l'elemento fusibile meccanico per entrambi i versi di rotazione del primario.

La figura successiva mostra i test definiti dal DVP per il solo *park-lock*.

Fig. 1.20: elenco delle prove del piano di validazione del park-lock OG

In riferimento alla fig. 1.20, le prove che l'ultima fase prototipale del dispositivo deve superare sono le seguenti:

1) ***Vehicle impact test***.

Questo test simula l'impatto tra il veicolo elettrico su cui verrà montata la trasmissione, parcheggiato e con il *park-lock* agganciato, e un secondo veicolo di uguale peso in moto con una certa velocità. La procedura di prova è la seguente: il dispositivo viene innestato e si applica al primario la coppia corrispondente alla collisione; dopo la prova, tutti i componenti del *park-lock* sono analizzati e la presenza di cricche viene esaminata mediante metodi non distruttivi.

2) ***Ratchet test***.

Si tratta di un test volto a verificare che il *park-lock*, a seguito della generazione del comando di innesto, agganci nel corretto range di velocità del veicolo (da 2 a 3 km/h); infatti, al di sopra di un certo valore non deve mai innestare (per impedire un blocco involontario dell'asse con conseguente perdita di controllo del veicolo), al di sotto di un secondo valore deve essere sicuramente innestato (perché sintomo di un'attivazione volontaria in previsione di un parcheggio). Il test viene realizzato per entrambi i versi del moto secondo queste modalità: il primario viene portato in rotazione ad una velocità relativamente elevata, si comanda l'innesto del *park-lock* e il primario è rallentato gradualmente finché non viene fermato dall'ingresso della testa dell'arpione in un vano: la velocità di aggancio è quella immediatamente antecedente allo stop; i valori ricavati nelle molteplici prove a banco vengono utilizzati per stimare statisticamente il range di aggancio del *park-lock*.

3) ***Max static torque test***.

Questo test consiste nell'applicazione di una rampa di coppia al primario mentre il dispositivo di sicurezza è agganciato, fino al punto in cui si ha la rottura di un componente (fusibile meccanico). Il risultato della prova (realizzata una volta in avanti e una all'indietro) corrisponde all'individuazione della massima coppia sopportabile dal *park-lock* e dell'elemento fusibile dell'intero sistema di trasmissione (riduttore con *park-lock*).

4) Oscillating torque test.

Tale prova deve accertare la capacità del componente di sicurezza di sopportare oscillazioni di coppia nella posizione di parcheggio, ossia innestata. Quindi, a *park-lock* agganciato, il primario viene sollecitato con una coppia oscillante in modo quasi sinusoidale tra due valori limite; vengono eseguiti due test, uno in avanti e uno all'indietro.

5) Spring fatigue test.

Si tratta del test di fatica della molla e verifica la sua capacità di raggiungere N cicli di carico senza mostrare deterioramento funzionale. L'inizio del ciclo prevede che la molla sia sottoposta al solo precarico, prosegue fino alla torsione massima di lavoro (condizione in cui l'attuatore raggiunge il finecorsa di innesto mentre l'arpione rimane a contatto con la testa di un dente del rocchetto) e termina con il ritorno all'angolo di precarico. Prima e dopo la prova di fatica viene eseguito un *ratchet* test, al fine di verificare che l'aggancio avvenga nel range di velocità predefinito. Questa prova viene eseguita due volte: la prima per raggiungere l'obiettivo di N cicli e la seconda fino a rottura.

6) Under voltage test.

Il test simula la manovra di disinnesto quando la batteria è a bassa tensione e viene eseguito una volta per senso di rotazione del primario.

*7) **Fatigue test.***

La prova di fatica del *park-lock* ha lo scopo verificare la durabilità del dispositivo nelle normali condizioni di lavoro e viene realizzata in accordo con i duty-cycle riportati nella tabella 1.1. I valori centrali della tabella sono stati oscurati per proteggere la riservatezza dei dati *Oerlikon Graziano*.

		Temperature [°C]						
Total cycles	**Slope [%]**	**-40**	**-20**	**0**	**23**	**50**	**70**	**90**
	33							
	15							
	8							
	3							
Cycle distibution for temperatures		**3%**	**6%**	**9%**	**20%**	**40%**	**17%**	**5%**

Tab. 1.1: tabella dei duty cycle della prova di fatica park-lock

In riferimento alla tabella 1.1, nella prima colonna da sinistra è riportato il numero di cicli da effettuare per ciascuna condizione di carico considerata; la seconda colonna da sinistra indica l'entità tali carichi e le altre colonne definiscono, per ogni carico, la distribuzione dei cicli su sette temperature significative. Si osserva che ciascun carico è espresso come pendenza percentuale: la coppia fornita al primario quando il *park-lock* innesta è pari a quella trasmessa dagli pneumatici quando veicolo elettrico è a pieno carico e parcheggiato su una strada con quella pendenza; è evidente, dunque, che ad una maggiore pendenza corrisponde un maggiore carico e il numero di cicli di prova decresce esponenzialmente all'aumentare della pendenza per questioni statistiche. Il test viene realizzato nella cella climatica così da seguire il profilo di temperatura di figura 1.21; anche in questo caso, le durate in minuti della varie fasi sono nascoste per riservatezza.

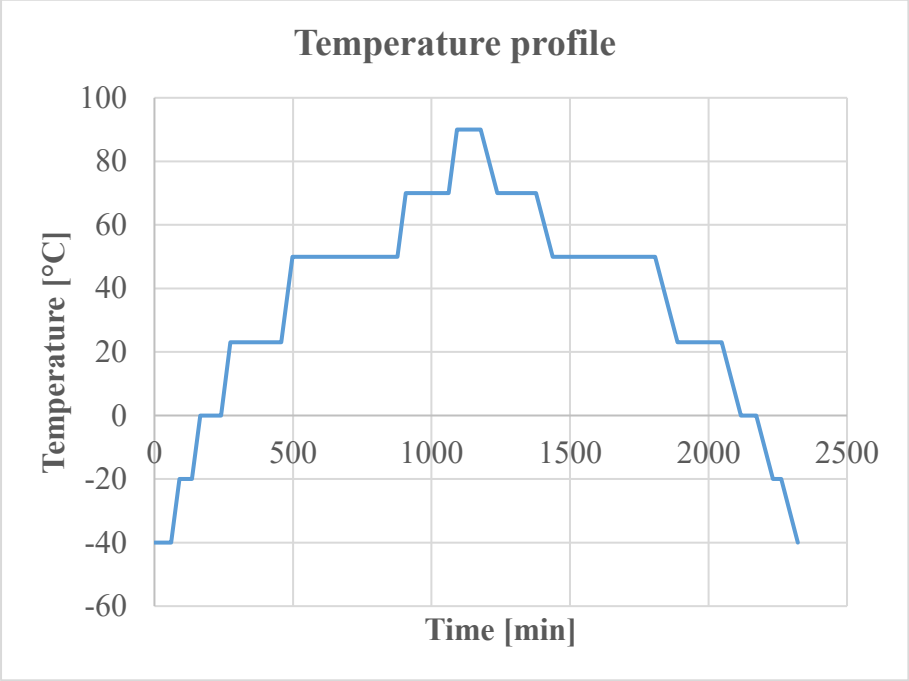


Fig. 1.21: profilo di temperatura per prova di fatica park-lock

Inoltre, per verificare la capacità della leva porta-magnete di sopportare il *creeping* e di sopportare cicli di sollecitazione prolungati, in parallelo al test di fatica precedente sono previste le seguenti prove:

1. Due riduttori sono posti nella cella climatica con il *park-lock* ingaggiato.
2. Due riduttori realizzano altrettante prova di fatica della sola leva da $3 * 10^5$ cicli ciascuna; per questo test l'attuatore compie ciclicamente la corsa intera e, al fine di sollecitare la leva il più possibile (*worst case*), il primario viene posto in una posizione per cui l'arpione può entrare completamente in un vano del rocchetto (condizione di minima coppia frenante sull'uscita della MGU).

Il fine ultimo delle attività di tesi è stato configurare due centraline elettroniche identiche per il controllo simultaneo di due *park-lock* e, in particolare, l'esecuzione automatizzata di prove di fatica relative al dispositivo. In particolare, le due centraline sono stati impiegate principalmente per gestire la prova di fatica molla da N cicli e le due prove di fatica leva da $3 * 10^5$ cicli; sono state inoltre

utilizzate per eseguire test ciclici di tenuta guarnizioni e durabilità MGU che non sono contenuti nel DVP presentato prima. L'immagine successiva mostra due prototipi di trasmissione per veicolo elettrico (uno fissato su una staffa e l'altro disposto a terra), sistemati all'interno della cella climatica, i cui *park-lock* stanno affrontando due delle prove di fatica contemplate dal DVP: quella della molla e una delle due della leva; ciascun dispositivo di sicurezza è controllato da una centralina da me configurata, con la quale comunica attraverso i cablaggi del sensore di posizione e dell'alimentazione.

Fig. 1.22: coppia di prototipi in cella per prove di fatica park-lock

1.4 Attrezzatura utilizzata

Ho realizzato tutte le attività legate alla tesi presso la sede di Rivoli della *Oerlikon Graziano*, nell'edificio adibito alle attività di *testing* sui prototipi; l'immagine seguente presenta quella che è stata la mia principale postazione di lavoro.

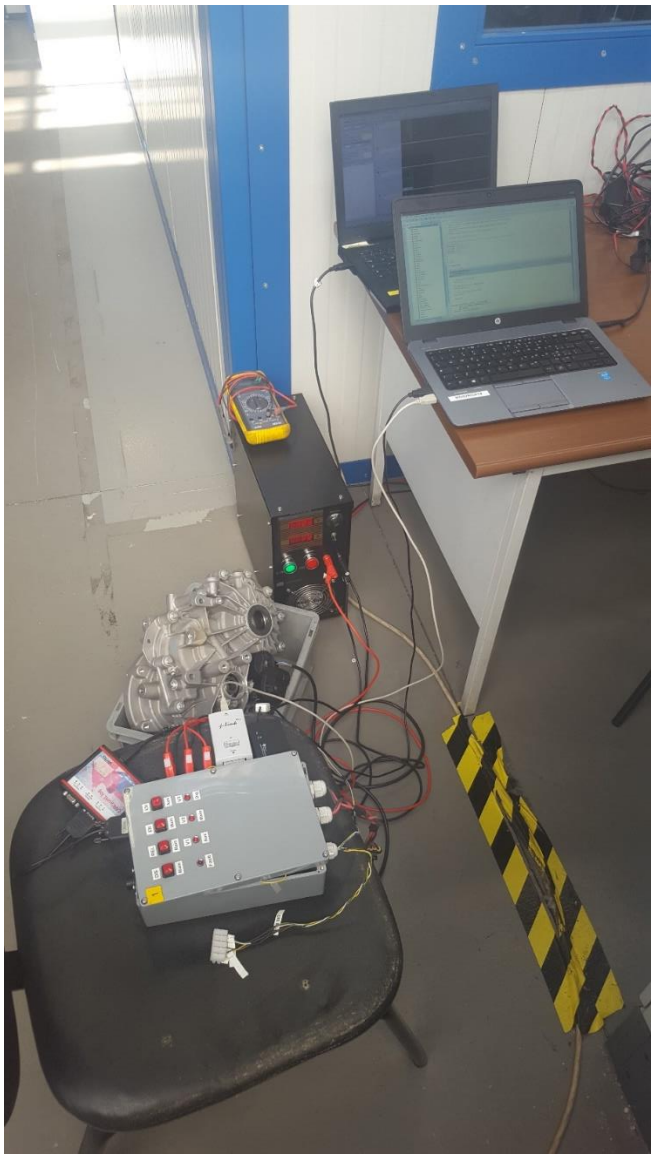


Fig. 1.22: attrezzatura della postazione di lavoro

Quindi vengono presentati i principali elementi della postazione:

- Prototipi di trasmissione per veicolo elettrico, dotati di primario con rocchetto e di assieme *park-lock* completo di MGU e sensore Hall.

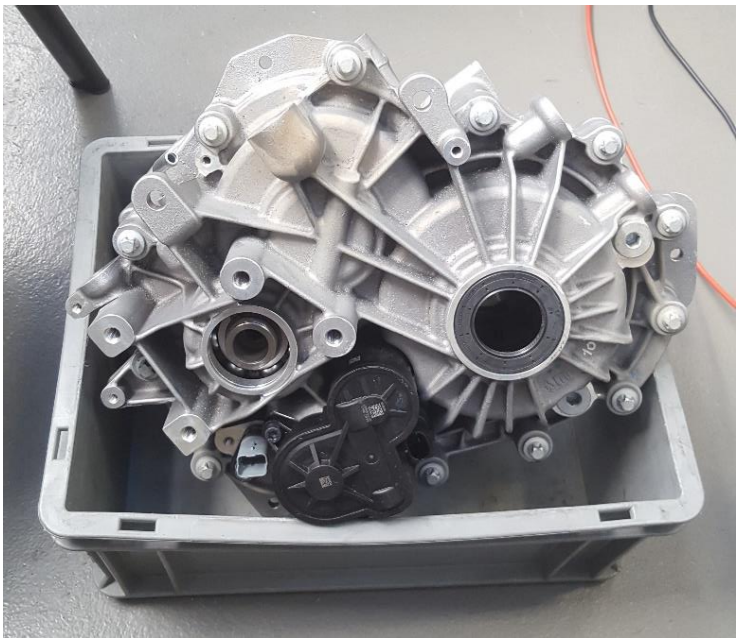


Fig. 1.23: prototipo della trasmissione elettrica

- Alimentatore in corrente continua, capace di fornire corrente mantenendo la tensione in uscita costante a 12V.

Fig. 1.24: alimentatore in corrente continua

- Due centraline per controllo di motori *brushed* DC (*Direct Current*).

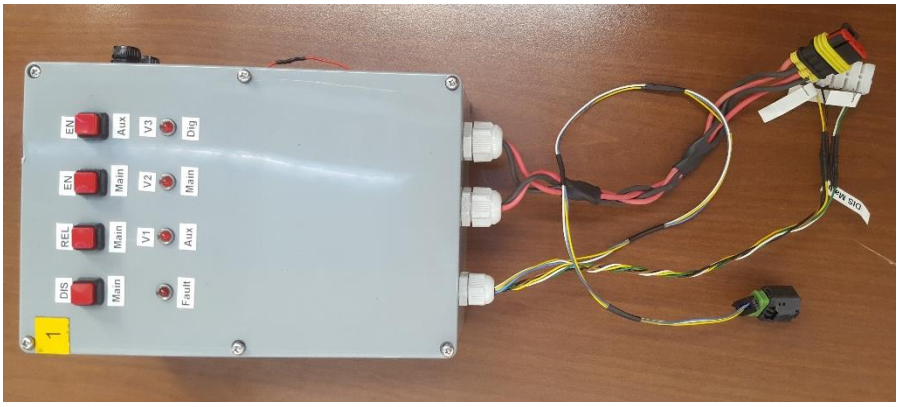


Fig. 1.25: una delle due centraline gemelle utilizzate

- Cavi per l'alimentazione delle centraline

Fig. 1.26: coppia di cavi per l'alimentazione di una centralina

- Cavi per l'alimentazione delle MGU impiegate.

Fig. 1.27: cavo per alimentare una MGU

- *Vector CANcaseXL Log* per l'interfacciamento tra il PC (*Personal Computer*) con il progetto *CANape* e le due centraline di controllo.

Fig. 1.28: Vector CANcaseXL Log

- Coppia di cavi CAN - CAN per collegare le due centraline al *CANcase*.

Fig. 1.29: cavo CAN-CAN

- Cavo USB-USB per il collegamento tra *CANcase* e PC.

Fig. 1.30: cavo USB-USB

- Sonda di programmazione *J-link* per i *core* delle centraline.

Fig. 1.31: sonda di programmazione core

- Sonda di programmazione *MSP-FET* per i *driver* della centralina.

Fig. 1.32: sonda di programmazione driver

- Due PC portatili: uno per usare gli IDE dei software delle schede elettroniche e l'altro per l'interfaccia di controllo delle prove.

Fig. 1.33: coppia di computer portatili

- Tester o multimetro.

Fig. 1.34: tester

- Scheda *Arduino*-clone.

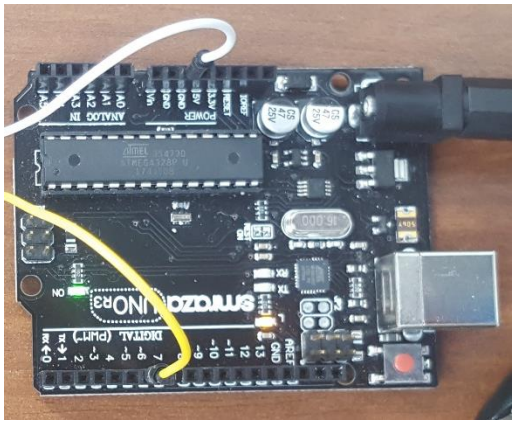


Fig. 1.35: scheda Arduino-clone

CAPITOLO 2

La centralina di controllo park-lock

In questo capitolo verranno trattati gli argomenti più lontani da quelli caratteristici del corso di studi in Ingegneria Meccanica che ho frequentato, ossia le tematiche relative all'ambito elettronico ed informatico. A seguito di una prima parte di contestualizzazione inerente il controllo del *park-lock* a bordo veicolo (*on board*), ci si soffermerà sulla descrizione sintetica dei componenti della centralina impiegata per l'esecuzione dei test e del loro funzionamento. Successivamente, verrà descritta e commentata la struttura generale dei progetti sorgente utilizzati per produrre i *firmware* da caricare sulle schede elettroniche della centralina e si procederà, in particolare, alla schematizzazione del loro funzionamento ciclico mediante diagrammi a blocchi.

2.1 Il controllo on board del park-lock

In generale, con il termine 'centralina' si intende un qualsiasi dispositivo elettronico a cui fanno capo uno o più altri componenti elettronici e/o elettromeccanici per quanto riguarda l'alimentazione, il controllo e/o il funzionamento [17]; si tratta di una definizione alquanto generale ed è facile comprendere che questo genere di dispositivo (chiamato anche ECU, *Electronic Control Unit*) è largamente utilizzato in tutti i contesti lavorativi (in particolare in quelli industriali) in cui si desidera, ad esempio, far realizzare in modo controllato e/o automatizzato operazioni specifiche ad elementi dotati di una certa componente elettronica. Ciononostante, comunemente quando si sente nominare il termine 'centralina' si pensa immediatamente all'ambito dell'autoveicolo (a cui appartiene anche il *park-lock*), poiché è nota l'abbondanza di questo genere di componente e la loro importanza, soprattutto dal punto di vista della sicurezza e delle prestazioni.

La maggior parte dei veicoli (da quelli commerciali a quelli più sportivi) prodotti oggi in serie sono dotati di un elevato numero di sistemi elettronici di controllo, alias di centraline [18]. L'incremento della quantità di elettronica presente nei prodotti dell'industria automotive è da attribuire a due fattori concorrenti:

- Da un lato, la continua crescita delle richieste da parte dei clienti, che esigono maggiore sicurezza e maggiore comfort durante la guida.
- Dall'altro, l'incremento del numero e della severità delle leggi che mirano a ridurre la produzione delle sostanze nocive dei gas di scarico e a diminuire il consumo di combustibile.

Le centraline elettroniche a cui viene assegnato l'adempimento di questi compiti in modo automatizzato sono da lungo tempo utilizzate nel campo del controllo dei motori endotermici, così come in quello delle trasmissioni; sono fondamentali, inoltre, nell'implementazione del sistema antibloccaggio (ABS) e di quello antislittamento (ASR). L'immagine successiva riporta un esempio di moderna architettura di comunicazione tra i sottosistemi fondamentali di un moderno veicolo [18].

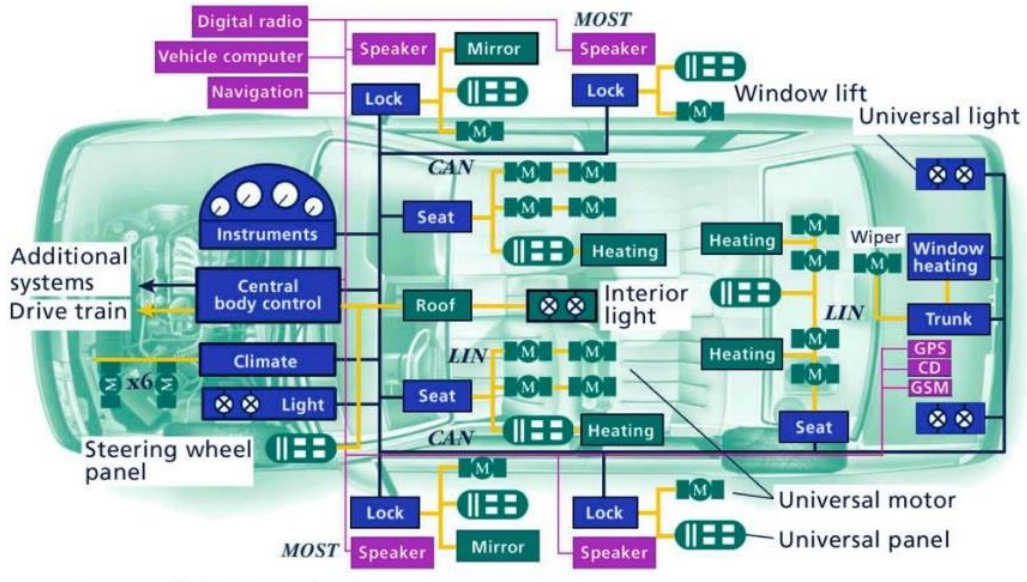


Fig. 2.1: tipica architettura di comunicazione interna tra i dispositivi di un moderno veicolo

Il *park-lock Oerlikon Graziano* è un dispositivo di sicurezza azionato elettro-meccanicamente che, per le sue delicate caratteristiche strutturali e funzioni, necessita dell'interfacciamento con un'opportuna unità di controllo *on-board*; tale entità si deve occupare principalmente del controllo delle fasi di attuazione e del continuo monitoraggio del funzionamento dispositivo. La componentistica hardware che la implementa può essere realizzata in una centralina a sé stante oppure, come avviene frequentemente, integrata in una delle molteplici unità di controllo a bordo veicolo; ad esempio, nel caso nella trasmissione per veicolo elettrico su cui il *park-lock* testato è montato, la circuiteria di controllo è integrata nell'inverter per questioni di praticità produttiva. La figura 2.2 rappresenta una possibile soluzione [19] per l'interfacciamento tra i vari dispositivi elettronici ed elettromeccanici che contornano una trasmissione automatica dotata di *park-lock* ridondato sia nell'attuazione elettrica che nella rilevazione della posizione.

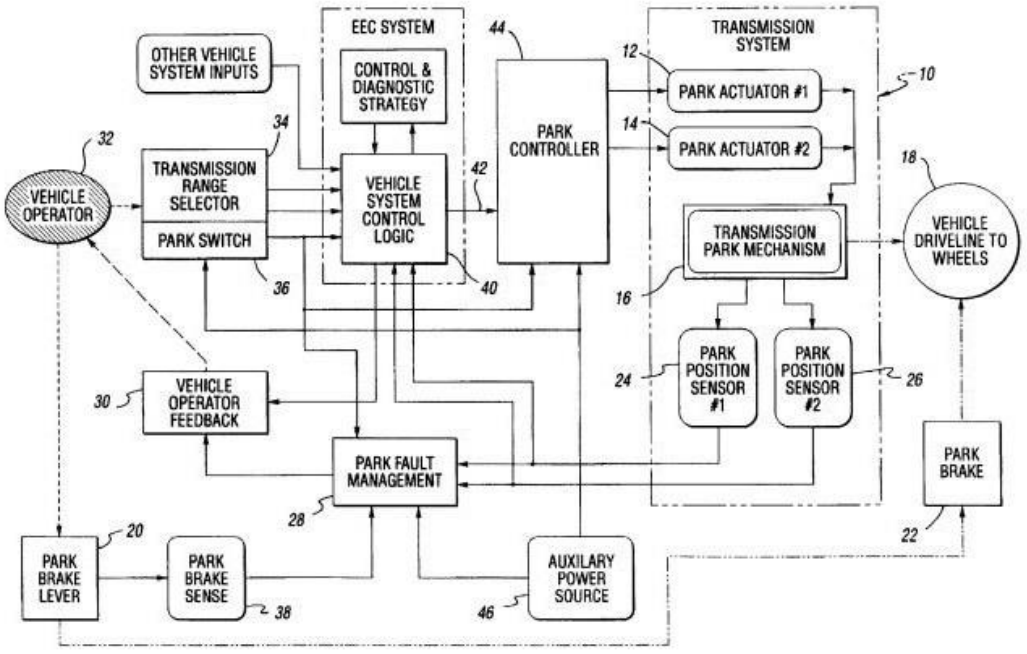


Fig. 2.2: possibile configurazione di un sistema on board di controllo park-lock

In riferimento alla figura precedente, si ipotizzi che l'automobile su cui il sistema in questione è implementato si sia appena fermata e il guidatore (32) sposti la leva del cambio (34) in corrispondenza della posizione di P (*Parking*). L'informazione relativa a questa azione dell'utente del veicolo viene trasmessa

elettronicamente (attraverso una opportuna rete di comunicazione) ad una designata unità di controllo *on-board* (40), la quale riceve il segnale e lo rimanda alla componentistica hardware a cui è demandato il controllo dell'attuazione del dispositivo di sicurezza (44). Di conseguenza, il controllore del *park-lock* applica un algoritmo di controllo che definisce l'andamento della tensione di comando dell'attuatore principale (12); quindi, i componenti della trasmissione del *park-lock* (16) vengono messi in movimento (16) fino al raggiungimento della condizione di innesto.

Il moto del *park-lock* viene seguito mediante un primo trasduttore di posizione (24), che è ridondato esattamente come l'attuatore. Il feedback di posizione può essere riportato direttamente all'unità di gestione *park-lock* nel caso di controllo in anello chiuso; però, è fondamentale la trasmissione del segnale ad una seconda unità (28) che si occupa del riconoscimento e della gestione dei *fault* inerenti al dispositivo di sicurezza. Questa ultima unità è fondamentale dal punto di vista della *safety*: se si registra un fenomeno associabile ad un errore di funzionamento e/o ad un danneggiamento di uno dei componenti (ad esempio, un valore di posizione fuori range oppure il mancato assorbimento di corrente da parte del carico), viene prodotto un segnale elettrico ad esso correlato; questo segnale attraversa la rete interna di comunicazione e raggiunge rapidamente un'unità (30) che avverte il guidatore del problema e il centro di controllo principale (40), il quale mette in atto la strategia migliore per evitare danni.

Si desidera sottolineare che sia l'unità a bordo veicolo di cui si è trattato poco sopra che il dispositivo elettronico da me configurato per la realizzazione dei test di fatica sono 'centraline per il controllo di dispositivi *park-lock*' a tutti gli effetti; però, l'applicazione specifica e l'ambiente che li circonda determina delle notevoli differenze tra una soluzione e l'altra. Per comprendere la questione basta tenere in considerazione che il *park-lock* e tutto il sistema di controllo a bordo veicolo devono essere certificati con livello ASIL pari a D. La sigla ASIL sta per *Automotive Safety Integrity Level* ed si riferisce ad uno schema di classificazione dei rischi definito dalla normativa ISO 26262 [20]. Il livello ASIL D corrisponde al riconoscimento del massimo grado di pericolo (inteso come rischio di

danneggiamento) associato al sistema in esame, tipicamente correlato alla possibilità di procurare ferite mortali; di conseguenza viene richiesto il più alto livello di garanzia nella definizione di *safety goals* sufficienti e soprattutto nel loro effettivo raggiungimento. La certificazione ASIL D è richiesta a tutti i sistemi di sicurezza che si trovano a bordo veicolo (fig. 2.3), incluso quello di controllo del *park-lock*.

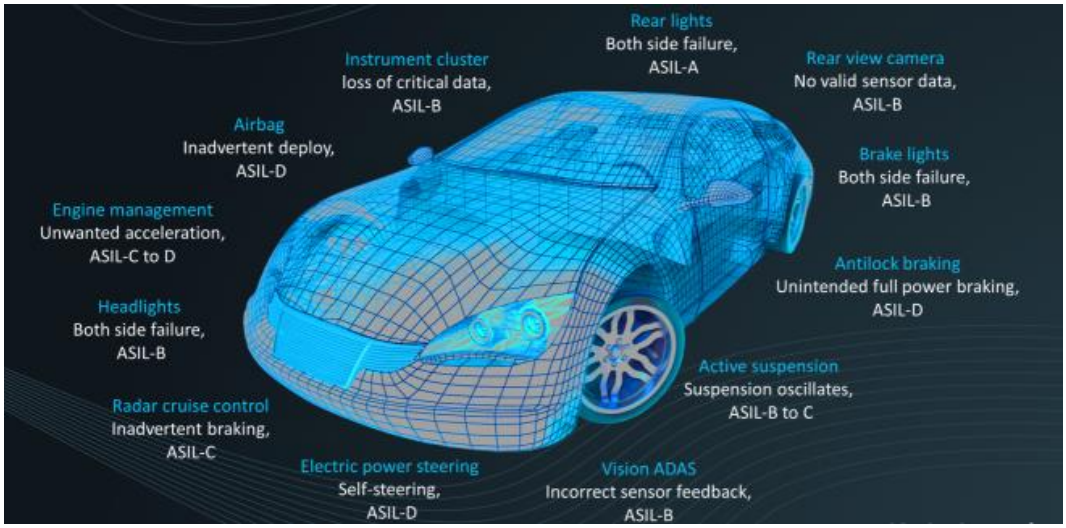


Fig. 2.3: distribuzione dei livelli ASIL richiesti per i dispositivi elettronici di un veicolo

A livello hardware, è evidente che le schede e i microprocessori utilizzati nella implementazione *on-board* sono molto più robusti, performanti e certamente compatti rispetto a quanto necessario per realizzare semplici test ciclici di aggancio e sgancio; inoltre, anche la rete di comunicazione *on-board* sarà più rapida e soprattutto *fault-tolerant* (ossia insensibile agli errori) rispetto a quella utilizzata per trasmettere le informazioni al PC di interfaccia per i test di fatica. Dal lato software, le soluzioni implementate nell'elettronica a bordo veicolo, in termini di algoritmo di controllo e di riconoscimento degli errori, non possono che essere più raffinate di quanto è possibile trovare su un'altra centralina; in particolare, per i motivi sopra citati, la validazione di un software per la gestione di un dispositivo di sicurezza a bordo veicolo è molto più severa e stringente di quella affrontata da un qualsiasi programma ad uso industriale.

2.2 I componenti hardware della centralina

Ciascuna delle due centraline prima analizzate e poi configurate per lo svolgimento delle attività di tesi è stata costruita e programmata (dal fornitore dell'azienda) per una specifica funzione: il controllo di un *park-lock* ridondato, caratterizzato da un'attuazione principale realizzata mediante un motorino a spazzole (*brushed*) DC e una secondaria ottenuta per mezzo di un solenoide. In realtà, le schede presenti possono essere riprogrammate e i vincoli reali di funzionamento si riducono alle specifiche dell'hardware e alle capacità di chi riprogramma le schede. In generale, i due *driver* a bordo della ECU consentono di azionare in modo indipendente e controllato due attuatori operanti in corrente continua a bassa tensione (12V). Evidentemente, la realizzazione del controllo di un *park-lock*, ridondato o non, non richiede unicamente un'adeguata struttura hardware (in termini di tipologia di schede utilizzate e di relativo cablaggio) ma anche l'integrazione nei codici sorgente di librerie e funzioni coerenti con il tipo di applicazione.

Per quanto riguarda la struttura generale, la centralina (mostrata aperta in fig. 2.4) è dotata di due schede *driver* che offrono la possibilità di controllare contemporaneamente due attuatori elettrici del tipo nominato; sono presenti, inoltre, una scheda *core* con *board* di espansione, la quale soprintende al funzionamento dei *driver* e alle interazioni tra sistema e utente esterno, e una schedina 'filata' che opera il condizionamento del segnale dal sensore di posizione e genera i 5V.

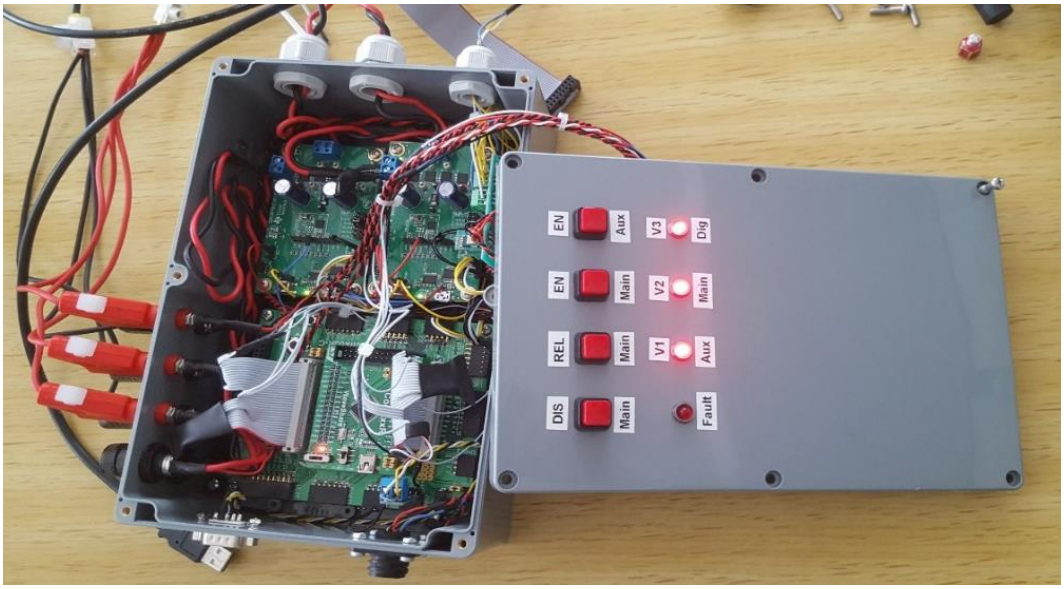


Fig. 2.4: centralina da riprogrammare aperta

Passando ad una descrizione di maggiore dettaglio, l'hardware della ECU si basa su due insiemi di componenti elettronici facilmente reperibili in commercio:

- Il kit di sviluppo *Open205R-C*, distribuito dalla *Waveshare* e mostrato in figura 2.5.

Fig. 2.5: kit di sviluppo Open205R-C

- Il modulo di valutazione *DRV8701EVM*, distribuito dalla *Texas Instruments* (TI) e presentato in fig. 2.6.

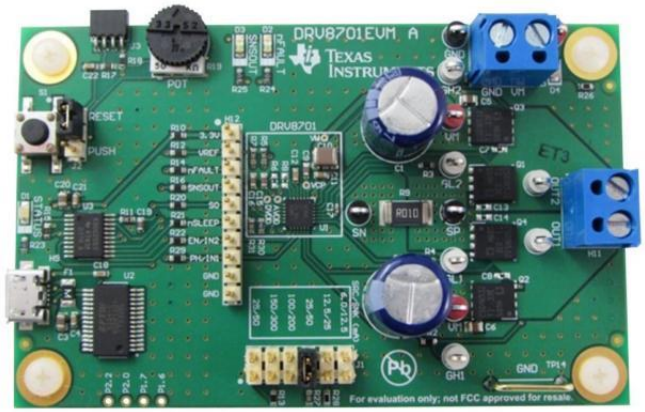


Fig. 2.6: modulo di valutazione DRV8701EVM

Il kit *Open205R-C* di fig. 2.5 appartiene alla famiglia dei kit di sviluppo (*developement board*) prodotti per semplificare l'interfacciamento e lo sfruttamento delle potenzialità da parte dell'utente programmatore dei microcontrollori di tipo *STM32*: trattasi di dispositivi a 32 bit prodotti dalla ST e basati sul processore *Arm Cortex-M*. In particolare, *Open205R-C* è costituito da due componenti ben distinti:

1) Una scheda *core* (*core board*) di nome *Core205R* (fig. 2.7), che costituisce l'alloggiamento del microcontrollore *STM32F205RBT6* (o MCU, *MicroController Unit*) dotato di processore *Cortex-M3*.

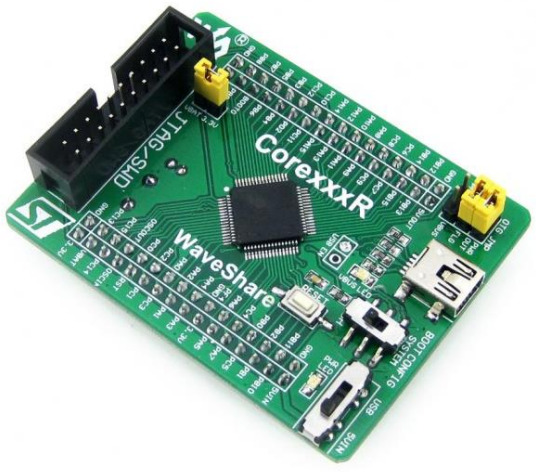


Fig. 2.7: core board Core205R

2) Una scheda madre (*mother board*), che possiede tutte le porte e i pin necessari per la comunicazione del *core* con le altre schede della centralina e con l'esterno.

Il microcontrollore *STM32F205RBT6* [21] è un dispositivo elettronico integrato caratterizzato da prestazioni elevate, garantite dal *core* a 32 bit che ne costituisce il cuore di calcolo e che può operare fino ad una frequenza di 120MHz. Inoltre, questa MCU presenta molte *features* che la rendono una scelta di primo livello tra i componenti appartenenti alla medesima categoria; ad esempio, è dotata di memoria *Flash* fino ad 1 MB, di tre convertitori A/D e due D/A da 12 bit, di 140 porte per I/O generici e 15 interfacce di comunicazione. La scheda *core* del kit *Open205R-C* rappresenta il 'cervello' della centralina, in quanto si occupa del coordinamento dei due *driver* e della gestione delle interazioni tra utente e centralina (ricezione comandi, attivazione led di *fault*, etc.); in particolare, tale scheda si occupa dell'adempimento di tre processi (*task*) eseguiti in parallelo:

1. Gestione della comunicazione con le schede *driver*.
2. Costruzione e trasmissione dei messaggi CAN.
3. Gestione degli *input* e degli *output* generici (GPIO).

Invece, la figura 2.6 rappresenta il modulo di valutazione (*evaluation module*) *DRV8701EVM* [22], indicato generalmente nella tesi come '*driver*' per le funzioni da esso svolte; si tratta di un componente elettronico nato con lo scopo di semplificare lo sfruttamento delle potenzialità del *driver* per *H-Bridge DRV8701*, prodotto (così come il modulo) dalla TI [23]. Il componente *DRV8701* è un amplificatore di potenza: esso converte, infatti, il segnale di bassa potenza proveniente da un IC (*Integrated Circuit*) nella corrente elevata (fino a qualche A) necessaria per commutare rapidamente i MOSFET di potenza del ponte H posto a valle. Questo *driver* può guidare i 4 MOSFET di tipo *N-channel* di un singolo *H-Bridge* e la sua applicazione tipica corrisponde al pilotaggio di un motore DC a spazzole bidirezionale, alimentato con tensione nel range 12-24V; si tratta, comunque, di un dispositivo molto versatile e che trova applicazioni anche

nell'ambito della robotica, della domotica, delle macchine utensili e del controllo di pompe e valvole industriali.

La figura 2.8 schematizza l'interazione tra i diversi elementi del modulo *DRV8701EVM* [22] nel caso in cui questo venga utilizzato come componente a sé stante, non asservito ad una seconda scheda di comando (come invece avviene nella ECU analizzata).

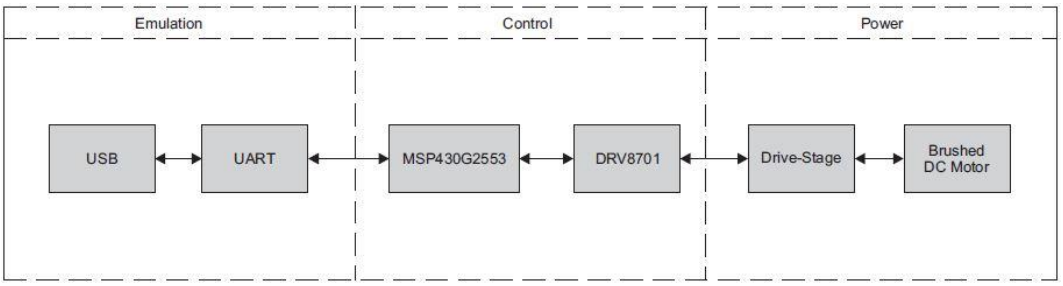


Fig. 2.8: schema a blocchi del funzionamento del DRV8701EVM

Il primo blocco sulla sinistra ('*Emulation*') identifica la porta seriale USB per la comunicazione con l'utente esterno e il microprocessore che si occupa della conversione dei messaggi da protocollo USB a protocollo UART. Il blocco centrale ('*Control'*) include il microprocessore *MSP430G2553* [24], il 'cervello' del modulo, e il *driver DRV8701* precedentemente descritto; è evidente l'importanza tra questi due elementi: il primo elabora un programma che conduce alla definizione di un riferimento per l'azione a valle del secondo e, viceversa, il secondo trasmette al primo informazioni sul funzionamento dell'hardware a valle e sulla comparsa di eventuali errori. L'ultimo blocco ('*Power'*) vede il ponte-H ('*Drive Stage*') ricevere il comando PWM dal *driver* e determinare, a sua volta, la tensione di alimentazione del motore *brushed* DC.

La centralina configurata per la realizzazione delle prove di fatica è stata sviluppata per il controllo in contemporanea di due attuatori elettrici e per ciascuno di essi è prevista l'associazione ad una scheda *driver* dedicata al fine del controllo. È vitale sottolineare che i *driver* che compongono la centralina sono schede 'intelligenti', ovvero dotate di microprocessore programmabile (MSP430G2553) su cui viene implementato, principalmente, l'algoritmo di controllo dell'attuatore.

La scelta di dotare l'ECU di schede programmabili riduce sia il quantitativo di calcoli richiesti al *core* che la quantità di dati trasmessi tra le schede e ciò rende il sistema più rapido ed efficiente rispetto ad una soluzione centralizzata; alla scheda *core* rimane, così, unicamente la trasmissione dei comandi prodotti dall'utente e la raccolta delle informazioni utili relative al *core*.

L'elettronica della centralina viene completata da una scheda 'filata' (fig. 2.9), ossia una scheda priva di microprocessore, che assolve a due compiti molto importanti:

- Genera la tensione di 5V per l'alimentazione del sensore e quella di 3.3V richiesta dalle schede.
- Converte il segnale in PWM proveniente dal sensore di posizione in un segnale analogico trasmesso poi sia al *core* che ai due *driver* (condizionamento del segnale).

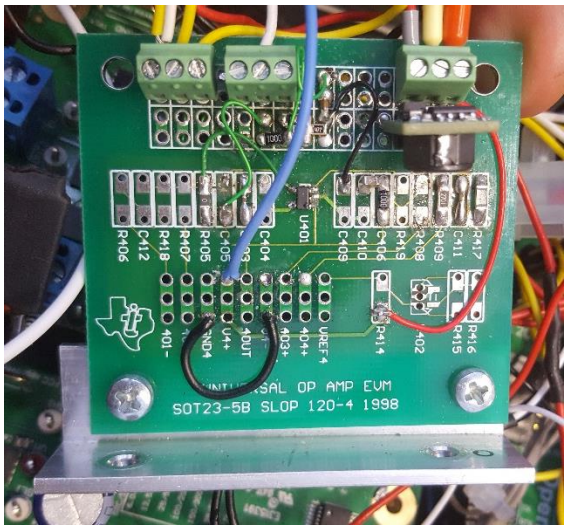


Fig. 2.9: scheda 'filata'

L'immagine successiva schematizza la distribuzione della potenza e la configurazione della comunicazione interna alla centralina.

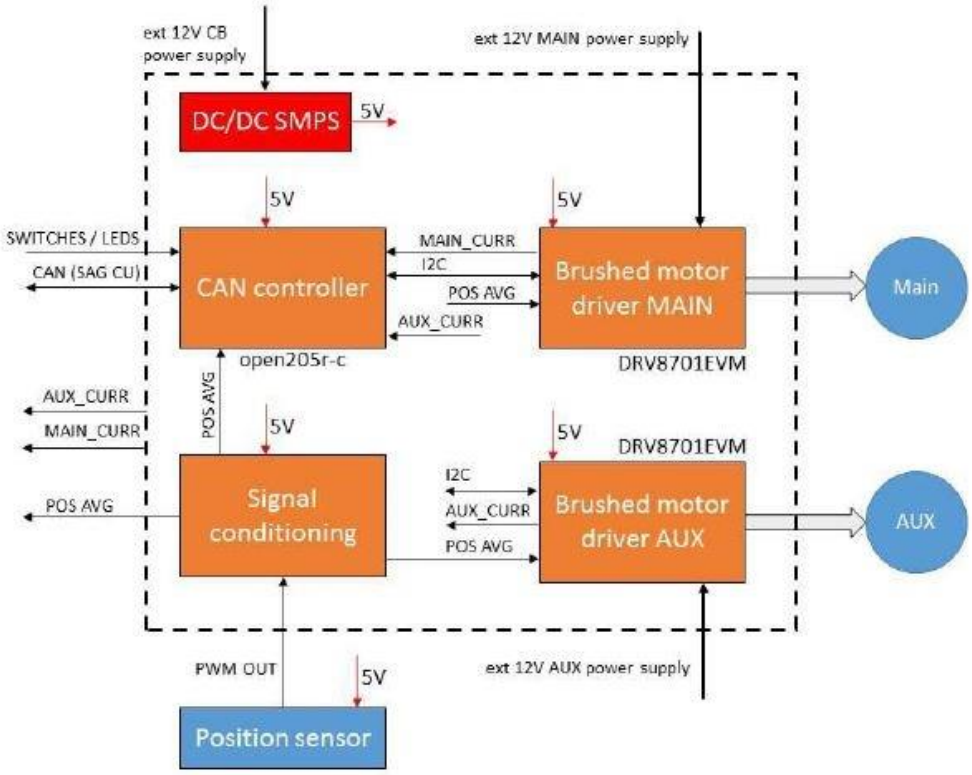


Fig. 2.10: distribuzione della potenza e vie di comunicazione della centralina

In riferimento alla figura 2.10, i rettangoli colorati in arancione rappresentano, da quello in alto a sinistra e in senso orario, i seguenti nuclei di funzionamento: la scheda *core*, i due *driver* e la parte di scheda filata che opera il condizionamento del segnale. Il rettangolo rosso in alto corrisponde all'elemento SMPS (*Switched-Mode Power Supply*) che produce i 5V (e i 3.3V mediante partitore di tensione), mentre quello azzurro in basso identifica il sensore di posizione. Infine, i due cerchi azzurri sulla destra indicano i due attuatori elettrici comandati e controllati per mezzo della centralina; questi due sono indicati con 'MAIN' ed 'AUX' poiché essa è stata originariamente programmata per controllare un'unica MGU per *park-lock* dotata di un motorino elettrico per l'attuazione principale (MAIN) e di un solenoide ausiliario (AUX), utilizzato solo in caso di *failure* del precedente.

I 5V generati dalla scheda filata alimentano le tre schede con microprocessore ed il sensore di posizione. Il *core* comunica con entrambi i *driver* mediante protocollo I2C (*Inter Integrated Circuit*), rispetto al quale il primo agisce da *master*

mentre i secondi operano da *slave*; ciò vuol dire che è il *core* a decidere quando ciascun *driver* può ricevere/trasmettere dati lungo il bus di comunicazione seriale [25]. Inoltre la scheda *core*, come già anticipato, si occupa della trasmissione/ricezione dei messaggi CAN e dei segnali di attivazione di led e *switch*. Ciascun *driver* è collegato all'alimentazione a 12V e comanda l'attuatore elettrico associato attraverso una coppia di morsetti di uscita. Infine, tutte le schede 'intelligenti' ricevono il segnale di posizione condizionato prodotto dalla scheda filata: i *driver* lo utilizzano nell'algoritmo di controllo e per l'identificazione dei *fault*, invece il *core* lo inserisce nel messaggio CAN di uscita.

L'immagine seguente illustra schematicamente le vie di comunicazione della centralina con l'esterno.

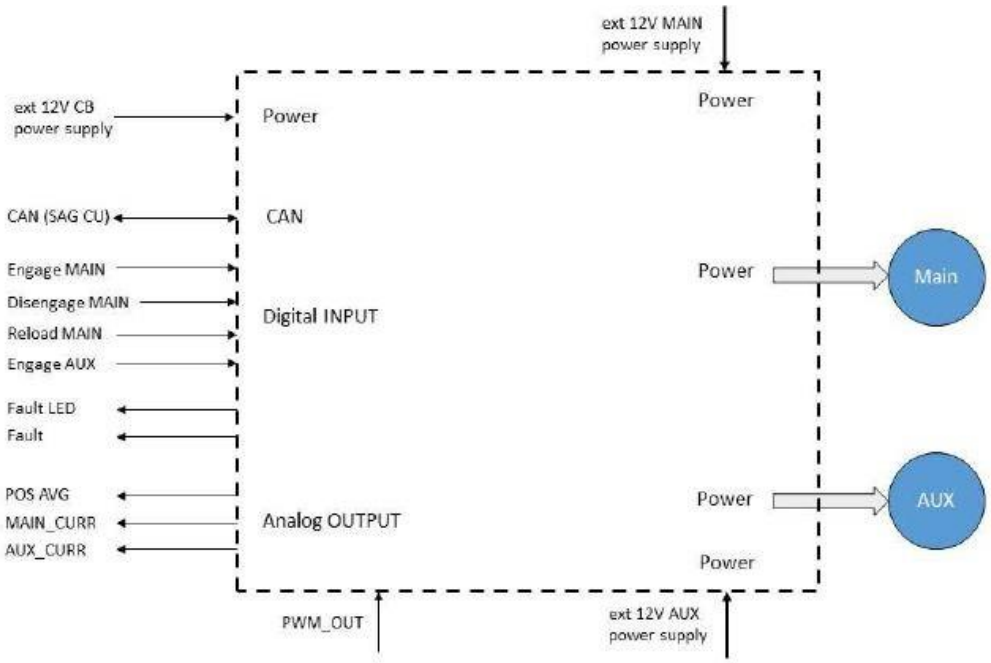


Fig. 2.11: connessioni esterne della centralina

In riferimento alla figura 2.11, a destra si osservano i collegamenti di ingresso a 12V richiesti dai *driver* e quelli di uscita per il comando degli attuatori. Dalla parte sinistra si notano invece, dall'alto verso il basso:

- L'ingresso dei 12V di alimentazione della scheda filata.
- La connessione bidirezionale per i messaggi CAN.

- Gli ingressi digitali collegati ai pulsanti sulla scatola.
- Le uscite digitali per l'accensione dei led di errore e di presenza tensione.
- Le uscite analogiche 0-3.3V per i segnali di errore, di posizione e di corrente.
- L'ingresso del segnale PWM generato dal sensore Hall.

La figura 2.12 mostra il pannello di controllo con i pulsanti di comando e i led di errore; invece, la figura 2.13 permette di osservare il connettore per la comunicazione CAN (a sinistra) e quello per la lettura delle uscite analogiche (a destra).

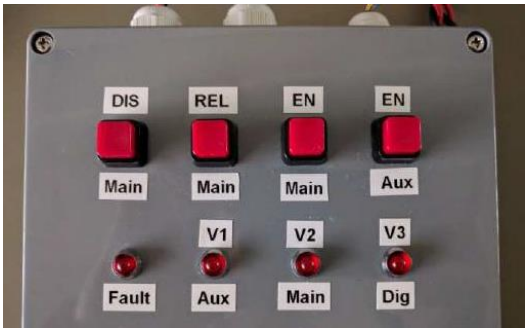


Fig. 2.12: pannello di controllo della centralina

Fig. 2.13: connettori per CAN e segnali analogici

La figura 2.12 permette di spiegare la funzione di default (ossia implementata nei sorgenti originali) dei quattro pulsanti presenti sul pannello:

- *DIS Main*: comanda il disinnesto del *park-lock* per azione dell'attuatore principale.

- *REL Main*: comanda il *reload* del *park-lock* per azione dell'attuatore *MAIN*; si tratta dell'aumento della torsione della molla a dispositivo già innestato grazie ad un'ulteriore rotazione dell'alberino nel verso di innesto. La struttura attuale del *park-lock* non consente la rotazione desiderata a causa del contatto leva-finecorsa e quindi premere questo pulsante non porta ad alcun effetto utile.
- *EN Main*: ordina l'innesto del *park-lock* ad opera dell'attuatore principale.
- *EN Aux*: aziona il solenoide ausiliario che dovrebbe portare al disinnesto o all'innesto di emergenza del *park-lock*. Poiché il solenoide esercita la forza in modo indipendente dal verso con cui la corrente percorre i suoi avvolgimenti, è presente un unico pulsante adibito alla sua attuazione.

2.3 L'architettura software della centralina

Per consentirmi di apportare modifiche al funzionamento della centralina in modo autonomo, è stato necessario richiedere all'azienda fornitrice del dispositivo elettronico di consegnarci anche i sorgenti delle schede utilizzate. In informatica, con l'espressione 'codice sorgente', o semplicemente 'sorgente', si intende il testo dell'algoritmo di un programma scritto in un certo linguaggio di programmazione e compreso all'interno di un file sorgente [26]. Il codice sorgente di una scheda, pertanto, incorpora in modo strutturato e codificato tutte le istruzioni che devono essere realizzate ordinatamente dal microprocessore affinché la scheda assolva correttamente al suo scopo. Un sorgente (o meglio un insieme di sorgenti) deve subire delle elaborazioni per essere convertito in un programma eseguibile in modo diretto e ciclico dal relativo processore; in generale, la trasformazione da codice sorgente ad eseguibile incorpora vari passaggi quali la precompilazione, la compilazione, il *linking* e infine il caricamento sull'hardware.

Tornando al caso specifico della centralina, ci sono stati forniti due progetti sorgente scritti nei linguaggi C e C++: uno per il *core* e un altro per i due *driver*. Per modificare i file di codice e gestire le fasi precedentemente nominate, relative

al pre-processamento dei sorgenti, mi sono servito di due ambienti di sviluppo integrato (IDE, *Integrated Developement Environment*) molto diffusi ed interamente gratuiti: IAR per il *core* e Code Composer Studio (CCS) per i *driver*, l'utilizzo dei due programmi ha richiesto notevole sforzo di apprendimento e la lettura dei corrispondenti manuali di utilizzo. [27][28]

Fig. 2.14: loghi di IAR (sinistra) e CCS (destra)

Progetto sorgente della scheda core

L'immagine successiva presenta l'interfaccia grafica di IAR, l'ambiente di sviluppo integrato utilizzato per la gestione dei sorgenti associati al *core*.

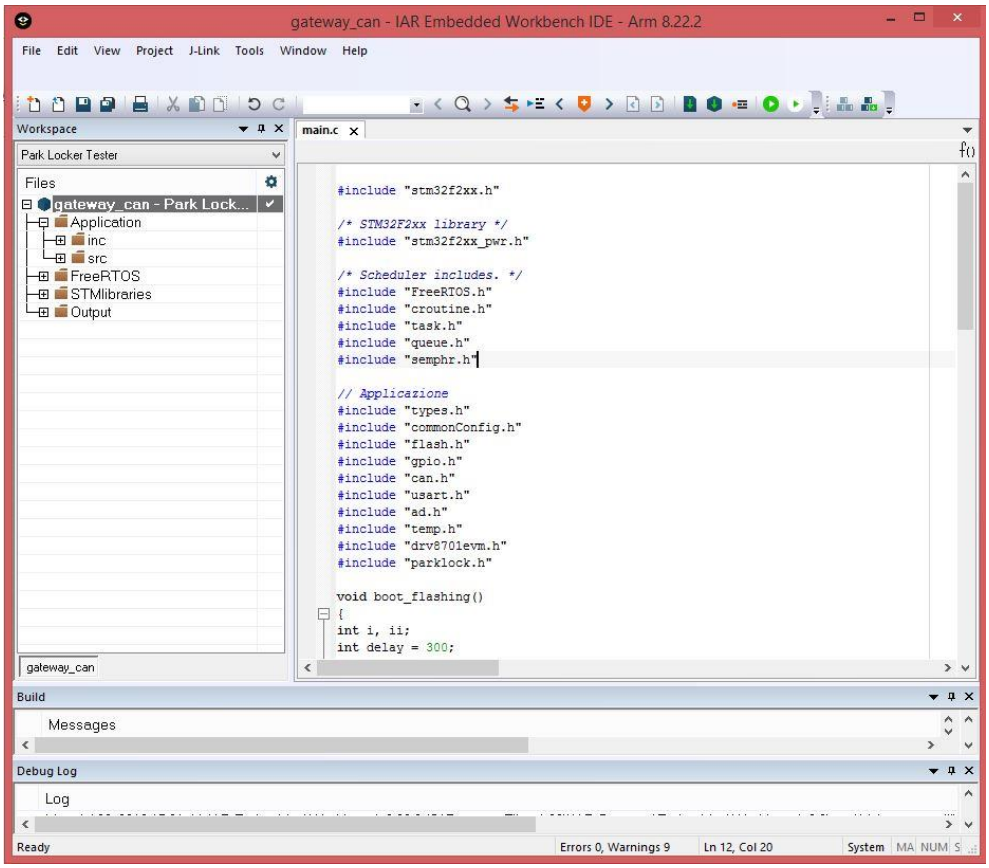


Fig. 2.15: interfaccia grafica IAR

Senza entrare nei dettagli della struttura dell'interfaccia di figura 2.15, poiché si ritiene tale azione inutile in questo contesto di sintesi, sulla sinistra è presente una finestra con nome *workspace*, ossia 'spazio di lavoro', nella quale si possono inserire uno o più progetti con cui si desidera operare; con il termine 'progetto' si fa riferimento ad un insieme di sorgenti, librerie, etc., anche racchiusi in cartelle differenti, che sono necessari per la produzione di un determinato eseguibile. In fig. 2.16 l'unico progetto aperto nel *workspace* è il *gateway_can*, ossia il progetto che ingloba tutti i file che servono a produrre l'eseguibile del *core*.

L'immagine seguente permette di visualizzare le cartelle componenti il progetto del *core*.

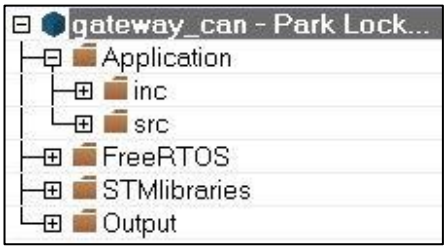


Fig. 2.16: albero del progetto sorgente del core

Secondo figura 2.16, il progetto include le seguenti cartelle:

- *Application*: racchiude i file sorgente (*src*) e gli *header (inc)* che descrivono l'algoritmo che deve essere eseguito ciclicamente dal *core*. Il nome della cartella deriva dal fatto che tali file sono strutturati ed organizzati in un modo che dipende dalla specifica applicazione, in termini di hardware impiegato e funzioni che si desidera realizzare.
- *FreeRTOS*: insieme di file che implementano un sistema operativo gratuito, fondamentale per le operazioni di inizializzazione e la gestione dei *task* del *core*.
- *STMlibraries*: raccolta di librerie standard utili.
- *Output*: cartella in cui viene salvato l'eseguibile.

I sorgenti (caratterizzati dall'estensione '.c') presenti nella sottocartella *src* di *Application* si possono osservare nella figura successiva.

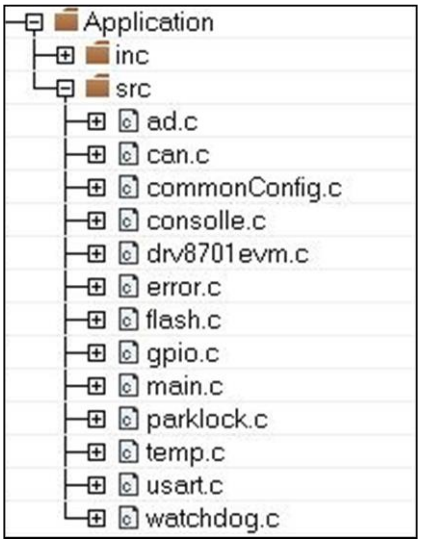


Fig. 2.17: sorgenti dell'applicazione del progetto core

Per prima cosa è opportuno analizzare la struttura di *main.c*, che permette di comprendere la struttura dell'intero progetto. In generale, negli insiemi di sorgenti agglomerati per arrivare alla creazione di un singolo eseguibile, il sorgente con nome *main* è il più importante, in quanto è il primo a venire considerato nell'ambito della compilazione e quindi detta la struttura dell'eseguibile finale. La figura seguente (2.18) mostra il contenuto della funzione *main* in *main.c.*

```
int main(void)
{

// 1. Inizializzazione delle periferiche/funzionalità
    commonConfig_init();
    flash_init();

    gpio_init();
    CAN0_init();

    USART_Configuration();
    USART_NVIC_Config();

    I2C_Configuration();

    ad_init();

    printf("\n\rHello!");
    boot_flashing();

// 2. Creazione dei task
    gpio_createTask();
    can_createTask();
    parklock_createTask();

#ifdef OS_DEBUG
    Free_heap_space = xPortGetFreeHeapSize();
#endif /* OS_DEBUG */

    /* Start the scheduler */
    vTaskStartScheduler();

    for(;;);
}
```

Fig. 2.18: corpo del sorgente main.c

La prima metà del codice riguarda la chiamata di una serie di funzioni di inizializzazione che servono a configurare opportunamente, ad esempio, la comunicazione CAN (*CAN0_init*) oppure quella I2C (*I2C_init*) o ancora il funzionamento del convertitore analogico-digitale del *core* (*ad_init*). Quindi, vengono richiamate le funzioni che 'creano' (nel senso di configurano) e descrivono i processi che devono essere ciclicamente realizzati dal *core*. Per processo (o *task*), in informatica, si intende "il compito specifico di un programma applicativo, di una procedura o di una sequenza di istruzioni del sistema operativo; in particolare, in un sistema operante in multiprogrammazione, ciascuno dei programmi o delle funzioni contemporaneamente attivi o attivabili" [29]. Nel caso del progetto associato al *core*, i *task* sono tre e creati attraverso altrettante funzioni:

- Gestione degli input ed ouput generici, in particolare riconoscimento degli eventi associati alla premuta di pulsanti e accensione del led di

fault quando necessario: questo *task* è definito dalla funzione *gpio_createTask.*

- Definizione dei segnali CAN di input e output, ricezione ed elaborazione del contenuto dei messaggi in ingresso e trasmissione di quelli in uscita; questo processo è descritto e configurato con *CAN_createTask.*
- Lettura della posizione dal sensore e degli stati di errore, di innesto/disinnesto e dei valori di corrente relativi ai due attuatori, oltre alla trasmissione ai *driver* dei comandi dati dall'utente: tale processo è implementato in *parklock_createTask*.

Una volta creati i *task*, la funzione *vTaskStartScheduler* avvia lo *scheduler* del sistema operativo, che è un programma del sistema operativo che si occupa della pianificazione dei *task* che devono essere eseguiti in parallelo; in particolare, sfruttando opportuni algoritmi di *scheduling*, stabilisce l'ordine temporale con cui vengono soddisfatte le richieste di accesso alle risorse, evitando 'conflitti' tra i processi [29] deleteri per il funzionamento del sistema.

La figura successiva riassume graficamente i processi gestiti dal *core*.

Fig. 2.19: processi realizzati dalla scheda core

Per quanto riguarda gli altri sorgenti di figura 2.18, i punti successivi ne riportano sinteticamente le finalità.

- *ad.c*: inizializza il convertitore analogico/digitale.
- *can.c*: include tutte le funzioni inerenti il *task* del CAN.

- *commonConfig.c*: definisce alcune variabili e funzioni utili per il funzionamento del sistema operativo.
- *consolle.c*: implementa la possibilità di interagire con la centralina per riga di comando.
- *DRV8701EVM.c*: caratterizza la comunicazione I2C verso i *driver*.
- *error.c*: definisce delle variabili inerenti la gestione dei *fault*.
- *flash.c*: presenta alcune funzioni relative alla memoria *flash*.
- *gpio.c*: configura la corrispondenza tra i pin e gli input/output generici e si occupa della loro lettura/scrittura.
- *parklock.c*: include le principali variabili e funzioni che il *core* impiega per comunicare con i *driver*.
- *temp.c*: riguarda il controllo della temperatura, in questo caso inutile in quanto al *core* non è collegata una termosonda.
- *usart.c*: configura la comunicazione seriale mediante porta USART.
- *watchdog.c*: implementa il sistema di supervisione *watchdog*.

La figura che segue rivela i file *header* contenuti nella sottocartella *inc* di *Application*.

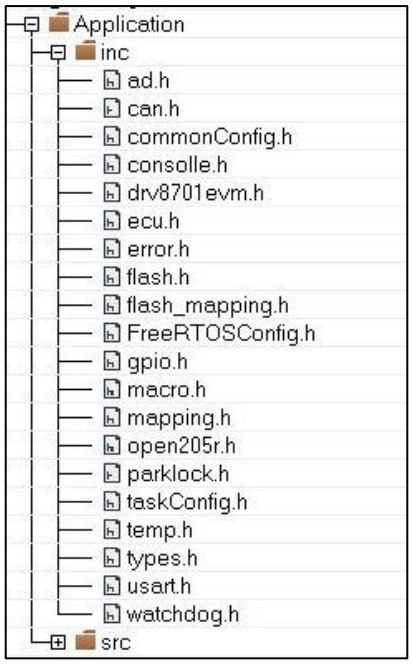


Fig. 2.20: headers dell'applicazione del progetto core

Si ricorda che un file *header* (o di intestazione) è un tipo di file che serve a facilitare l'impiego delle librerie da parte del programmatore e nel linguaggio C è caratterizzato dall'estensione '.h'. Un *header* è un file di testo che contiene principalmente i prototipi (i.e. le dichiarazioni) delle funzioni che sono definite nel file sorgente con lo stesso nome e con estensione '.c'; inoltre, un *header* può contenere dichiarazioni di variabili globali e di macro, definizioni di costanti e definizioni di tipi. Tutte le dichiarazioni e definizioni di un *header* possono essere scritti direttamente nel file oppure inseriti per inclusione di librerie standard supportate dal compilatore [30].

Riassumendo, i principali vantaggi legati all'impiego degli *header files* in un progetto sono due:

1) Se all'interno di un sorgente si desidera chiamare o al limite definire una funzione, allora non è necessario inserire prima esplicitamente la sua dichiarazione ma è sufficiente includere (mediante la direttiva di compilazione *#include*) l'*header* che ne contiene il prototipo all'interno del sorgente stesso.
2) La dichiarazione di una variabile globale all'interno di un *header* fa sì che il compilatore non restituisca errore quando, nella compilazione di un sorgente, trova un richiamo a tale variabile senza che questa sia definita nello stesso file, in quanto sa che la definizione è contenuta da un altro sorgente che la utilizza.

Non è interessante riassumere il contenuto dei singoli *header* in quanto principalmente dichiarano funzioni e variabili poi utilizzate nei sorgenti che li includono; ritengo invece importante fare qualche osservazione sul file *taskConfig.h*, in quanto definisce i parametri costanti su cui si basa la gestione dei processi.

```
// GPIO TASK
#define TASKCONFIG_GPIO_NAME                    "GPIO"
#define TASKCONFIG_GPIO_STACK                   256
#define TASKCONFIG_GPIO_PRIORITY                4
#define TASKCONFIG_GPIO_TIME_PERIOD             50  /* [ms] */
#define TASKCONFIG_GPIO_TICK_PERIOD             ((TickType_t)MACRO_MS_TICK(TASKCONFIG_GPIO_TIME_PERIOD))

// CAN TASK
#define TASKCONFIG_CAN_NAME                     "CAN"
#define TASKCONFIG_CAN_STACK                    1024
#define TASKCONFIG_CAN_PRIORITY                 6
#define TASKCONFIG_CAN_TIME_PERIOD              2  /* [ms] */
#define TASKCONFIG_CAN_TICK_PERIOD              ((TickType_t)MACRO_MS_TICK(TASKCONFIG_CAN_TIME_PERIOD))

// PARKLOCK TASK
#define TASKCONFIG_PARKLOCK_NAME                "PARKLOCK"
#define TASKCONFIG_PARKLOCK_STACK               1024
#define TASKCONFIG_PARKLOCK_PRIORITY            5
#define TASKCONFIG_PARKLOCK_TIME_PERIOD         20  /* [ms] */
#define TASKCONFIG_PARKLOCK_TICK_PERIOD         ((TickType_t)MACRO_MS_TICK(TASKCONFIG_PARKLOCK_TIME_PERIOD))
```

Fig. 2.21: corpo di taskConfig.h

In figura 2.21 si può osservare la suddivisione del file in tre sezioni distinte, una per ogni *task*: gestione dei GPIO, gestione della comunicazione CAN e gestione dell'interfacciamento con il *park-lock*. All'interno di ogni sezione si ritrovano le definizioni delle seguenti costanti (direttiva *#define*):

- *_NAME: nome associato al *task*.
- *_STACK: dimensione dello spazio in cui vengono salvate le variabili locali associate al *task*.
- *_PRIORITY: priorità del *task*, informazione fondamentale per consentire l'azione dello *scheduler* del sistema operativo.
- *_TIME_PERIOD: periodo in *ms* di esecuzione del *task*.
- *_TICK_PERIOD: numero di cicli di clock corrispondenti al periodo del *task*.

Le informazioni più importanti estrapolabili dall'immagine 2.23 sono due:

1) Il *task* a cui viene attribuita la massima priorità è quello relativo al GPIO, quindi segue la gestione dei *driver* e infine la comunicazione CAN. Si ricorda che la priorità detta l'ordine con cui le richieste di accesso alle risorse da parte dei vari processi vengono soddisfatte; dunque, ad esempio, se in corso di esecuzione sia il *task* GPIO che quello CAN desiderano usufruire della medesima risorsa (ad esempio un elemento in memoria), il *task* GPIO ha la precedenza.

2) Il processo che viene eseguito con frequenza più elevata è quello di ricezione e trasmissione dei messaggi CAN, con periodo di 2ms; segue lo scambio di comandi e dati con i *driver*, con periodo di 20ms, e l'aggiornamento dei GPIO, con periodo di 50ms.

Progetto sorgente delle schede driver

La figura seguente riporta l'interfaccia grafica di *Code Composer Studio* (CCS), l'IDE utilizzato per la gestione dei file del progetto sorgente dei *driver*.

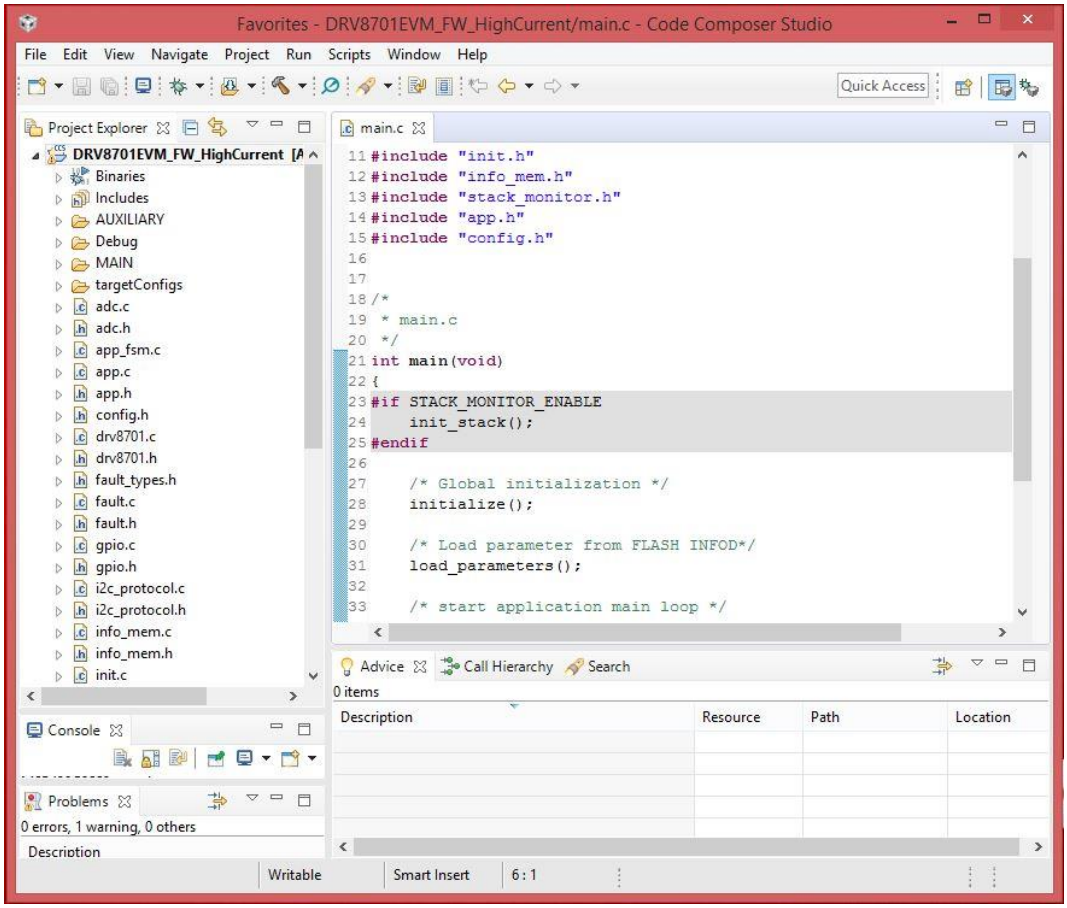


Fig. 2.22: interfaccia grafica di CCS

Per poter comprendere la strutturazione del progetto originale (così come di quelli modificati) è necessario sottolineare che esso è stato impostato per consentire la produzione di due eseguibili diversi, ciascuno caricato poi come *firmware* su una scheda *driver*. Infatti, la centralina è stata originariamente progettata per controllare una MGU dotata di un motorino *brushed* DC e di un solenoide ausiliario di

emergenza. I due attuatori elettrici differiscono nella struttura ma anche nelle modalità di azione; ad esempio, mentre il motore a spazzole può produrre coppia in entrambi i sensi di rotazione in base al verso della corrente che percorre gli avvolgimenti, il solenoide esercita una forza di orientamento indipendente dal senso della corrente. Quindi, il motorino deve essere controllato da un *driver* in grado di decidere il verso, oltre che il modulo, della tensione di comando, mentre il *driver* del solenoide può agire in modo unidirezionale.

Quindi, benché i sorgenti originali siano gli stessi per entrambe le schede, è possibile declinarli in fase di compilazione secondo due configurazioni differenti, indicate con i termini MAIN e AUX. A livello di sorgente, la distinzione tra le due configurazioni è realizzata mediante delle 'direttive di precompilazione'; trattasi di semplici linee di codice che vengono lette ed interpretate dal preprocessore (avviato con la compilazione) e consentono, ad esempio, di celare al compilatore dei segmenti di codice qualora sia o non sia definita una certa costante. La figura che segue (2.23) mostra il menu a tendina di compilazione (*Build*).

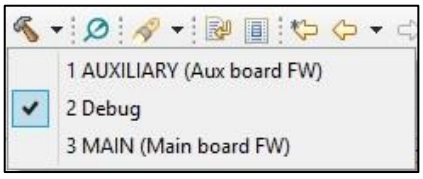


Fig. 2.23: menù di compilazione

Il menù precedente consente di scegliere tra tre modalità di compilazione del progetto, ciascuna delle quali conduce ad un diverso eseguibile finale:

1. *AUXILIARY (Aux board FW)*: porta all'eseguibile che, almeno originariamente, dovrebbe essere caricato sul *driver* AUX che comanda il solenoide di sicurezza;
2. Debug: modalità utilizzata per il controllo degli *stack* di memoria e che non è stata utilizzata per il raggiungimento degli scopi della tesi;
3. *MAIN (Main board FW)*: porta originariamente al *firmware* del *driver* MAIN che controlla il motore *brushed*.

In realtà, è possibile compilare il progetto come MAIN ma poi caricare l'eseguibile ottenuto sul *driver* AUX e viceversa: il nome della configurazione del *firmware* non è vincolato a quello della scheda su cui viene caricato. Dunque, la distinzione effettuata tra le due configurazioni di compilazione è utile solo perché consente di ottenere due *firmware* differenti partendo dallo stesso progetto sorgente. Ad esempio, i sorgenti originali prevedono che il *driver* MAIN sia caratterizzato con il *firmware* MAIN e controlli l'azionamento elettrico annesso secondo un *loop* chiuso di posizione con *loop* interno di corrente, mentre il *driver* AUX (dotato di *firmware* AUX) comandi il solenoide con una rampa di corrente; attraverso opportune modifiche ai sorgenti e, in particolare, alle direttive di precompilazione si possono determinare due *firmware* simili che permettano ai *driver* di comandare indipendentemente e in modo ciclico i motorini a spazzole di due MGU distinte.

La figura 2.24 mostra le sottocartelle della cartella principale del progetto (*DRV8701EVM_FW_HighCurrent*), presenti nella finestra *Project Explorer*.

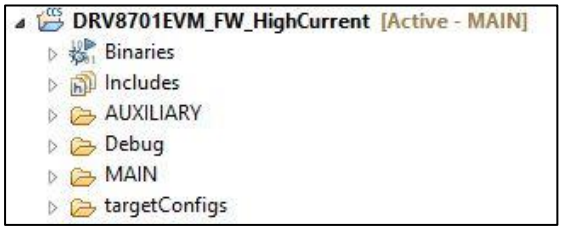


Fig. 2.24: sottocartelle del progetto dei driver

Le sottocartelle sono le seguenti:

- *Binaries*: contiene tutti gli eseguibili generati partendo dal progetto e declinandolo secondo le tre possibili configurazioni.
- *Includes*: racchiude tutte le librerie standard associate al progetto.
- *AUXILIARY*: include gli eseguibili associati alla configurazione AUX e i file necessari per il corretto caricamento del *firmware*.
- Debug: contiene gli eseguibili e i file complementari associati alla configurazione di debug.
- *MAIN*: racchiude gli eseguibili e i file complementari associati alla configurazione MAIN;

- *targetConfigs*: include un file con estensione *.ccxml* generato automaticamente dall'IDE in base alle proprietà del microprocessore della scheda e delle impostazioni della comunicazione con esso; si tratta di un file indispensabile per il debug e per il caricamento del *firmware*.

Il funzionamento del microprocessore del *driver* non viene coordinato da un sistema operativo, sono infatti sufficienti dei costrutti standard che fanno compiere al sistema delle operazioni a seconda dei comandi provenienti via I2C dal *core* e del valore di alcune variabili di stato.

Per quanto riguarda la descrizione del funzionamento dei *driver*, per motivi di sintesi e chiarezza si preferisce evitare di entrare nel dettaglio dei sorgenti e degli *header* del progetto, a favore dell'utilizzo di intuitivi diagrammi a blocchi per rappresentare le porzioni di codice di maggiore interesse. Inoltre, non ci si soffermerà in questo capitolo né sull'algoritmo originale di controllo e neppure sulle caratteristiche della rampa di corrente, in quanto tali argomenti verranno analizzati in modo approfondito nei capitoli dedicati al controllo di motori elettrici *brushed*.

L'immagine successiva mostra i cinque processi che vengono eseguiti sequenzialmente dal *driver* nel *loop* principale di funzionamento.

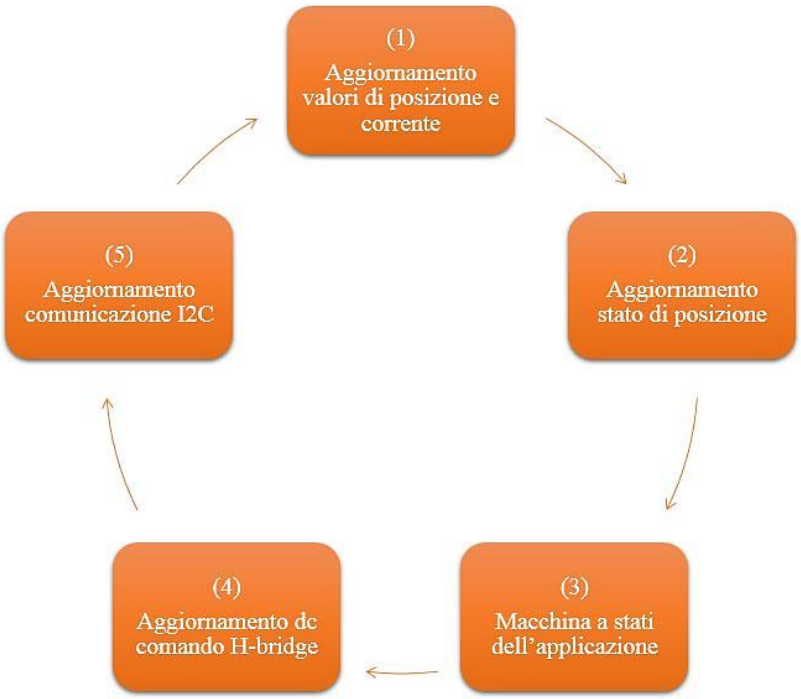


Fig. 2.25: fasi principali del ciclo di funzionamento dei driver

Dopo la fase di inizializzazione (del *clock*, del convertitore A/D, della comunicazione I2C, etc.) che avviene non appena la scheda è alimentata, i processi indicati in figura 2.25 sono eseguiti ciclicamente in senso orario da (1) a (5). Le righe seguenti riassumono quanto accade per ciascuna delle cinque fasi.

(1) Aggiornamento dei valori di posizione e corrente

Il processo di aggiornamento dei valori di *duty cycle* di posizione e di corrente erogata al carico è più complesso di quanto non potrebbe sembrare. Per cominciare è opportuno specificare come le due grandezze in questione vengano misurate. Il *driver* riceve dalla scheda filata una tensione nel campo 0-3.3V proporzionale al *dc* del segnale PWM (generato con frequenza 1000Hz) prodotto dal sensore Hall; la scheda filata opera il condizionamento del segnale mediante un filtro e un partitore di tensione. Il valore della corrente erogata al carico è misurato, invece, mediante un trasduttore che impiega un resistore *shunt* da 5mΩ quale elemento sensibile (converte la corrente in una piccola differenza di tensione) ed è completato da un

amplificatore con guadagno 20V/V e da un partitore di tensione (guadagno 0.5714), da cui deriva di nuovo una tensione analogica nel range 0-3.3V (fig. 2.26) [22-23].

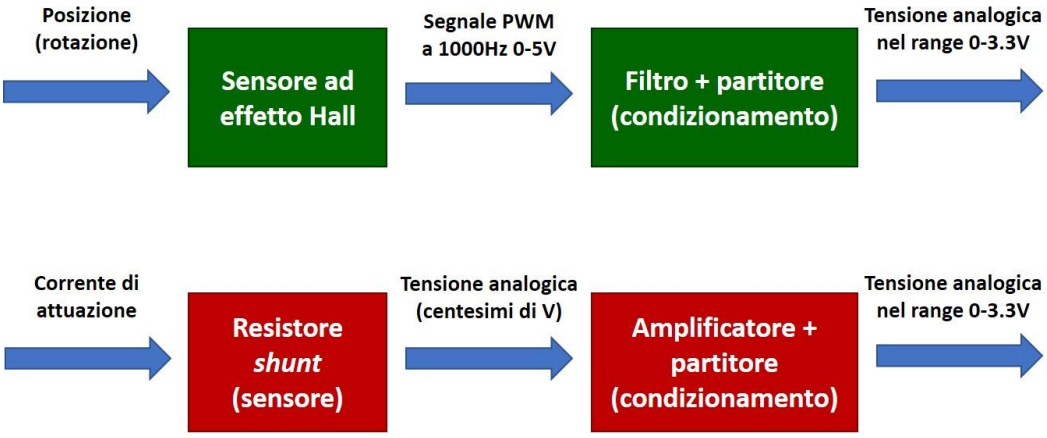


Fig. 2.26: trasduzione di posizione e corrente

Le tensioni prodotte dai trasduttori sono trasferite ad un convertitore A/D a 10 bit operante a 16MHz e i valori da questo generati vengono salvati in due *buffer*, da 8 bit per la posizione e da 16 bit per la corrente; il convertitore riceve il valore misurato 0-3.3V e restituisce un numero intero tra 0 e 2^{10}-1 ad esso proporzionale. La relazione tra il valore 'ingegneristico' della grandezza di interesse e quello convertito e memorizzato nel *buffer* dipende dalle caratteristiche della catena di misura e dalla risoluzione di conversione; la relazione tra il valore memorizzato e il corrispondente reale è stata determinata mediante le operazioni di calibrazione statica, di cui si tratterà successivamente.

Le variabili che vengono aggiornate al termine della fase in questione non, si riferiscono a valori istantanei ma a valori mediati: ad ogni ciclo di esecuzione del *firmware* del *driver* viene realizzata la media sui campioni contenuti nel *buffer* e quindi salvata in memoria. Siccome successivamente verranno richiamati nella trattazione, si sottolinea che il *dc* medio di posizione viene memorizzato nella variabile *POS_AVG_VAL* mentre quello di corrente in *SO_MCU_VAL*; le loro dichiarazioni e quelle dei due *buffer*, incluse in *adc.h*, si possono vedere nella figura successiva.

```
extern uint16_t POS_AVG_VAL;                /* dc medio di posizione */
extern volatile uint16_t pos_avg_buffer[];  /* buffer per dc di posizione */

extern uint16_t SO_MCU_AVG;                 /* corrente media al motore  */
extern volatile uint16_t so_avg_buffer[];   /* buffer per corrente al motore */
```

Fig. 2.27: alcune dichiarazioni in adc.h

(2) Aggiornamento dello stato di posizione

Lo 'stato di posizione' (da intendersi come stato derivante dal sensore di posizione) è rappresentato dalla variabile *pos_status* e dipende, come si può intuire, dal valore del *dc* di posizione; in particolare, è determinato dalla relazione tra il *dc* e le soglie di innesto e disinnesto del *park-lock*, definite a livello di sorgente. La figura successiva mostra la dichiarazione di *pos_status*, la definizione del suo tipo *pos_status_t*, la dichiarazione delle soglie e dell'isteresi e quella della funzione per l'aggiornamento di *pos_status* (tutte inserite nell'*header position.h).*

```
/* possibili stati di posizione  */
typedef enum
{
    POS_ENGAGED = 0,
    POS_UNKNOWN,
    POS_DISENGAGED,
    POS_MAX
} pos_status_t;


/* Soglie e isteresi */
extern uint16_t POS_ENGAGED_THR;    // soglia di engage [0 - 1023]
extern uint16_t POS_DISENGAGED_THR; // soglia di disengage [0 - 1023]
extern uint16_t POS_THR_HYSTERESIS; // isteresi sensore [0 - 1023]

extern pos_status_t pos_status; /* stato di posizione */

void update_pos_status(); /* funzione per update stato di posizione */
```

Fig. 2.28: definizioni e dichiarazioni in position.h

I valori delle soglie di innesto (*POS_ENGAGED_THR*), di disinnesto (*POS_DISENGAGED_THR*) e dell'isteresi (*POS_THR_HYSTERESIS*) sono definiti all'interno di *position.c* e posti uguali ai valori di default, a loro volta specificati in *config.h*. Le due soglie, inoltre, possono essere definite mediante una procedura di calibrazione manuale avviabile premendo contemporaneamente per 3s i pulsanti DISENG MAIN e RELOAD.

L'attribuzione del valore del *pos_status* avviene secondo la logica seguente:

- Se il valore della posizione supera quello della soglia di innesto maggiorato dell'isteresi, allora *pos_status = POS_ENGAGED*: il *park-lock* è innestato.
- Se il valore della posizione è inferiore a quello della soglia di disinnesto meno l'isteresi, allora *pos_status = POS_DISENGAGED*: il *park-lock* è disinnestato.
- Se non è vera nessuna delle due condizioni precedenti, ossia il valore di posizione è compreso tra le due soglie, allora *pos_status = POS_UNKNOWN*: il *park-lock* è in uno stato 'sconosciuto', nel senso di intermedio e privo di particolare interesse.

Il termine di isteresi tiene conto della non idealità del sensore Hall: ad uno stesso angolo di rotazione possono corrispondere due valori differenti di *dc* a seconda che l'angolo sia stato raggiunto ruotando l'albero in senso orario o in senso antiorario; dunque è corretto considerare un certo scostamento, dipendente dal sensore, tra le soglie definite staticamente e il valore che il sensore in moto restituisce al raggiungimento di queste posizioni limite. Si osserva che, ovviamente, la correzione delle due soglie è di segno opposto poiché vengono raggiunte con movimenti in verso opposto.

(3) Macchina a stati dell'applicazione

Con il termine 'applicazione', in informatica, vengono identificati uno o più programmi eseguiti su un dispositivo informatico con lo scopo di realizzare funzionalità e/o servizi utili per l'utente e da questo selezionabili mediante un'opportuna interfaccia [30]. Quindi la differenza tra un'applicazione (o programma applicativo) e, ad esempio, il sistema operativo con i suoi componenti è che, mentre il primo viene implementato ed eseguito per assolvere ad una specifica esigenza dell'utente, i secondi sono fondamentali per il funzionamento generale del dispositivo.

Nel caso del progetto del *driver*, per 'applicazione' si intende l'insieme di attività che permettono alla scheda *driver* di controllare intelligentemente il moto di un motore elettrico o l'attivazione di un solenoide. Il cuore dell'applicazione è

rappresentato da un insieme di funzioni (distribuite tra *app.c* e *app_fsm.c*), la cui logica di esecuzione è definita mediante una 'macchina a stati': con questa espressione si desidera indicare un sistema il cui comportamento è definito, ad ogni ciclo, dal valore di una determinata variabile di stato [30]. La macchina a stati è implementata mediante una funzione definita in *app_fsm.c*, la quale sfrutta un classico costrutto *switch-case* per eseguire un certo pacchetto di funzioni in base al valore assunto dalla variabile di nome *app_status*.

L'immagine seguente mostra la definizione del tipo di *app_status* (di nome *status_t*) e la dichiarazione di *app_status*, entrambe contenute nell'*header app_fsm.h*.

```
/* tipo dello stato dell'applicazione */
typedef enum
{
    STATUS_IDLE_UNKNOWN = 0,    // stato inattivo
    STATUS_ENGAGE,              // innesto in corso
    STATUS_ENGAGED,             // innesto completato
    STATUS_DISENGAGE,           // disinnesto in corso
    STATUS_DISENGAGED,          // disinnesto completato
    STATUS_RELOAD,              // reload in corso
    STATUS_CALIBRATION,         // calibrazione in corso
    STATUS_INIT,                // inizializzazione
    STATUS_PARAM_UPDATE         // aggiornamento dei parametri
}status_t;

extern status_t app_status; /* stato dell'applicazione */
```

Fig. 2.29: definizione di status_t e dichiarazione di app_status

La definizione di *status_t* in fig. 2.29 è interessante in quanto permette di osservare tutti i valori che *app_status* può assumere, ciascuno dei quali è stato commentato a lato. La figura sottostante riporta invece un diagramma a blocchi che schematizza il funzionamento della macchina a stati di *app_fsm.c* e quindi la correlazione tra il valore assunto da *app_status* e le istruzioni eseguite del *driver*.

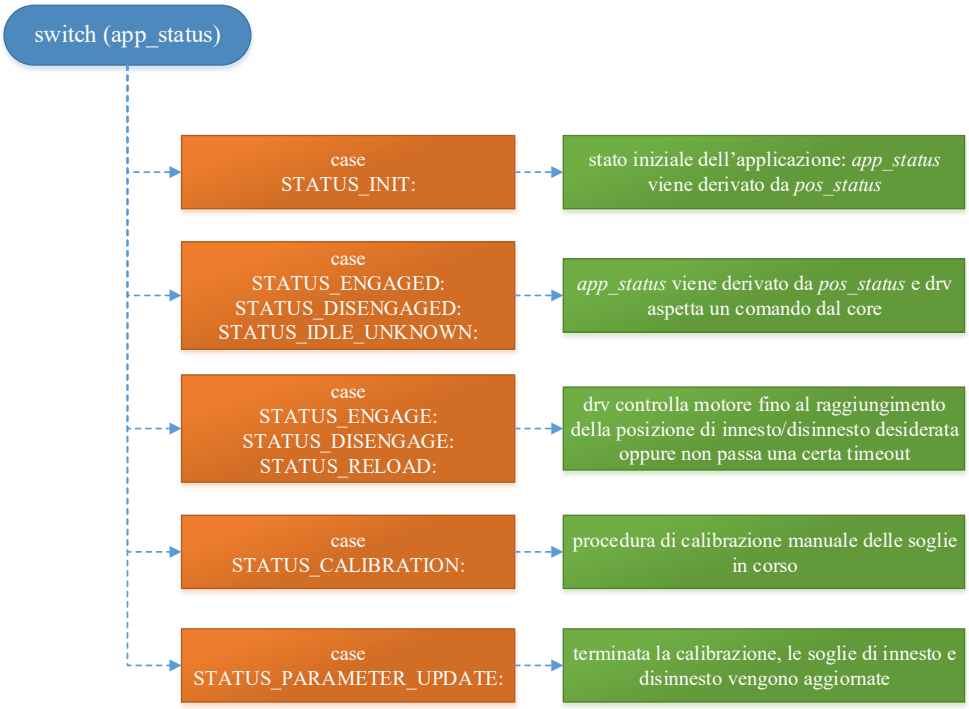


Fig. 2.30: schema della macchina a stati dell'applicazione

In riferimento alla figura 2.30, si osserva che quando l'applicazione riconosce lo stato di *park-lock* innestato (*STATUS_ENGAGED*), quello di *park-lock* disinnestato (*STATUS_DISENGAGED*) oppure il dispositivo si trova in una posizione intermedia tra le due soglie mentre non è in corso un'attuazione controllata del motore o solenoide (*STATUS_IDLE_UNKNOWN*), allora il *driver* continua ad aggiornare lo stato dell'applicazione in base a quello di posizione e aspetta la trasmissione di un'istruzione da parte del *core*. La ricezione di un comando porta ad una nuova fase di funzionamento e comporta l'aggiornamento di molti parametri inerenti il controllo dell'attuazione; ad esempio, in fig. 2.31, si leggono le operazioni che, secondo il progetto originale, il *driver* MAIN deve eseguire quando riceve il comando di *engage*.

```
case CMD_ENGAGE:
    drv8701_phase = ENGAGE_PHASE;
    reset_control();
    position_set_point = POS_ENGAGED_THR + (POS_THR_HYSTERESIS<<1);
    set_timeout(fsm_timeout,ENGAGE_TIMEOUT);
    check_pos_status = POS_ENGAGED;
    check_in_progress = false;
    app_status = STATUS_ENGAGE;
    break;
```

Fig. 2.31: interpretazione del comando CMD_ENGAGE

In seguito alla ricezione di CMD_*ENGAGE* il *driver* deve eseguire le seguenti istruzioni:

- La fase di *engage* è riconosciuta come attuale fase di funzionamento.
- I parametri di controllo (*dc* di comando del motore, variabili degli anelli di controllo, etc.) sono resettati.
- Il *set point* dell'anello di posizione viene impostato ad un valore adeguato, coerente con il comando ricevuto.
- La *timeout* per la permanenza della macchina a stati nella fase di controllo motore viene impostata ad un valore predefinito.
- Lo stato di posizione in innesto (*POS_ENGAGED*) viene riconosciuto come termine di confronto per l'interruzione della fase di *engage*.
- Lo stato dell'applicazione (*app_status*) diventa *STATUS_ENGAGE*: questo porterà il *driver* a modificare il proprio comportamento al ciclo successivo e quindi ad iniziare il controllo del motore.

(4) Aggiornamento dc di comando H-Bridge

Ciascuna delle due schede *driver* della centralina può comandare un attuatore elettrico DC e controllare in modo continuo la sua tensione di comando (e quindi anche la potenza erogata) grazie all'impiego di un classico ponte H (o *H-Bridge*) e di un adeguato *driver* per il controllo dei MOSFET (il *DRV8701*). La struttura del ponte H della scheda *DRV8701EVM* è schematizzata nella figura che segue [23].

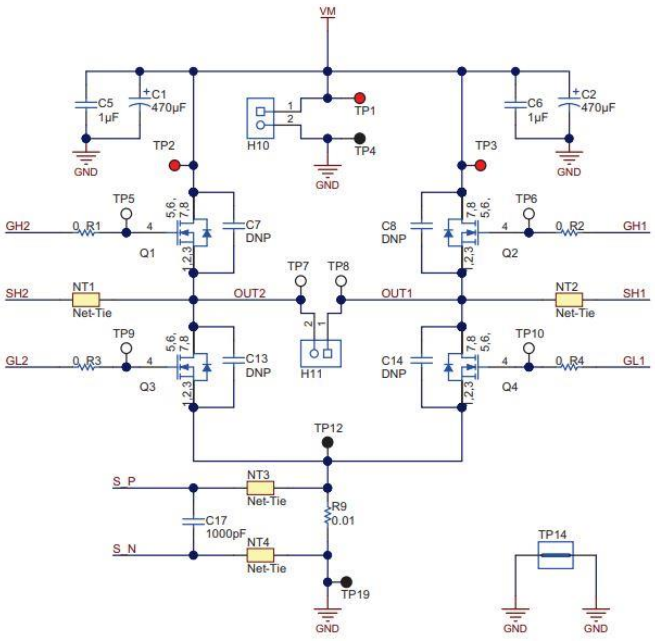


Fig. 2.32: H-Bridge del driver

Il ponte H è un dispositivo elettronico che consente di imporre la polarità della tensione applicata ad un carico. In riferimento alla fig. 2.32, il ramo superiore e quello inferiore sono collegati, rispettivamente, alla tensione costante di 12V fornita dall'alimentatore (*VM*) e alla massa (*GND*). Il carico, ossia l'attuatore, viene alimentato in corrispondenza di *H11* (posizione centrale), mentre sui quattro rami laterali sono disposti altrettanti MOSFET (*Q1, Q2, Q3, Q4*), ciascuno accompagnato da un diodo disposto in parallelo che impedisce alla corrente di scorrere nel verso opposto a quello previsto. I quattro dispositivi di commutazione possono essere comandati in modo indipendente e consentono di applicare la tensione di comando agli avvolgimenti in un verso o nell'altro, per la movimentazione (tensione di comando e corrente concordi) ed, eventualmente, la frenata (tensione di comando e corrente discordi) del carico.

I MOSFET vengono comandati in PWM (*Pulse Width Modulation*) mediante l'azione di *DRV8701*. Il microcontrollore *MSP430G2553* genera un segnale di riferimento a bassa tensione (0-3.3V) che raggiunge il *driver* per *H-Bridge*; questo produce un segnale PWM ad alta corrente che comanda uno o più MOSFET, la cui selezione dipende dal verso in cui si desidera far ruotare il motore. Le proprietà induttive e capacitive degli avvolgimenti, unite all'elevata dinamica del treno di

impulsi, fanno sì che il carico veda l'applicazione di una tensione analogica nel range 0-12V e proporzionale sia al segnale di riferimento proveniente dalla MGU che al *duty cycle* del segnale PWM di comando. La tecnica PWM è utilizzata molto frequentemente per il pilotaggio di attuatori (ma anche di lampade) operanti in corrente continua; infatti, questo tipo di modulazione consente di trasportare un'informazione analogica tramite un segnale digitale, portando sia risparmio in termini di componenti (ad esempio, non è necessario un convertitore D/A) che riduzione delle dissipazioni energetiche.

Tornando *loop* di funzionamento della scheda, ad ogni ciclo la macchina a stati produce un certo valore di *dc* di comando (associato alla variabile intera *DRV8701_pwm*); si osserva che il *duty cycle* è, in generale, diverso da zero solo nel caso in cui lo stato dell'applicazione preveda l'attuazione del motore/solenoide (ossia se *app_status* è uguale a *STATUS_ENGAGE*, *STATUS_*DIS*ENGAGE* e *STATUS_RELOAD*). La macchina a stati definisce anche la polarità della tensione di comando attraverso la variabile *DRV8701_phase*.

(5) <u>Aggiornamento della comunicazione I2C</u>

Nell'ultima fase del ciclo di funzionamento, il *driver* aggiorna il contenuto di due registri che il *core* legge periodicamente tramite bus I2C e le cui informazioni vengono inserite all'interno del messaggio CAN di uscita:

1. Lo stato dell'applicazione *app_status*.
2. La variabile intera *fault_mask*, il cui valore rappresenta l'eventuale stato di errore del sistema.

Per quanto riguarda la condizione di *fault*, il *driver* è in grado di rilevare cinque errori, corrispondenti ad altrettanti codici esadecimali:

- 0x01: sensore dell'attuatore MAIN fuori range.
- 0x02: sensore dell'attuatore AUX fuori range.
- 0x10: timeout per l'attuatore MAIN.
- 0x20: attuatore MAIN non connesso elettricamente.
- 0x40: attuatore AUX non connesso elettricamente.

CAPITOLO 3

L'interfacciamento centralina – utente con CANape

Nel presente capitolo verranno descritte le principali caratteristiche sia a livello logico che di hardware dello standard di comunicazione CAN, molto utilizzato nell'campo dell'automotive; quindi, si procederà a delineare le funzionalità del software *CANape* e le modalità secondo cui esso è stato utilizzato per controllare da PC il funzionamento della centralina e delle sue schede.

3.1 La comunicazione CAN

Le origini del protocollo CAN

La comunicazione di tipo CAN (*Controller Area Network*) è stata originariamente sviluppata da *Bosch* nel 1985 per le reti a bordo veicolo. In passato, i dispositivi elettronici operanti on board venivano connessi medianti sistemi di cablaggio 'punto a punto', ossia la trasmissione di informazioni tra due componenti necessitava di un collegamento elettrico dedicato. Il sempre maggiore utilizzo di elementi elettronici sui veicoli aveva prodotto come risultato la presenza di un'ingombrante quantità di fili elettrici, tanto pesante quanto costosa. Pertanto, il cablaggio dedicato è stato via via rimpiazzato da reti di comunicazione a bordo veicolo operanti secondo lo standard CAN: trattasi di un sistema a bus seriale ad alta integrità che permette a dispositivi intelligenti di passarsi informazioni mediante una rete durevole e poco costosa. Un grande vantaggio apportato dall'utilizzo di questo standard è che le unità di controllo elettronico (le ECU) collegate al bus possono essere dotate di una singola interfaccia CAN invece che di numerose porte e pin analogici e digitali (fig. 3.1) [32].

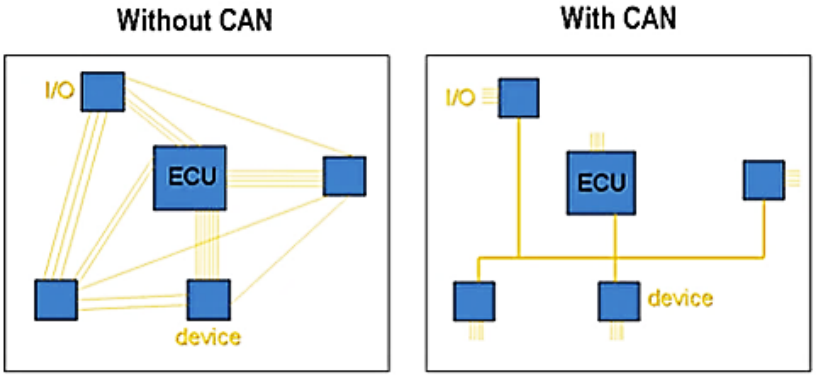


Fig. 3.1: confronto tra comunicazione tradizionale e comunicazione CAN

Ogni dispositivo della rete CAN deve possedere un chip dedicato alla comunicazione CAN e quindi deve essere 'intelligente'. Tutti gli apparecchi connessi al bus vedono tutti i messaggi trasmessi: è compito del singolo dispositivo il riconoscimento dell'eventuale rilevanza per sé di un messaggio e quindi decidere se leggerlo interamente ed elaborarne il contenuto oppure filtrarlo. Inoltre, ogni messaggio CAN è caratterizzato da una priorità: se due nodi desiderano trasmettere due segnali sullo stesso bus contemporaneamente, quello con priorità più elevata viene trasmesso mentre l'altro viene posticipato.

La logica di trasmissione dei messaggi e la risoluzione dei conflitti

La trasmissione dei messaggi (chiamati *Data Frame*) secondo lo standard CAN e la relativa interpretazione si basa sulla distinzione tra bit 'dominanti' e bit 'recessivi': i primi corrispondono agli 0 logici mentre i secondi agli 1 logici. Nel caso in cui un nodo della rete trasmetta un bit dominante e un secondo nodo un bit recessivo, il primo predomina sul secondo ed è l'unico bit a venire trasmesso sul bus: in questo modo si realizza un AND logico e la figura seguente ne mostra la tabella di verità (fig. 3.2) [32].

Stato del bus quando due nodi trasmettono

	dominante	**recessivo**
dominante	***dominante***	*dominante*
recessivo	**dominante**	***recessivo***

AND logico

	0	1
0	0	0
1	0	1

Fig. 3.2: tabella di verità dei bit dominanti e recessivi

È fondamentale comprendere il fatto che la rete CAN è di tipo *peer-to-peer*; ciò significa che non è presente un nodo (*master*) che controlli quando gli altri nodi (*slave*) possano accedere alla lettura e/o alla scrittura dei dati sul bus: quando un nodo è pronto a trasmettere un dato controlla se il bus è occupato e, se non lo è, immette il suo *Data Frame* sul canale di comunicazione. Si evidenzia che i messaggi trasmessi non presentano indirizzi né per l'identificazione del trasmettitore né per quella del ricevitore (o dei ricevitori); tuttavia, ogni messaggio CAN riporta una sequenza identificativa di bit (*Arbitration ID*) che ne identifica in modo unico il contenuto e la priorità: a seconda dell'*Arbitration ID* ogni nodo della rete decide se accettare o meno il messaggio che occupa il bus.

Per quanto concerne la risoluzione del conflitti, se due o più nodi iniziano la trasmissione nel medesimo istante, la collisione dei messaggi viene evitata mediante l'implementazione del metodo di accesso alla rete di tipo CMSA/CA + AMP. Ognuno dei nodi in trasmissione invia i bit del proprio identificatore e monitora il livello del bus: finché i bit provenienti dai trasmettitori sono identici non accade nulla; quando, però, un nodo trasmette un bit recessivo (valore 1) ma ne legge uno dominante (valore 0) allora interrompe la trasmissione dei dati e passa in modalità di ascolto: in questo modo l'informazione con priorità più alta vince il conflitto e viene comunicata via bus per prima. Naturalmente gli altri *Data Frame* che hanno preso parte al conflitto non vengono scartati: i nodi 'sconfitti' trasmetteranno il proprio messaggio in seguito, sempre seguendo la stessa modalità di accesso al bus dati (fig. 3.3) [32].

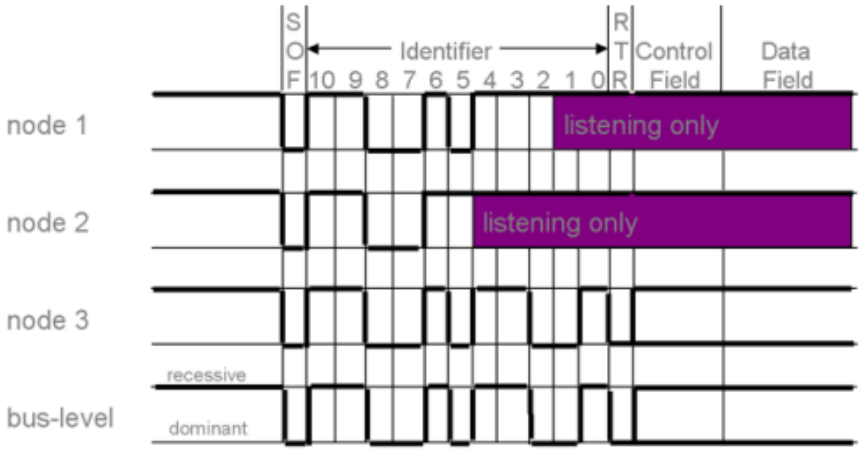


Fig. 3.3: esempio di conflitto di trasmissione su rete CAN

Le caratteristiche di trasmissione hardware

A livello hardware, tutte le ECU della rete CAN sono connesse tra di loro mediante un bus a due fili a doppino intrecciato, con impedenza nominale caratteristica di 120Ω. Per poter utilizzare una rete CAN, ogni nodo deve essere dotato della seguente elettronica (fig. 3.4) [32].

- CPU (microcontrollore dedicato o processore *host*).
  Si tratta dell'elemento che definisce il contenuto dei messaggi e quali di questi trasmettere: sensori, attuatori e dispositivi di controllo devono essere dotati di un processore o devono poter usufruire di un processore *host*.
- Controllore CAN.
  Si tratta di un componente spesso integrato nel microcontrollore. Per quanto riguarda la ricezione, questo *chip* immagazzina i bit ricevuti per via seriale dal bus fino a quando il messaggio non è stato completamente ricevuto, il quale verrà quindi passato al processore. Per la trasmissione, invece, il processore invia il messaggio al controllore CAN, che a sua volta lo trasmette serialmente sul bus quando questo è libero.
- Transceiver. Quando il nodo è in ricezione, converte il flusso di dati dai livelli logici di tensione a cui lavora il bus CAN (tipicamente 0-

5V) a quelli a cui opera in controllore CAN (di solito 0-3.3V); il contrario avviene, naturalmente, per la trasmissione.

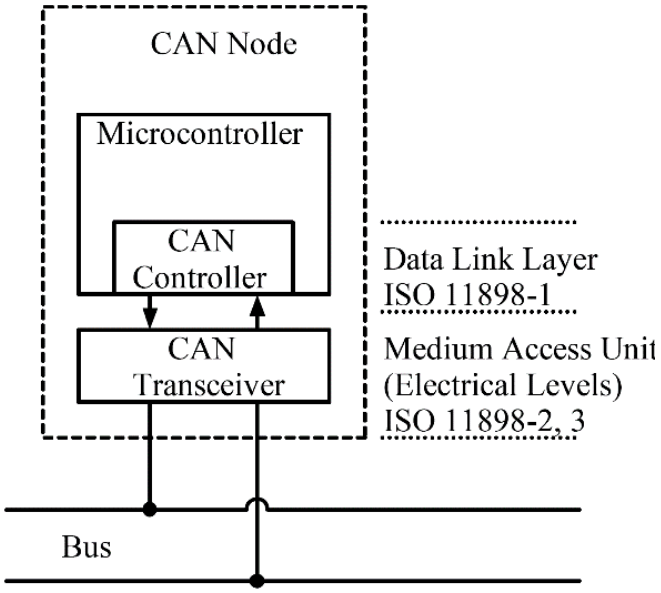


Fig. 3.4: elementi hardware di interfacciamento nodo – bus CAN

La struttura del messaggio CAN (Data Frame)

L'immagine seguente rappresenta la struttura di un *Data Frame* trasmesso su bus CAN; si sottolinea che un nodo trasmette un *Data Frame* quando desidera farlo o quando gli viene richiesto di farlo da un altro nodo [32].

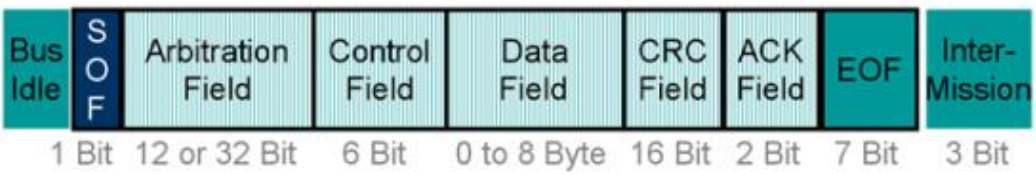


Fig. 3.5: struttura del Data Frame CAN

La funzionalità delle varie sequenze di bit che compongono un messaggio CAN sono brevemente spiegate nelle righe seguenti.

1. ***Start Of Frame (SOF).***

Si tratta di un bit dominante che realizza la sincronizzazione tra gli oscillatori del nodo trasmettitore e di tutti quelli ricevitori; in realtà, è la transizione dello stato

del bus da recessivo a dominante ad avviare la sincronizzazione, la quale viene abilitata dal SOF non solo in stato di bus inattivo ma anche in quello di sospensione e per l'ultimo bit di spazio *inter-frame*.

2. *Arbitration Field*

È il campo che definisce il contenuto e la priorità del messaggio; tale caratterizzazione è realizzata mediante i bit di identificazione, che possono essere 11 o 29 a seconda del formato del *Data Frame*. La figura 3.6 mostra la struttura dell'*Arbitration Field* nei suoi due possibili formati [32].

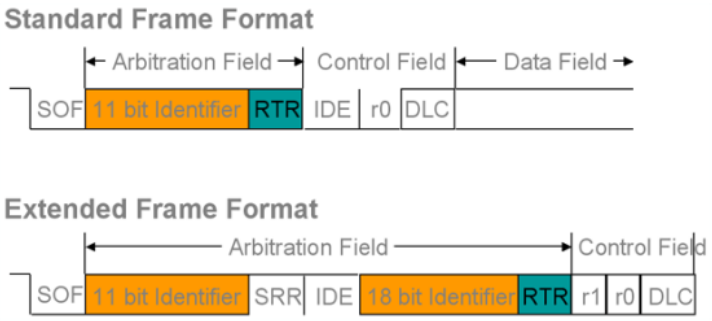


Fig. 3.6: formati dell'Arbitration Field

3. *Control Field.*

Questo campo specifica il numero di byte che compongono il messaggio utile (contenuto nel successivo *Data Field*) trasportato dal *Data Frame*. Il numero di byte utili è indicato dal valore del *Data Length Code* (DLC), codice di 4 bit incluso proprio nel *Control Field*. Il massimo numero di byte di dati utili trasmissibili in un *Data Frame* è otto; il DLC assume i valori da 0 a 7 per indicate dimensioni del *Data Field* da 0 a 7 byte, mentre tutti gli altri numeri rappresentabili con i 4 bit dei CRC (da 9 a 15) indicano che il *Data Field* è lungo 8 byte e possono essere anche impiegati per scopi legati alla specifica applicazione CAN.

4. *Data Field.*

Come già anticipato, è la sequenza di bit che trasporta la parte utile del messaggio CAN; può essere lungo un massimo di 8 byte e la sua lunghezza è

indicata dal DLC. Nella maggior parte dei casi, il *Data Field* trasmette più dati ed informazioni, i quali prendono il nome di 'segnali'.

5. ***CRC Field.***

Il *Cyclic Redundancy Check* (CRC) *Field* viene utilizzato per individuare possibili errori di trasmissione; è costituito da 15 bit che compongono una sequenza di controllo, derivata da un algoritmo di ridondanza ciclica, e da un bit di delimitazione.

6. ***Acknowledge (ACK) Field.***

Si tratta di un campo lungo 2 bit che contiene l'*ACK Slot* e l'*ACK Delimiter*. Il trasmettitore di *Data Frame* invia bit recessivi per entrambi i bit del campo; un ricevitore informa il trasmettitore di aver ricevuto il messaggio correttamente mandando un bit dominante durante l'*ACK Slot*. Se il nodo trasmettitore individua un riconoscimento positivo (ossia un *ACK Slot*) dominante, allora comprende che almeno uno dei nodi collegati al bus ha ricevuto correttamente il messaggio.

7. ***End Of Frame.***

Si tratta di una sequenza di 7 bit recessivi che comunicano ai nodi della rete che il *Data Frame* è terminato.

3.2 I messaggi CAN scambiati tra centralina e PC

Come già evidenziato, la trasmissione e ricezione di messaggi via CAN è uno dei tre *task* eseguiti dal *core*. La comunicazione CAN consente alla centralina di scambiare con un dispositivo intelligente (ad esempio un computer) una serie predefinita di informazioni, dati e comandi in modo bidirezionale. I normali elaboratori elettronici non sono in grado di interpretare messaggi costruiti secondo quel protocollo e, di conseguenza, mancano anche della porta CAN; per questo motivo, la comunicazione tra PC e centralina ha richiesto un'interfaccia hardware capace di convertire il *Data Frame* dallo standard di trasmissione CAN a quello USB: tale elemento intermedio è il *CANcase* (in particolare un *CANcaseXL Log*), sviluppato e venduto dalla *Vector Informatik*, il quale viene mostrato nella figura

successiva mentre è collegato ai cavi di comunicazione CAN e USB. Il *CANcase* utilizzato ha due porte CAN, corrispondenti ad altrettanti 'canali', il che vuol dire che questo dispositivo può essere collegato a due bus CAN in contemporanea (e costituire un nodo per ciascuna delle due) e trasferire distintamente i relativi messaggi ad un elaboratore.

Fig. 3.7: CANcase con cavi di collegamento

Lo schema di figura 3.8 esplicita i segnali contenuti nei *Data Frame* scambiati tra PC e centralina.

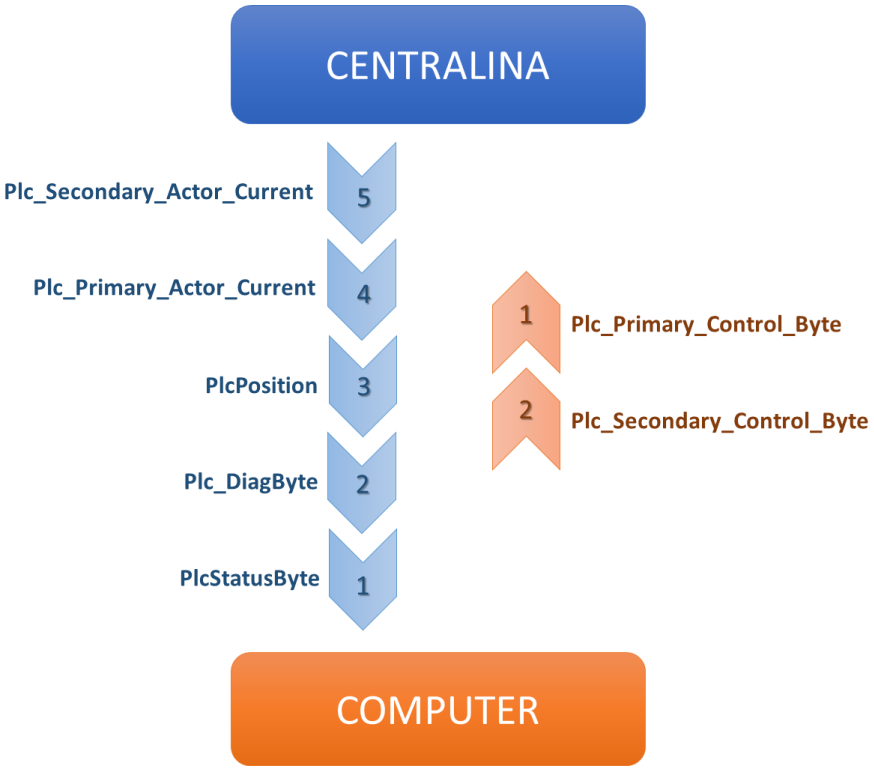


Fig. 3.8: segnali scambiati tra centralina e PC mediante CAN

Il messaggio CAN inviato dalla centralina è costituito dai seguenti 5 segnali:

1) ***PlcStatusByte.***

Riflette lo stato interno di entrambe le schede *driver*. I suoi possibili valori esadecimali, accompagnati dal nome con cui ogni dato viene interpretato da computer, sono:

- *0x0 (PLC_STATUS_ENGAGED)*: se il *park-lock* è innestato.
- *0x1 (PLC_STATUS_DISENGAGED)*: se il *park-lock* è disinnestato.
- *0x2 (PLC_STATUS_INIT)*: se c'è discrepanza tra il valore dello stato dell'applicazione per le due schede.

2) ***Plc_DiagByte.***

Informa dell'eventuale stato di errore (*fault*) di uno dei due *driver* o di entrambi. I valori esadecimali che può assumere sono i seguenti:

- *0x0*: nessun errore.
- *0x1*: valore del sensore del *driver* MAIN fuori scala.
- *0x2*: valore del sensore del *driver* AUX fuori scala.
- *0x10*: *timeout* del MAIN, ossia l'attuazione controllata eseguita dal MAIN in fase di *engage* o *disengage* è durata più del previsto.
- *0x20*: mancanza di connessione elettrica per il motore MAIN.
- *0x40*: mancanza di connessione elettrica per il motore AUX.

3) ***PlcPosition***: trasporta l'informazione del *duty cycle* di posizione.
4) ***Plc_Primary_Actor_Current***: riporta la corrente assorbita dal carico del *driver* MAIN.
5) ***Plc_Secondary_Actor_Current***: riporta la corrente assorbita dal carico del *driver* AUX.

La figura 3.9 mostra la definizione (mediante la direttiva di precompilazione *#define*) delle costanti che identificano la struttura del *Data Frame* di output della

ECU; le linee di codice riportate sono incluse nel file *can.c* del progetto sorgente del *core*.

```
/* PARKLOCK TX Offset dei segnali partendo dal bit 0 del byte 0 (in bit è incluso) */
#define PlcStatusByte                    0
#define Plc_DiagByte                     8
#define PlcPosition                      24
#define Plc_Primary_Actor_Current        42
#define Plc_Secondary_Actor_Current      62

/* PARKLOCK TX Dimensione in bit dei segnali */
#define PlcStatusByte_SIZE               8
#define Plc_DiagByte_SIZE                8
#define PlcPosition_SIZE                 16
#define Plc_Primary_Actor_Current_SIZE   14
#define Plc_Secondary_Actor_Current_SIZE 12
```

Fig. 3.9: parametri identificativi della struttura del Data Frame di output

Le prime cinque definizioni identificano i bit di partenza dei segnali in uscita, mentre le seconde cinque specificano la dimensione in bit di ciascuno di essi. Ad esempio, la sequenza di bit che rappresenta la corrente assorbita dall'attuatore primario parte dal bit 42 ed è lunga 14 bit, pertanto termina con il bit 55.

Il messaggio CAN ricevuto dalla centralina è composto, invece, dai seguenti segnali:

1. ***Plc_Primary_ControlByte***: byte per il controllo dell'attuatore primario, comandato dalla scheda MAIN.
2. ***Plc_Secondary_ControlByte***: byte per il controllo dell'attuatore secondario, comandato dalla scheda AUX.

Entrambi i segnali possono assumere i seguenti valori:

- *PLC_REQ_INIT*: è il valore trasmesso di default, al *driver* non viene richiesta nessuna azione.
- *PLC_REQ_DISENGAGE*: comanda al *driver* l'avvio della fase di disinnesto (se tale fase di funzionamento è prevista).
- *PLC_REQ_ENGAGE*: comanda al *driver* l'avvio della fase di innesto.

Come si può osservare, non è prevista la possibilità di ordinare ad un *driver* la fase di *reload* mediante la comunicazione CAN.

La figura 3.10 presenta le definizioni di *can.c* che caratterizzano la struttura del *Data Frame* di input.

```
/* PARKLOCK TX Offset dei segnali partendo dal bit 0 del byte 0 (in bit è incluso) */
#define Plc_Primary_ControlByte          0
#define Plc_Secondary_ControlByte        24

/* PARKLOCK TX Dimensione in bit dei segnali */
#define Plc_Primary_ControlByte_SIZE     8
#define Plc_Secondary_ControlByte_SIZE   8
```

Fig. 3.10: parametri identificativi della struttura del Data Frame di input

Come nel caso precedente, le prime due definizioni identificano l'offset in bit dei segnali rispetto al bit 0 del byte 0, mentre le ultime due ne indicano la dimensione in bit.

3.3 L'implementazione dell'interfaccia con CANape

CANape è un software sviluppato da *Vector Informatik* appositamente per la calibrazione in tempo reale degli algoritmi di controllo attuati delle ECU connesse ad una rete CAN. La calibrazione delle centraline permette di adattarne il comportamento a seconda dell'applicazione; ad esempio, in ambito automotive (dove questo strumento software è ampiamente utilizzato) l'utilizzo di *CANape* consente di caratterizzare il funzionamento di una o più ECU in modo diverso a seconda del modello o della variante del veicolo considerato. Il grande vantaggio legato all'impiego di uno strumento software quale *CANape* è che permette di modificare i parametri di funzionamento di un nodo senza dover aprire un IDE, variare il codice sorgente e, infine, caricarlo sull'elettronica associata: è sufficiente, invece, agire tramite le apposite finestre di calibrazione inserite nell'interfaccia grafica del software.

Le funzioni richieste per modificare i parametri di lavoro di un nodo del bus CAN sono *features* standard di *CANape*; inoltre, è possibile estendere le capacità del software in vari modi, ad esempio abilitando l'accesso in tempo reale a modelli *Simulink*, i quali possono ricevere in input grandezze trasmesse dai nodi e produrre in uscita andamenti simulati che possono essere visualizzati a fianco di quelli reali. *CANape* è dotato anche di un proprio linguaggio di programmazione, il CASL

(*CAlculation and Scripting Language*), che permette la scrittura di funzioni e script integrabili nell'esecuzione del software [33]. Si evidenzia che l'adempimento delle attività associate alla tesi non ha richiesto l'utilizzo di tutte le funzionalità di *CANape* ed in particolare di quelle più complesse e avanzate; in particolare, le schede della centralina non posseggono le caratteristiche necessarie per consentire le attività di taratura dei parametri di controllo attraverso finestre di calibrazione.

In definitiva, *CANape* è stato impiegato essenzialmente per le seguenti finalità:

- Per visualizzare l'andamento temporale delle informazioni (valore della posizione, delle correnti, stato dell'applicazione e di *fault*) trasmesse via CAN dalla ECU (ed in particolare dal *core*).
- Per trasmettere comandi alla scheda *core* in modo diretto, senza la necessità di premere il relativo pulsante sulla scatola.
- Per implementare funzioni e script utilizzati, in particolare, per il calcolo dei cicli eseguiti nelle prove a fatica.

A questo punto si può procedere alla descrizione della sequenza di passaggi che porta alla creazione di un progetto, completo di un'adeguata veste grafica, nell'ambiente *CANape*. Con il termine 'progetto', in questo caso, è da intendersi un'interfaccia software che permette di gestire un insieme di dispositivi elettronici intelligenti riconosciuti come nodi di una rete di comunicazione CAN. Come verrà ben sottolineato dai paragrafi successivi, la gestione dei nodi e del bus CAN ottenibile con *CANape* è veramente completa; ad esempio, il programma consente di definire la giusta corrispondenza tra le sequenze di bit di una *Data Frame* e le informazioni corrispondenti o, ancora, di impostare per ogni nodo la frequenza di lettura dei relativi messaggi trasmessi sul bus, per non parlare dell'utilità delle finestre di calibrazione a cui si è già accennato in precedenza.

In realtà, per la gestione delle nuove centraline per il controllo di attuatori *brushed* DC sono stati creati più progetti *CANape*, ciascuno nato con lo scopo di rispondere ad esigenze applicative specifiche. In generale, ad ogni rete CAN con struttura definita (in termini di nodi connessi e messaggi trasmessi) corrisponde un

singolo progetto; inoltre, a ciascun progetto sono state assegnate diverse configurazioni, le quali permettono di caratterizzarlo con diverse interfacce grafiche, funzioni e altri elementi, pur rimanendo identica la strutturazione della rete e quindi invariato il progetto stesso. Le righe seguenti saranno dedicate alla descrizione, passo per passo e con molte immagini, delle operazioni che hanno portato alla realizzazione del progetto per la gestione delle due centraline contemporaneamente; l'ultima parte presenterà, invece, la configurazione più impiegata del progetto di gestione di una singola centralina alla volta, in quanto si tratta dell'interfaccia utilizzata per tutte le attività di calibrazione e da cui verranno tratte le relative immagini.

Per avviare l'iter di creazione di un nuovo progetto (*project*) è necessario aprire il software *CANape*, in modo diretto o mediante l'icona di un progetto preesistente, e selezionare l'opzione '*New project*…' (fig. 3.11) nel menu a tendina che appare cliccando la voce *File* della barra del menu.

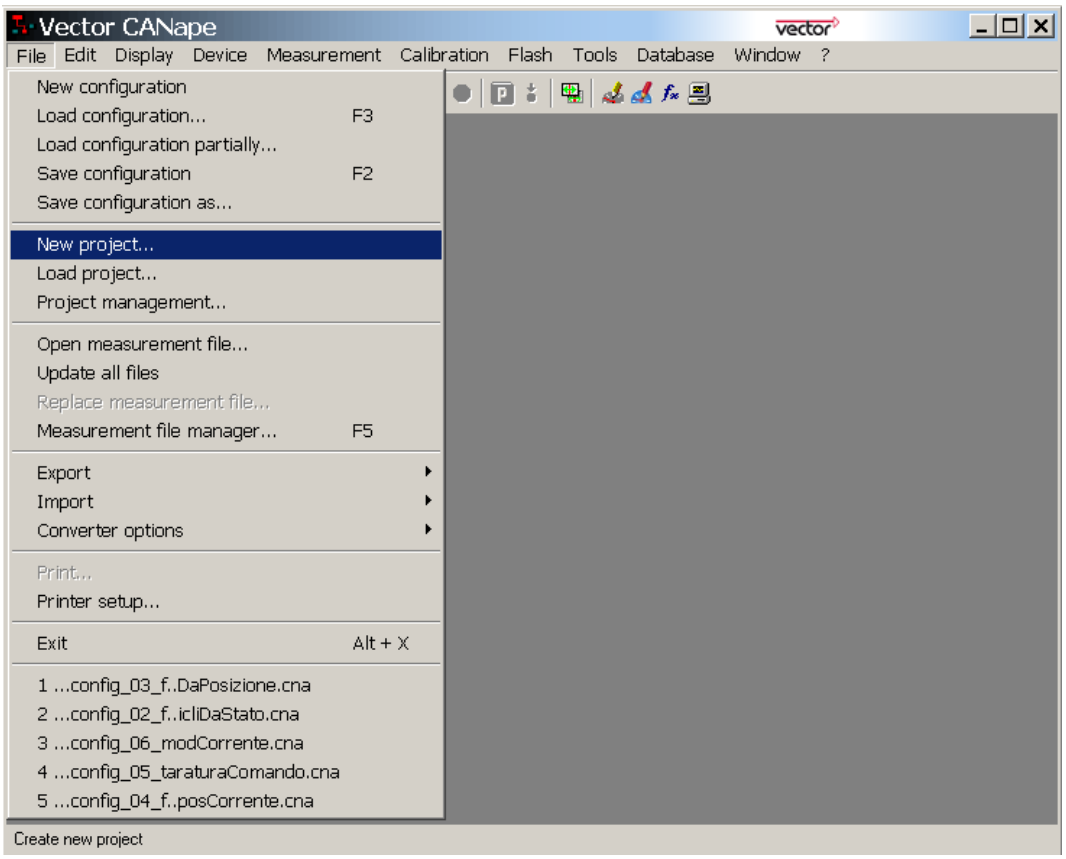


Fig. 3.11: selezione voce 'New project' dal menu File

La selezione effettuata fa partire una procedura guidata in cui viene domandato di specificare il nome del nuovo progetto (fig. 3.12) e una cartella in cui verranno salvati tutti i file importanti legati al progetto (fig. 3.13), tra cui i *datalog* di registrazione. Al progetto è stato assegnato il nome *DoubleECU_Project*.

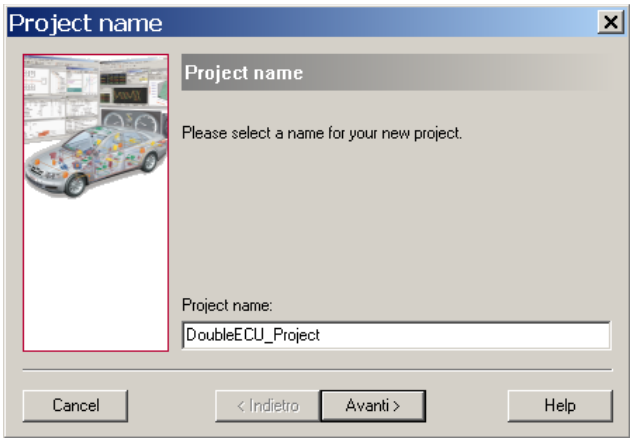


Fig. 3.12: definizione del nome del progetto

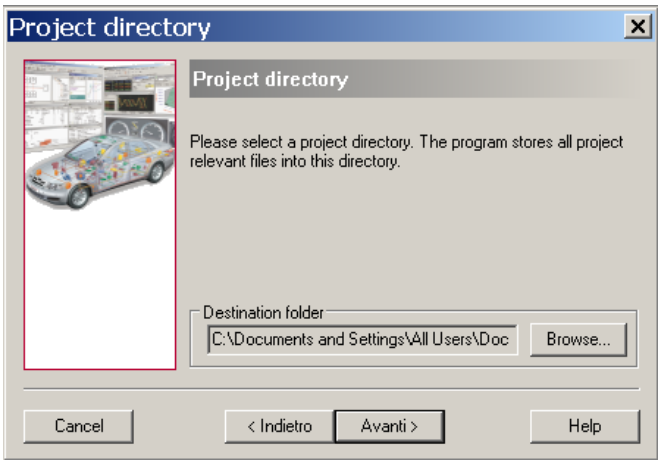


Fig. 3.13: indicazione della cartella di progetto

Il progetto così creato (per cui viene anche generato un collegamento sul desktop) è vuoto al momento, in quanto non sono ancora state specificate le caratteristiche della sua rete CAN. In questo caso, il software dovrà permettere la gestione di due nodi distinti: le due centraline (nominate ECU1 ed ECU2); a questi due si aggiunge un terzo elemento di interfaccia, il *CANcaseXL Log* (indicato semplicemente come *CANcase*). La presenza di questo ultimo dispositivo elettronico è fondamentale per poter usufruire delle potenzialità di *CANape*; esso,

infatti, esegue la conversione da protocollo CAN a USB e viceversa, permettendo al PC dotato di *CANape* di comunicare con le due centraline.

Il *CANcase* impiegato presenta una porta USB per l'interfacciamento con il PC e due porte CAN per quello con le due ECU. La figura successiva schematizza i componenti (nodi più interfaccia) della rete CAN e, in particolare, indica i protocolli di comunicazione impiegati.

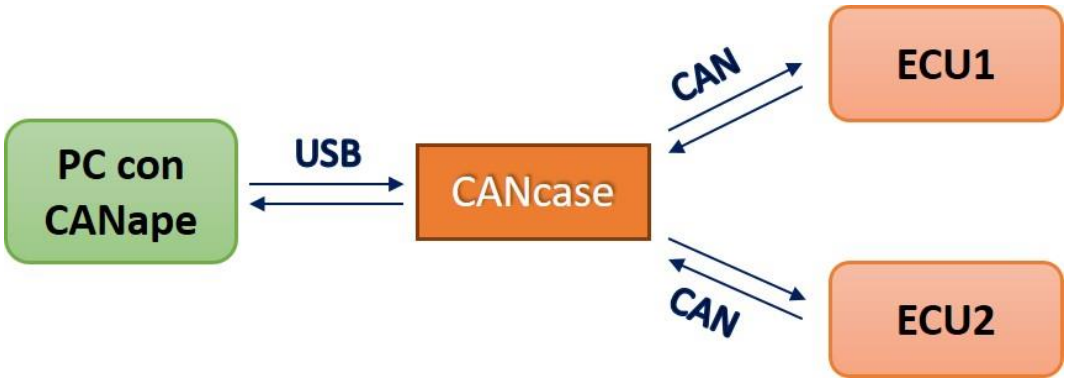


Fig. 3.14: protocolli di comunicazione tra gli elementi della rete CAN

Esistono diversi metodi per far riconoscere un nodo (*device*) ad un progetto e, quindi, renderlo parte della relativa rete CAN. In questo caso è stata scelta la generazione di un nuovo *device* di rete a partire da in database, in quanto era a disposizione il database della generica centralina di controllo *park-lock*; questo file è stato salvato in due copie nella cartella di progetto, poi rinominate e modificate in *CANape* per caratterizzare i due nodi in modo differente l'uno dall'altro, almeno per quanto concerne i nomi dei corrispondenti segnali.

I 'database file' sono dei semplici file di testo contenenti le informazioni necessarie per interpretare correttamente il contenuto dei *Data Frame* ricevuti da un nodo e per generare messaggi CAN verso nodo che esso possa leggere. Tra le tipologie di database più comuni ci sono le seguenti [32]:

- *FIBEX* database file (estensione *.xml*).
- *Vector* database file (estensione *.dbc*).
- *National Instruments CAN* database file (estensione *.ncd*).

In particolare, per ciascuno dei segnali di un *Data Frame* il database file associato al nodo trasmettitore definisce le regole per la conversione da sequenza di bit (numero binario) a valore di significato ingegneristico e con relativa unità fisica di misura. Nel testo di un database file per ogni segnale di ogni messaggio scambiato con un nodo sono salvate le seguenti informazioni:

- Nome del segnale.
- Bit iniziale e dimensione in bit.
- Ordine di disposizione dei byte (*Intel/Motorola*).
- Tipo di dato (con segno, senza segno o IEEE *float*).
- Fattore di scala e stringa con unità di misura.
- Range di variazione.
- Valore di default.
- Eventuale commento.

Le precedenti informazioni consentono al software di convertire agilmente l'informazione 'grezza' portata da un insieme di bit in un valore con significato fisico e comprensibile dall'utente.

Tornando alla configurazione del progetto *DoubleECU_Project*, le figure successive mostrano i passaggi per l'inserimento della ECU1 come *device* di rete a partire da un database file con estensione '*.dbc*'.

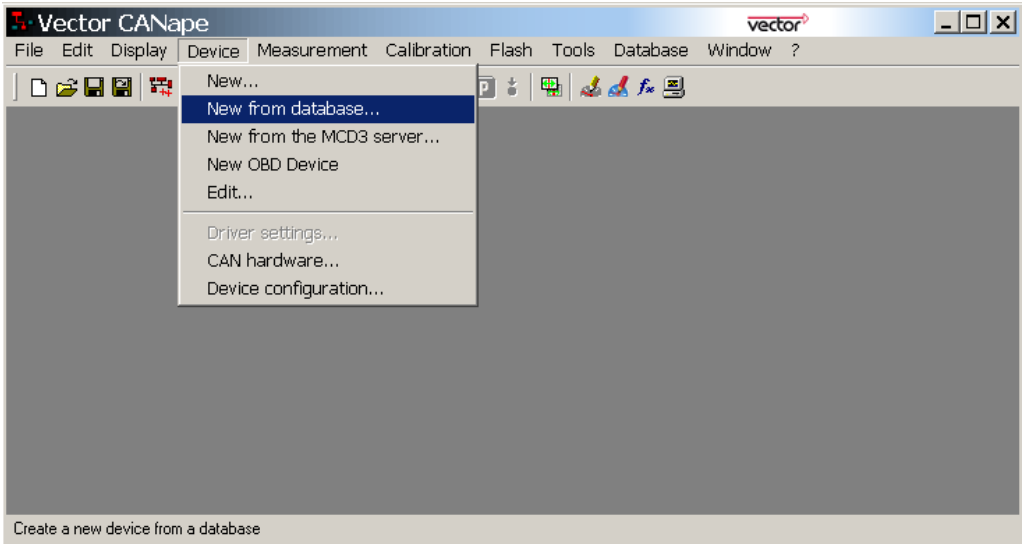


Fig. 3.15: selezione dell'opzione 'New from database…' dal menu Device

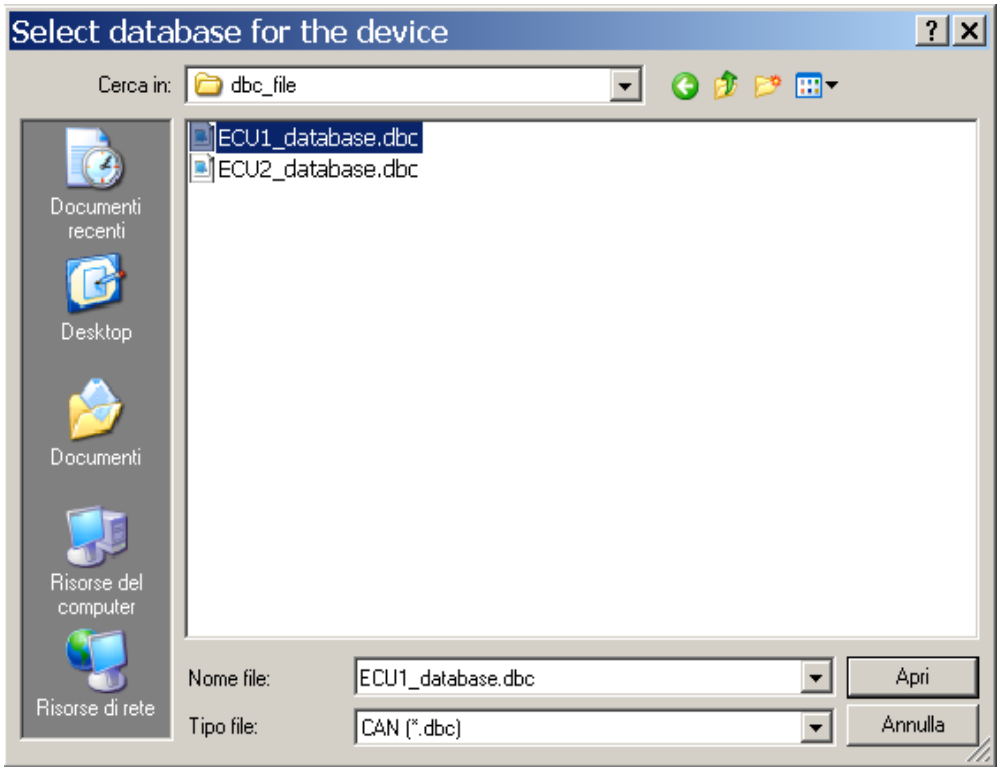


Fig. 3.16: selezione del database file della ECU1 in una sottocartella di progetto

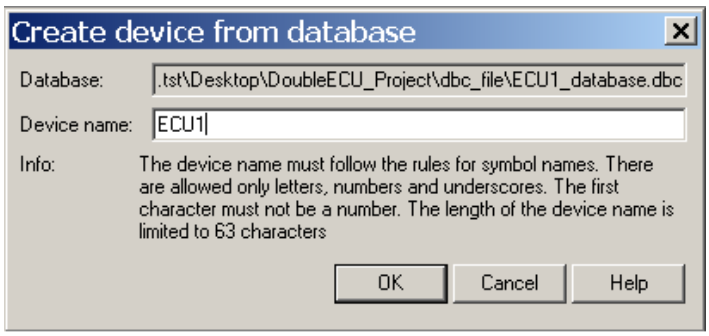


Fig. 3.17: associazione del nome ECU1 al nuovo nodo

A questo punto è necessario configurare la comunicazione tra il dispositivo appena aggiunto e il progetto *CANape*; per realizzare ciò è sufficiente cliccare la voce *Device* della barra del menu e selezionare l'opzione '*Driver settings*…' (fig. 3.18): questa azione determina la comparsa della finestra di figura 3.19.

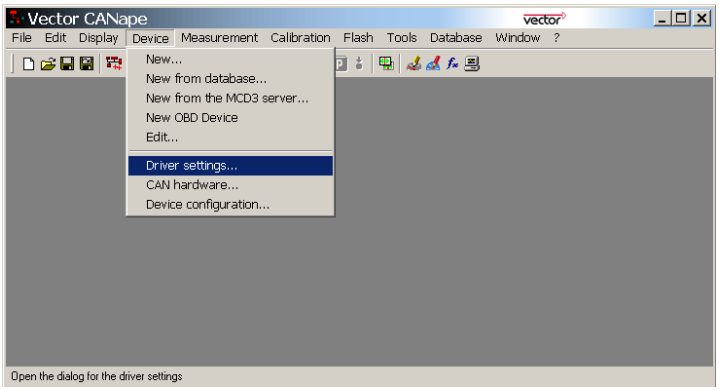


Fig. 3.18: selezione della voce 'Driver settings…' nel menu Device

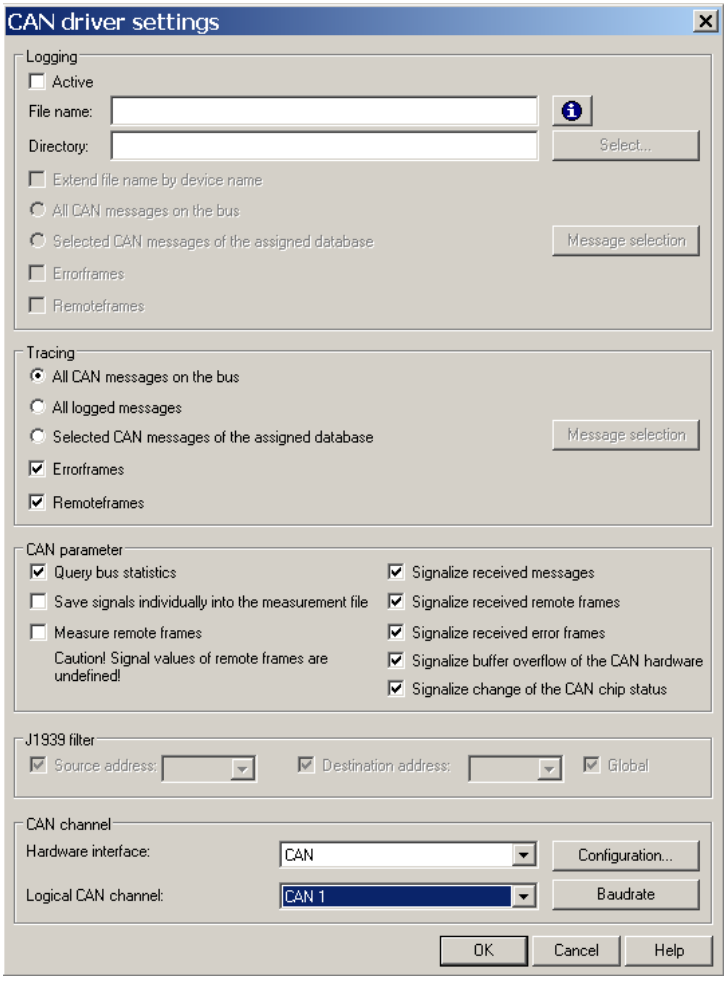


Fig. 3.19: finestra 'CAN driver settings'

Le impostazioni modificabili mediante la finestra precedente sono suddivise in gruppi di pertinenza (*Logging*, *Tracing*, ecc.) e tra questi il più importante è sicuramente: *CAN Channel*. Tale gruppo presenta due menu a tendina:

1) Il primo consente di scegliere il tipo di interfaccia hardware per mezzo della quale la ECU1 si connette alla rete del progetto e ovviamente è stata scelta quella CAN.
2) Il secondo permette di selezionare la porta (o canale) del *CANcase* a cui verrà collegata la ECU1: è stato indicato il *channel* CAN1.

A questo punto, è consigliabile controllare lo stato del *CANcase* per verificare che le impostazioni di comunicazione sia state recepite correttamente dal sistema; per realizzare ciò è sufficiente aprire il menu *Device* e cliccare sulla voce '*CAN hardware…*' (fig. 3.20): apparirà una finestra riassuntiva (fig. 3.21) di tutto ciò che riguarda i componenti di interfaccia.

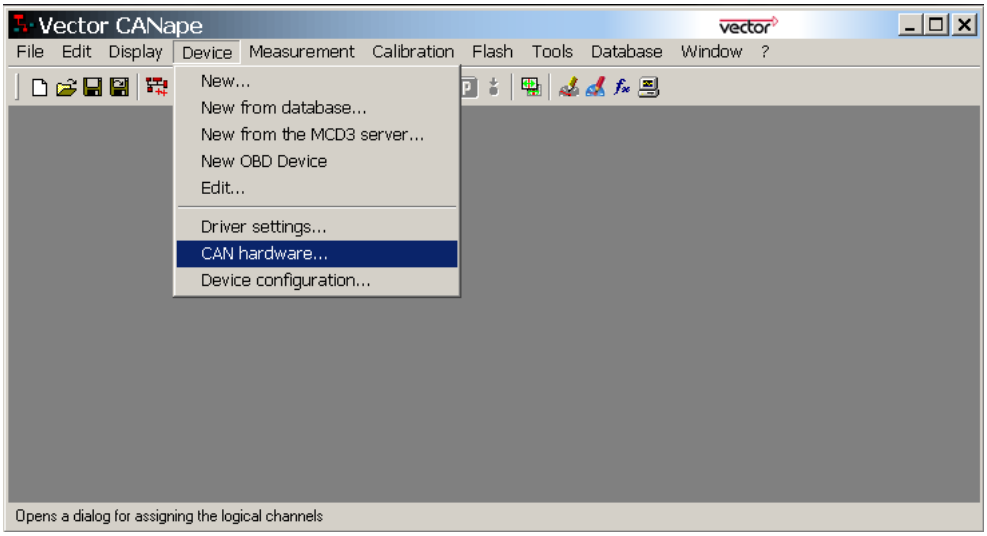


Fig. 3.20: selezione della voce 'CAN hardware' nel menu Device

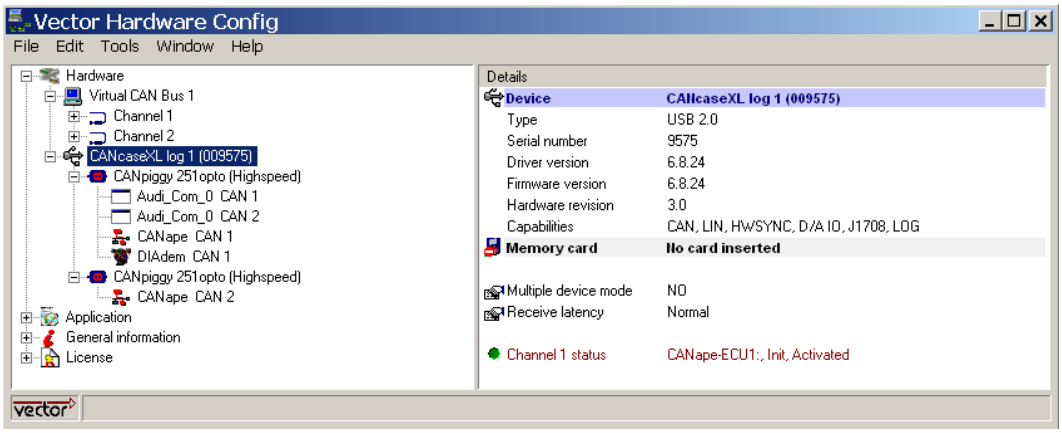


Fig. 3.21: finestra 'Vector Hardware Config'

La porzione di sinistra della finestra precedente elenca una serie di gruppi i cui elementi possono essere contratti o estesi a seconda delle esigenze di visualizzazione; la parte a destra riporta, invece, i dettagli relativi all'elemento che viene selezionato a sinistra. La prima voce in alto a sinistra (*Hardware*) raggruppa tutte le interfacce hardware, reali e simulate, e per ciascuna di esse permette di vedere i canali di comunicazione disponibili. In particolare, il gruppo *Hardware* (le cui voci vengono definite dal componente *Vector* collegato al PC) include:

- *Virtual CAN Bus 1*: si tratta di un simulatore di interfaccia hardware che riproduce via software il collegamento del PC ad un bus a due canali (*Channel* 1 e *Channel* 2). Spesso nascono delle situazioni in cui viene richiesta la realizzazione di un progetto *CANape* completo e la verifica del suo funzionamento senza che i nodi hardware della rete siano disponibili: le interfacce CAN virtuali consentono di impostare la comunicazione con dispositivi fittizi, simulati da apposite applicazioni, che però trasmettono al progetto messaggi del tutto simili a quelli che potrebbero essere prodotti da hardware fisico.
- *CANcaseXL log 1*: è il *CANcase* che realmente interfaccia il progetto con la rete CAN avente le due ECU come nodi. Questo dispositivo ha due porte CAN (*CANpiggy*) e per ciascuna di esse vengono elencate le applicazioni attive (*CANape*, *DIAdem*, ecc.). Siccome la voce in questione è stata selezionata, sulla destra se ne possono osservare i dettagli: nome, tipo di porta USB, numero seriale, protocolli di comunicazione supportati, presenza di *memory card* e così via. Si evidenzia che al fondo della finestra di destra è indicato che il canale 1 è attivo per la connessione con il *device* ECU1 tramite un progetto *CANape*: questo conferma che le operazioni di configurazione precedenti sono andate a buon fine.

Sotto il gruppo '*Hardware'* sulla sinistra si ritrovano: '*Application'* raccoglie tutti i software attivi sull'interfaccia, '*General Information*' le informazioni generali e '*License'* quelle relative alla licenza di utilizzo del dispositivo.

Tornando alla schermata home, è possibile visualizzare lo stato di connessione dei dispositivi associati al progetto selezionando la voce *Device* dalla barra del menu e cliccando '*Device configuration*…' (fig. 3.22). La finestra che appare (fig. 3.23) mostra lo stato di attività delle ECU (in questo caso è presente solo ECU1), il nome, lo stato online/offline, il *driver* di interfacciamento alla rete, la porta di comunicazione con il *CANcase*, il database che ne caratterizza i messaggi; sono inoltre presenti una barra dei menu e una delle applicazioni che consentono di eseguire molte operazioni sui *device* (calibrazione, *flash* di un nuovo programma, ecc) che però non sono state necessarie per il perseguimento dei fini della tesi. Come si può osservare, il nodo ECU1 presenta proprietà coerenti con quanto indicato nei passaggi precedenti.

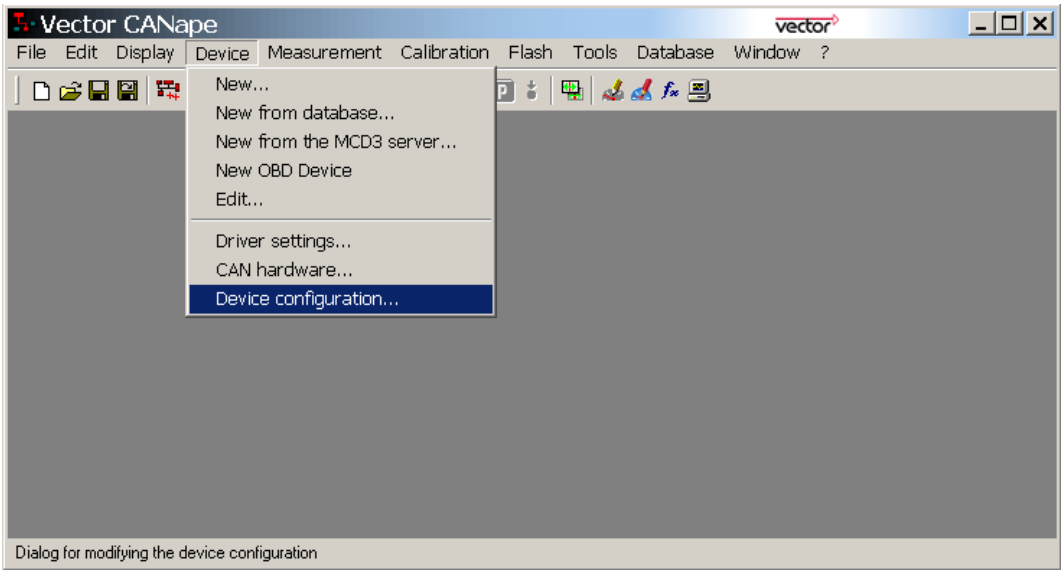


Fig. 3.22: selezione della voce 'Device configuration…' dal menu Device

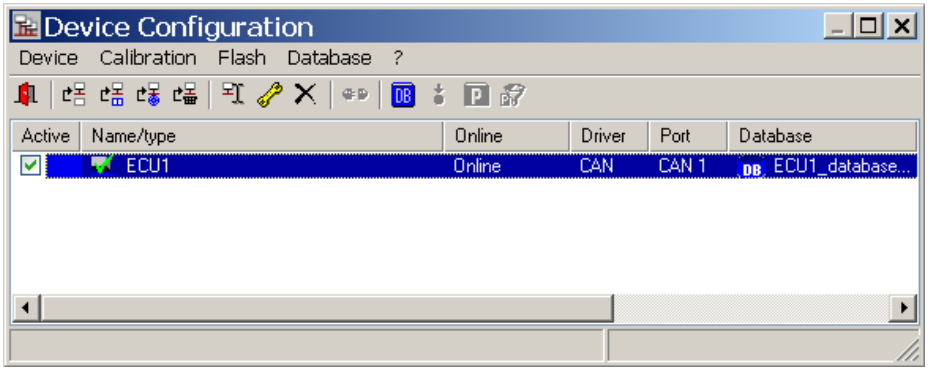


Fig. 3.23: finestra 'Device Configuration'

Il passaggio successivo prevede la verifica ed eventualmente la modifica/integrazione delle informazioni contenute nel database correlato al nodo appena aggiunto alla rete, il quale ne caratterizza completamente la comunicazione. Per realizzare ciò, una volta raggiunta la schermata principale del progetto, è sufficiente cliccare *Database* sulla barra del menu e quindi '*Editor*…' oppure si può selezionare l'apposita icona della barra delle applicazioni (fig. 3.24).

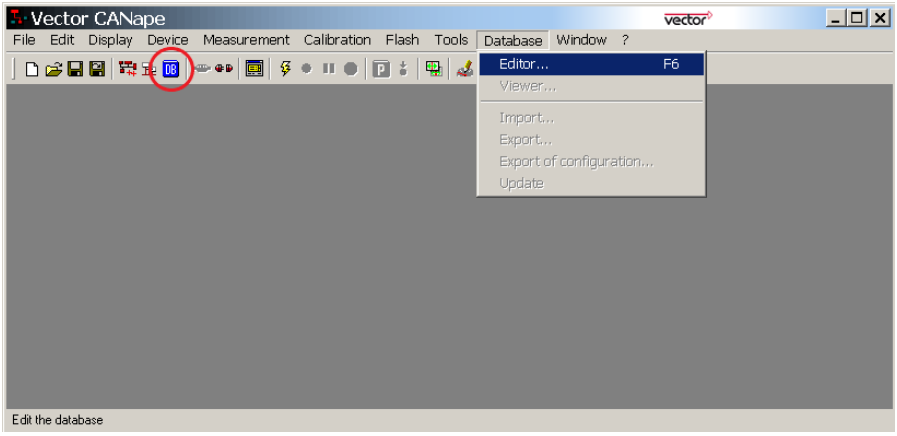


Fig. 3.24: selezione della voce 'Editor' dal menu Database

A seguito della selezione effettuata si apre la schermata dell'editor dei database integrato in *CANape* (fig. 3.25).

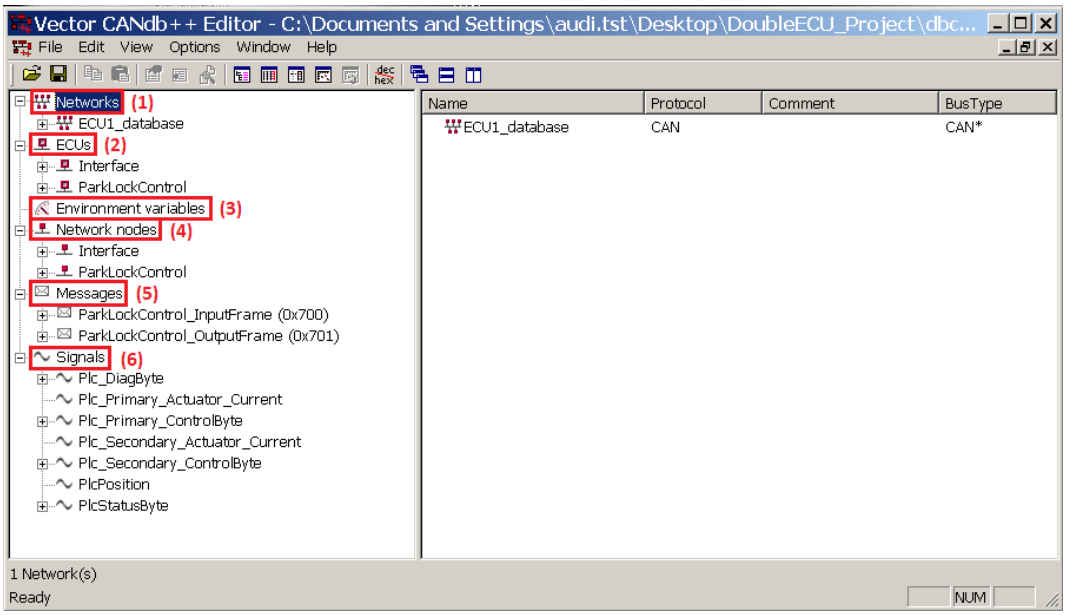


Fig. 3.25: finestra 'Vector CANdb++ Editor

Nella porzione di sinistra della finestra si trovano le seguenti voci, nella figura precedente sono cerchiate e numerate in rosso:

1) *Networks*: raggruppa le reti di comunicazione CAN che sono associate al database.
2) *ECUs*: include i dispositivi di controllo riconosciuti dal database.
3) *Environment variables*: riunisce le variabili relative all'ambiente del dispositivo del database.
4) *Network nodes*: include tutti i nodi della rete CAN considerati dal database.
5) *Messages*: raggruppa i messaggi CAN associati al database.
6) *Signals*: contiene i segnali CAN dei messaggi dell'elenco precedente e la cui decodifica dipende totalmente dal suddetto database.

Il database della ECU1 definisce le caratteristiche del dispositivo intelligente che, come si vede in fig. 3.25, è indicato con il generico nome di *ParkLockControl*. Per chiarezza, il nome della ECU e dei due messaggi ad essa associati (uno in ingresso e uno in uscita) previsti dal database sono stati modificati; inoltre, al nome dei segnali è stato aggiunto il pedice '1' per indicare che si riferiscono al controllo della ECU1 e non creare confusione quando verrà aggiunta la seconda ECU (fig. 3.26).

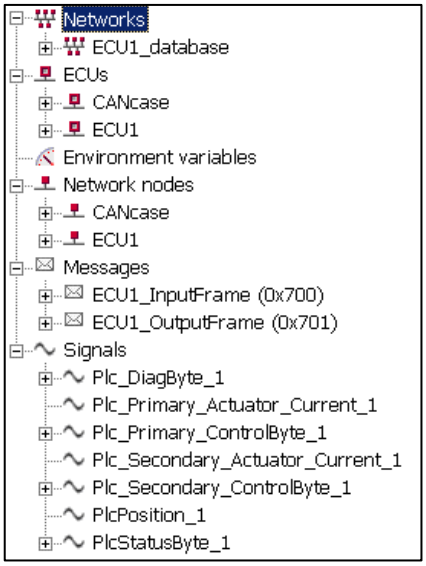


Fig. 3.26: nomi aggiornati di ECU, nodi, messaggi e segnali

Risulta quindi opportuno soffermarsi sulle caratteristiche e sulle proprietà dei messaggi e dei segnali scambiati tra il computer e la centralina 1, sempre grazie all'interfacciamento e alla conversione di protocollo realizzata dal *CANcase*. Le due immagini che seguono mostrano il contenuto del messaggi relativi alla ECU1 e permettono di realizzare qualche importante osservazione.

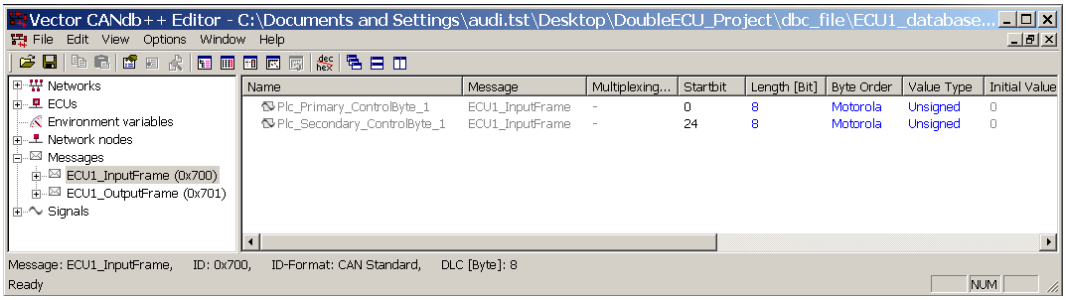


Fig. 3.27: segnali del messaggio in uscita dal PC verso la ECU1

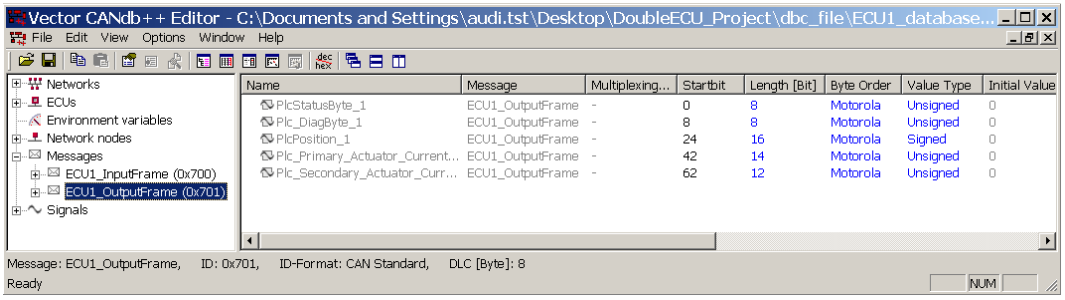


Fig. 3.28: segnali del messaggio in uscita dalla ECU1 verso il PC

Partendo dall'immagine 3.27, si osserva che i segnali portati dal messaggio CAN trasmesso dal *CANcase* alla ECU1 (di nome *ECU1_InputFrame*) sono:

- *Plc_Primary_ControlByte_1*: segnale per il controllo dell'attuatore principale della ECU1.
- *Plc_Secondary_ControlByte_1*: segnale per il controllo dell'attuatore secondario della ECU1.

Invece, nella figura 3.28 vengono presentati i segnali inclusi nel messaggio CAN inviato dalla ECU1 al *CANcase* e da questo trasmesso al computer:

- *PlcStatusByte_1*: segnale che riporta lo stato di funzionamento della ECU1.
- *Plc_DiagByte_1*: segnale che riporta lo stato di errore della ECU1.

- *PlcPosition_1*: segnale con il *duty cycle* di posizione dell'attuatore del *park-lock* controllato dalla ECU1.
- *Plc_Primary_Actuator_Current_1*: segnale con la corrente assorbita dell'attuatore principale del *park-lock* controllato dalla ECU1.
- *Plc_Secondary_Actuator_Current_1*: segnale con la corrente assorbita dell'attuatore secondario del *park-lock* controllato dalla ECU1.

Il contenuto dei messaggi corrisponde esattamente con quanto descritto nel sotto-capitolo 3.2. Selezionando un messaggio sulla sinistra, sulla destra non vengono solo mostrati i nomi dei segnali che lo costituiscono ma per ciascuno di essi vengono riportate delle informazioni molto importanti che li localizzano all'interno del messaggio stesso; in particolare, per ogni segnale viene indicato il bit di partenza (*Startbit*), la lunghezza in bit (*Lenght* [bit]) e l'ordine dei byte nel messaggio (*Byte Order*). Nel capitolo 2 è già stata sottolineata l'importanza di verificare la corrispondenza tra le informazioni sulla struttura dei messaggi di input e output contenute nel database e quelle indicate esplicitamente nei sorgenti della scheda della centralina che gestisce la comunicazione CAN.

Per disporre una panoramica completa di come i segnali vengano scorporati dal messaggio di appartenenza e trasformati in valori con significato fisico è necessario selezionare la voce *Signals* nell'interfaccia dell'editor e osservare il contenuto della finestra di destra (3.29).

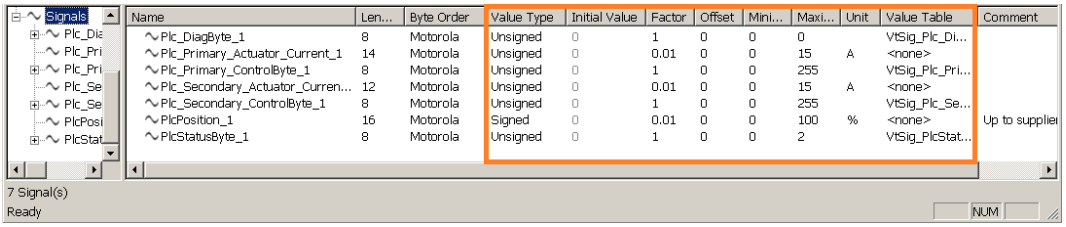


Fig. 3.29: proprietà dei segnali gestiti dal progetto CANape

Il rettangolo arancione nella figura precedente evidenzia una serie di attributi che caratterizzano ciascun segnale:

- *Value Type*: è il tipo del valore corrispondente alla nuda lettura della sequenza di bit (intero con segno o senza segno).
- *Initial Value*: è il valore iniziale del segnale fisico letto.
- *Factor*: è il fattore di scala che serve per passare dal valore intero rappresentato dai bit a quello di significato fisico visualizzato tramite l'interfaccia ed eventualmente memorizzato.
- *Offset*: è l'offset tra il valore in uscita rispetto a quello letto da segnale.
- *Minimum*: è il valore minimo indicato per il valore in uscita.
- *Maximum*: è il valore massimo indicato per il valore in uscita.
- *Unit*: è l'unità fisica del valore letto.
- *Value Table*: se necessario, indica la tabella di corrispondenza tra il valore del segnale letto e una stringa di caratteri da visualizzare a schermo (è il caso delle informazioni di stato, di *fault* e dei segnali di comando).

La compilazione corretta dei campi sopra citata dipende dal tipo di segnale considerato e del risultato delle procedure di calibrazione attuate per correlare nel modo corretto il valore letto tramite interfaccia e quello reale (valutato con uno strumento di misura di riferimento) della grandezza fisica di interesse; la calibrazione delle letture di posizione e di corrente verrà presentata nel capitolo 5, riservato alle attività di perfezionamento del sistema di controllo e di interfaccia.

Giunti a questo punto, il progetto *CANape* '*DoubleECU_Project*' risulta configurato (non ancora a livello grafico) per avviare la comunicazione con la ECU1, sia dal punto di vista hardware che da quello software e, soprattutto, per quanto riguarda l'interpretazione dei messaggi CAN. Si ricorda, però, che si desidera implementare un unico progetto di interfaccia, il quale sia in grado di gestire contemporaneamente l'esecuzione delle prove di fatica di due *park-lock*, uno per ogni centralina. Di conseguenza, è stato necessario far riconoscere a '*DoubleECU_Project*' e quindi caratterizzare un secondo dispositivo intelligente (la ECU2), anch'esso comunicante con il PC mediante lo stesso *CANcase* di interfaccia della ECU1. Pertanto, il database della ECU1 stato duplicato e

rinominato '*ECU2_database*' e sono state ripetuti passaggi descritti fino a questo punto, dall'aggiunta di un nuovo *device* al progetto all'aggiornamento del suo database.

Le figure successive testimoniano l'inserimento del nuovo dispositivo nel progetto (fig. 3.30 e fig. 3.31) e ne mostrano il database modificato (fig. 3.32).

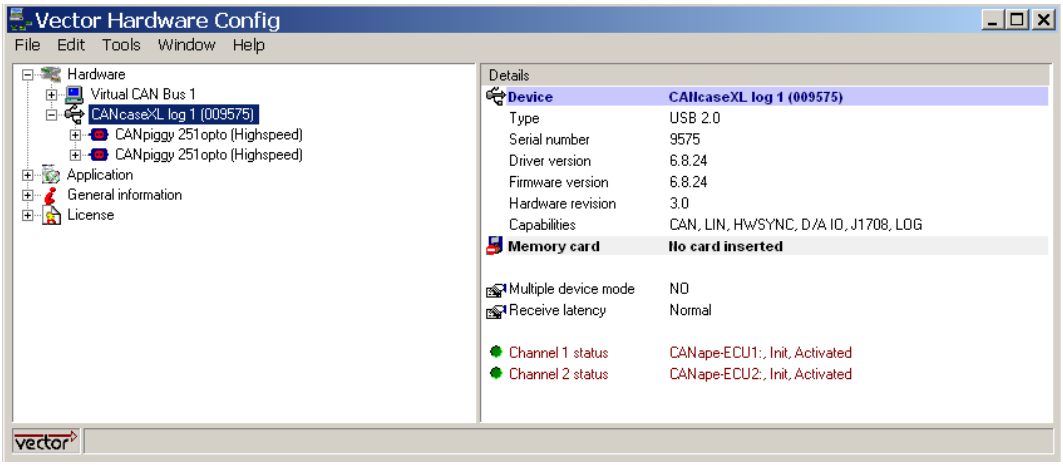


Fig. 3.30: riconoscimento della ECU2 da parte del Vector hardware

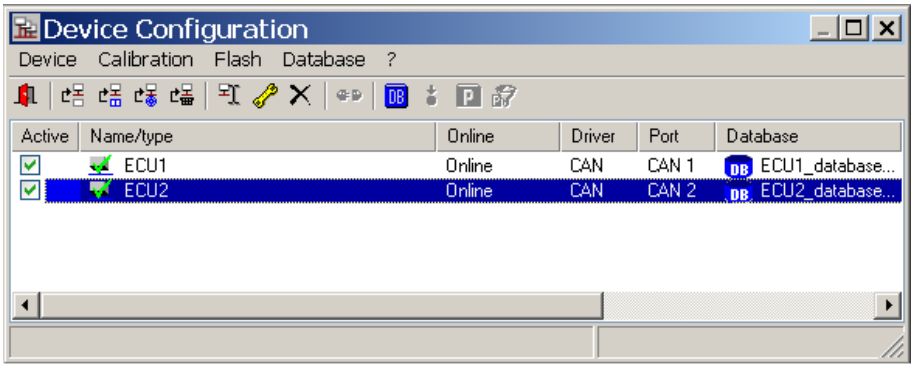


Fig. 3.31: riconoscimento della ECU2 come dispositivo configurabile

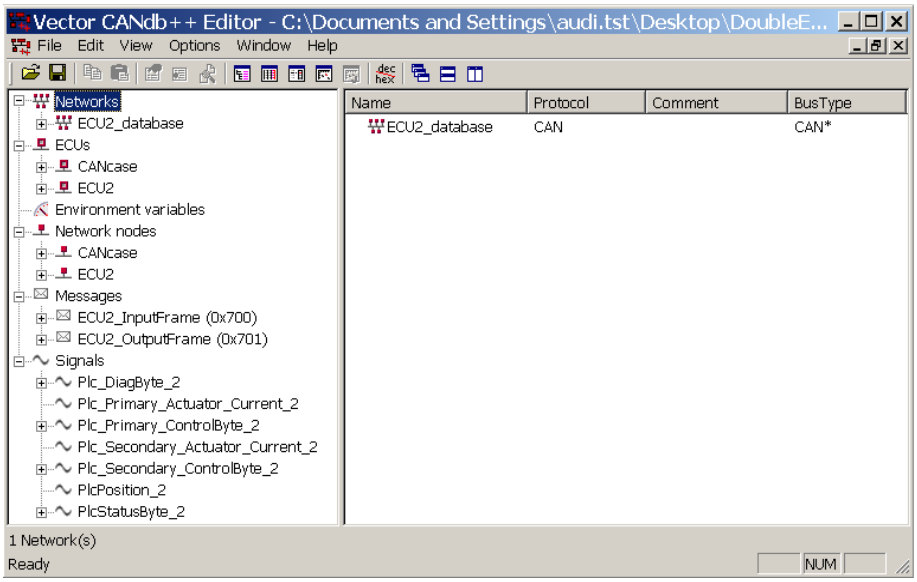


Fig. 3.32: database della ECU2 aggiornato

Prima di procedere alla creazione della parte grafica del progetto, è necessario definire le impostazioni inerenti alla lettura e alla memorizzazione dei segnali CAN. Per fare ciò, raggiunta la schermata principale del progetto bisogna cliccare F4 o aprire la scheda del menu *Measurement* e selezionare la voce '*Measurement configuration*…' (fig. 3.33): si aprirà l'interfaccia di configurazione delle impostazioni di misura (fig. 3.34).

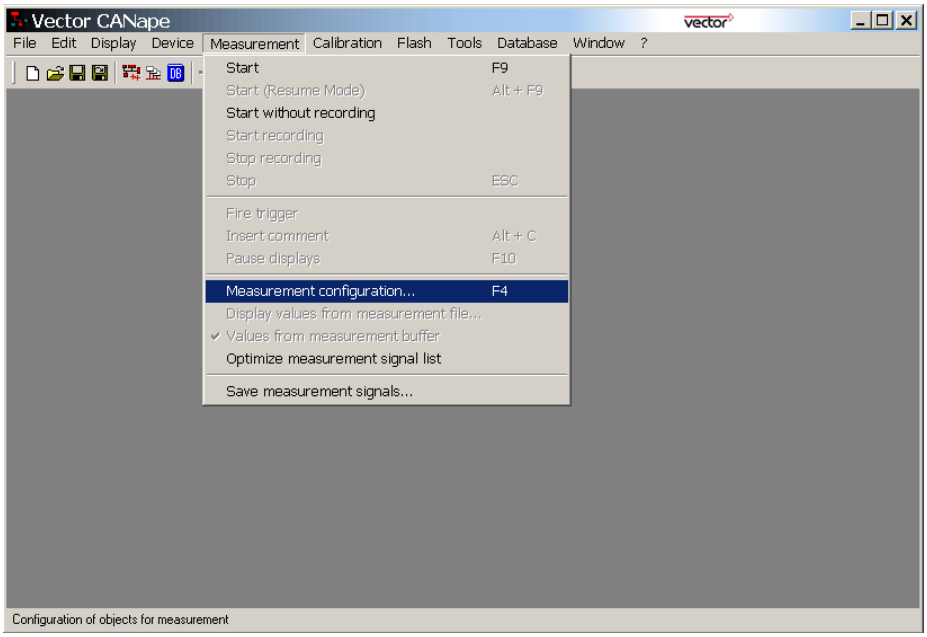


Fig. 3.33: selezione della voce 'Measurement configuration…' dal menu Measurement

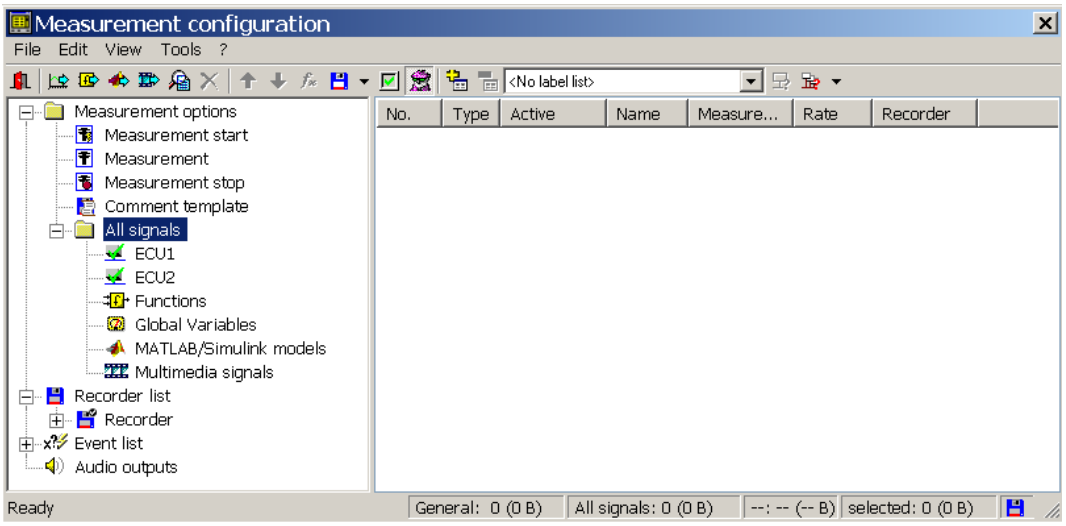


Fig. 3.34: finestra 'Measurement configuration'

L'interfaccia di figura 3.34 mostra, sulla sinistra, quattro cartelle raggruppanti le vari voci di modifica che riguardano la lettura dei due bus CAN (uno per ECU) e la registrazione dei relativi dati:

- *Measurement options*: permette di definire tutte le proprietà che concernono la lettura del valore dei segnali dal bus CAN. Infatti, non è detto che l'inizio, la fine e il periodo di misura di un dato segnale corrispondano a quelli di trasmissione dello stesso; l'identificazione delle caratteristiche della misura dipende dalle esigenze di memorizzazione e di lettura legate allo specifico segnale. La sotto-cartella '*All signals*' raggruppa tutti i segnali misurati dal progetto, i quali sono anche suddivisi per provenienza (dispositivo, funzione, spazio delle variabili globali, modello *MATLAB*/*Simulink*, canale multimediale).
- *Recorder list*: è la lista dei possibili percorsi di registrazione (*recorder*) disponibili per la memorizzazione dei dati sotto forma di *datalog*; ciascuno dei percorsi viene caratterizzato con un nome, un template di commento, un percorso file di salvataggio, ecc. Disporre di più *recorder* può risultare molto utile qualora si desideri monitorare applicazioni diverse con lo stesso progetto *CANape*: ad esempio,

consente creare *datalog* di registrazione aventi radice del nome differente a seconda del tipo di prova di fatica.

- *Event list*: raggruppa tutti gli eventi definiti dall'utente con lo scopo di rendere più semplice e completa la gestione dell'interfaccia; ad esempio, si può impostare che un parametro globale sia resettato alla premuta di una specifica combinazione di tasti del PC.
- *Audio outputs*: include tutti i segnali audio che sono stati introdotti per segnalare acusticamente l'avvenimento di eventi di interesse.

Come si può osservare nella figura 3.34, al momento il progetto non prevede la misura di alcun segnale in quanto è appena stato creato. Per aggiungere segnali alla lista è sufficiente cliccare sul nome di un dispositivo per accedere al suo database, selezionare i segnali da inserire (fig. 3.35) tra quelli da memorizzare e infine premere invio (fig. 3.36).

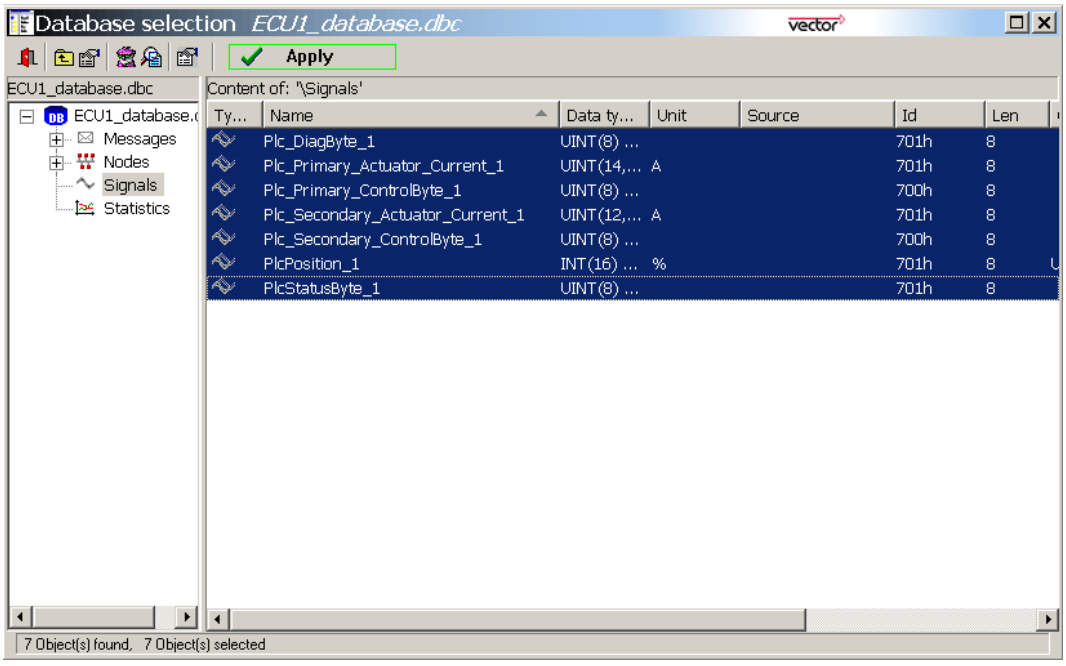


Fig. 3.35: selezione delle variabili da memorizzare nel database della ECU1

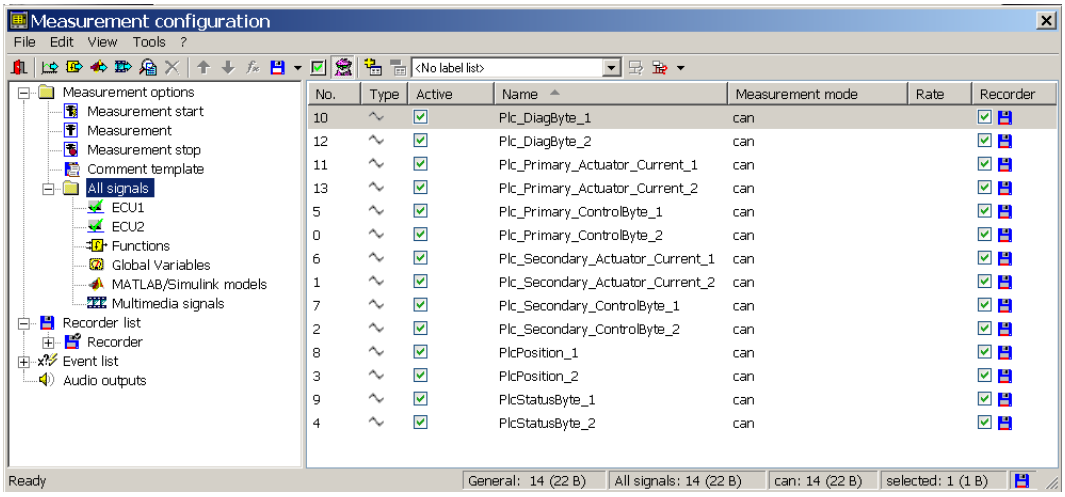


Fig. 3.36: segnali da memorizzare contenuti nella cartella 'All signals'

Per ciascuno dei segnali aggiunti all'elenco di misura del progetto vengono indicati i seguenti campi:

- Un numero identificativo progressivo.
- Il tipo, che ne discrimina l'origine (dispositivo, funzione, …).
- L'effettiva attivazione della sua misura.
- Il nome indicato nel database.
- La modalità di misura, ossia di lettura del valore; un segnale generato da un dispositivo fisico può essere letto a frequenza fissa definita dall'utente o alla stessa frequenza con cui viene trasmesso sul bus CAN dal dispositivo di origine: per l'applicazione di interesse è stata scelta la seconda possibilità per tutti i segnali coinvolti, in quanto si desidera che il tasso di lettura sia il più elevato possibile.
- Il periodo di lettura, mostrato solo per i segnali letti ad una frequenza definita dall'utente.
- L'attivazione della sua memorizzazione secondo un determinato percorso di registrazione.

La scheda '*Functions*' della cartella '*All Signals*' permette di visualizzare la funzioni che sono state definite in associazione al progetto e di specificarne le caratteristiche di esecuzione e memorizzazione. La figura 3.37 mostra le cinque funzioni che sono state introdotte per ampliare l'efficacia di '*DoubleECU_Project*'.

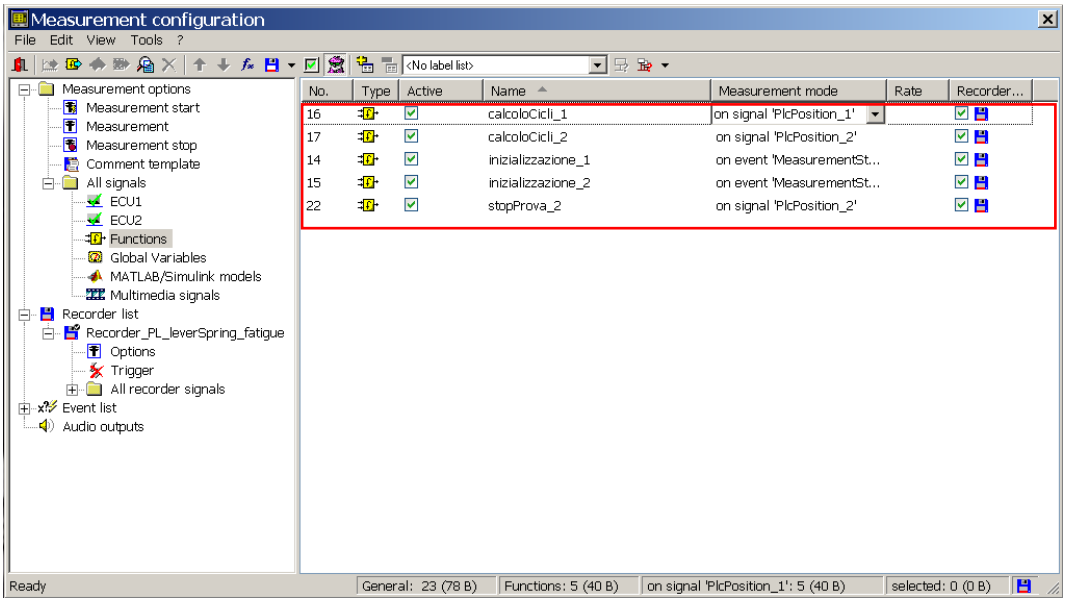


Fig. 3.37: funzioni di progetto raggruppate nella cartella 'Functions'

Le funzioni associate al progetto di interfaccia sono le seguenti:

- *calcoloCicli_1*: valuta il numero di cicli realizzati sia rispetto alla singola prova che in totale dall'attuatore controllato dalla ECU1.
- *calcoloCicli_2*: è come la funzione sopra ma per la ECU2.
- *inizializzazione_1*: definisce il valore iniziale delle variabili globali impiegate dalle funzioni che riguardano la ECU1.
- *inizializzazione_2*: è come la funzione sopra ma per la ECU2.
- *stopProva_2*: ferma la prova gestita dalla ECU2 quando viene raggiunto un valore limite di cicli totali, trasmettendo alla centralina un byte di controllo con il valore esadecimale (0x5a) che viene da essa identificato come un comando di arresto.

È interessante osservare la modalità di misura ('*Measurement mode*') delle varie funzioni, la quale ne definisce le caratteristiche di esecuzione e di rilevazione dei risultati. Le due funzioni di inizializzazione vengono eseguite solo una volta per prova, all'avvio della registrazione, in quanto attribuiscono alle variabili globali (soglie di innesto/disinnesto, *flag*, contatori) i valori iniziali che costituiscono il dato di partenza per l'esecuzione delle altre funzioni. Invece, le due funzioni di calcolo cicli sono eseguite in modo sincrono rispetto alla lettura del *duty cycle* di posizione

del relativo attuatore; questo perché, in entrambi i casi, l'adempimento di un ciclo viene identificato proprio sulla base della posizione; lo stesso vale per la funzione di stop prova.

Conclusa la parte di spiegazione e di definizione delle caratteristiche di misura e memorizzazione delle variabili di progetto, è possibile procedere con la descrizione dell'interfaccia grafica per l'utente (detta anche GUI, ossia *Graphical User Interface*), costruita principalmente per consentire l'osservazione dei parametri di interesse relativi alle prove di fatica controllate; inoltre, la GUI implementata permette di inviare comandi alle centraline in modo diretto e di consultare i messaggi con cui *CANape* aggiorna l'utente sullo stato di registrazione, sulla presenza di errori interni e così via.

Il primo passo verso la realizzazione di un'interfaccia consiste nel creare una configurazione per il progetto; le configurazioni costituiscono una feature molto importante di *CANape* poiché consentono di introdurre caratteristiche di interfacciamento specifiche (grafiche e non solo), pur mantenendo inalterate le proprietà di base del progetto (dispositivi comunicanti, database, …). Per creare una nuova configurazione di lavoro è sufficiente cliccare l'icona *File* nella barra dei menu e scegliere la voce '*New configuration*' (fig. 3.38).

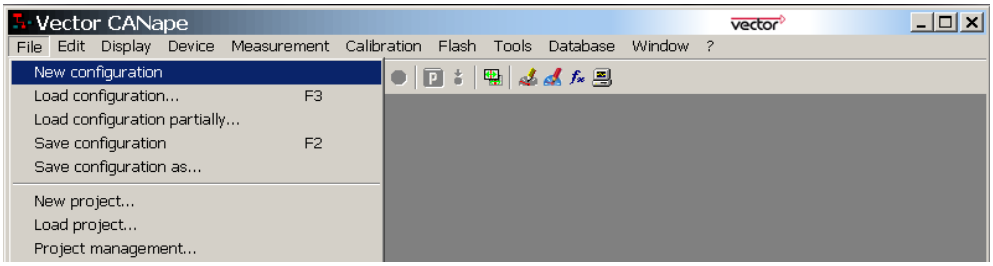


Fig. 3.38: selezione della voce 'New configuration' dal menu File

Una volta che la configurazione è stata memorizzata la schermata principale del progetto continuerà a rimanere di colore grigio. Per aggiungere elementi grafici è sufficiente cliccare con il tasto destro del mouse in uno spazio libero della schermata: apparirà così un menu con tutti i tipi di finestre aggiungibili, da quelle che servono semplicemente a visualizzare i dati a quelle che consentono dialogare con applicazioni specifiche. Ad esempio, se si desidera inserire una finestra grafica

per l'osservazione dell'andamento dei *dc* di posizione trasmessi dalle due ECU, allora bisogna scegliere prima '*Display window*' e dopo '*Graphic window*' (fig. 3.39).

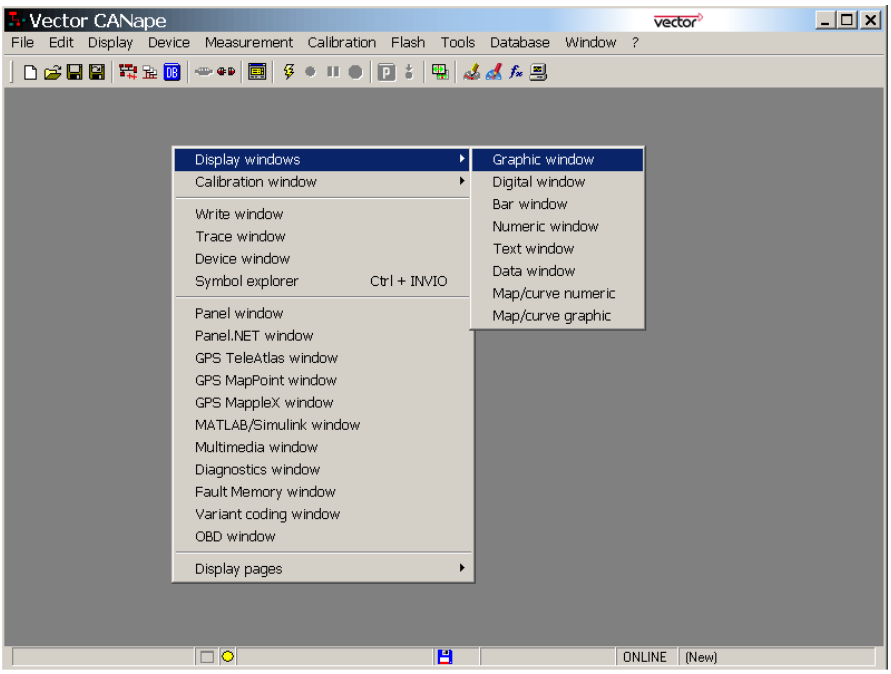


Fig. 3.39: selezione della voce 'Display windows'

Quindi, facendo click con il destro nella finestra vuota che è comparsa e selezionando '*Insert measurement signal…*' (fig. 3.40) si aprirà l'interfaccia di '*Measurement configuration*' per consentire la scelta dei segnali da visualizzare (fig. 3.41).

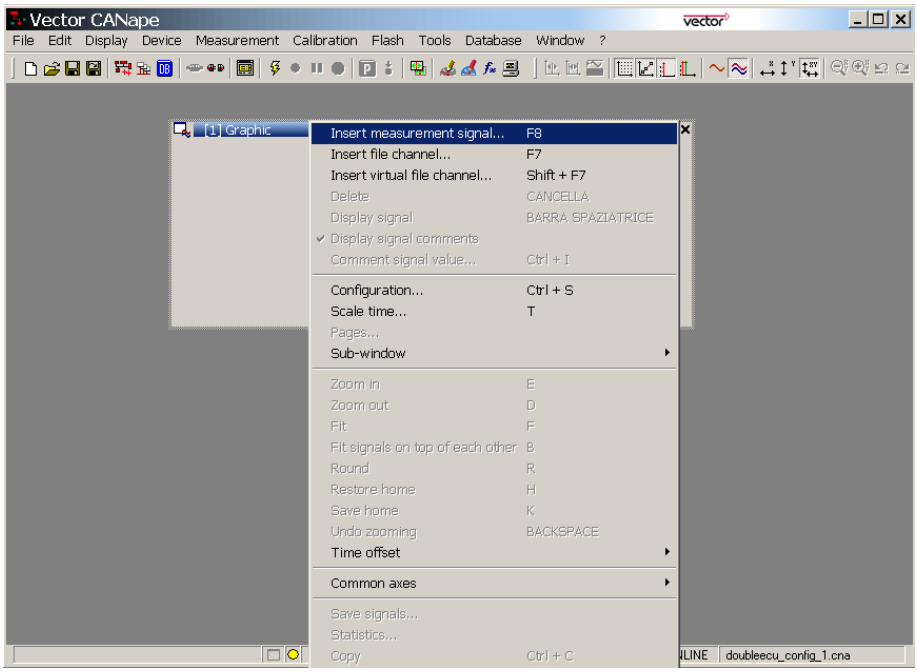


Fig. 3.40: selezione della voce 'Insert measurement signal…'

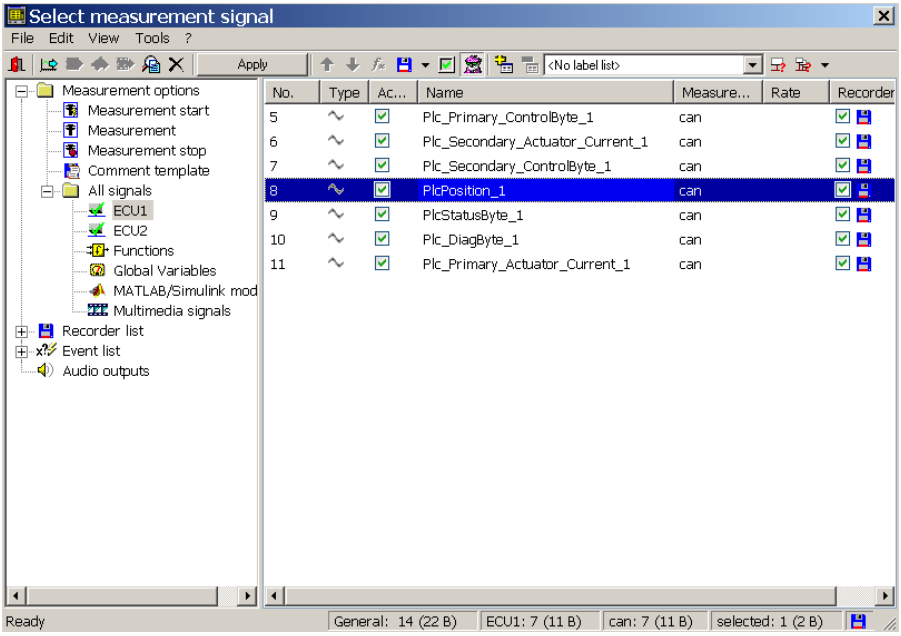


Fig. 3.41: interfaccia per la selezione dei segnali da aggiungere alla finestra grafica

Infine, selezionando i segnali *PlcPosition_1* e *PlcPosition_2* nella cartella '*All signals*', premendo il pulsante di invio e uscendo dall'interfaccia, nella schermata principale si ritroverà una finestra grafica predisposta per la visualizzazione dei due andamenti di posizione (fig. 3.42).

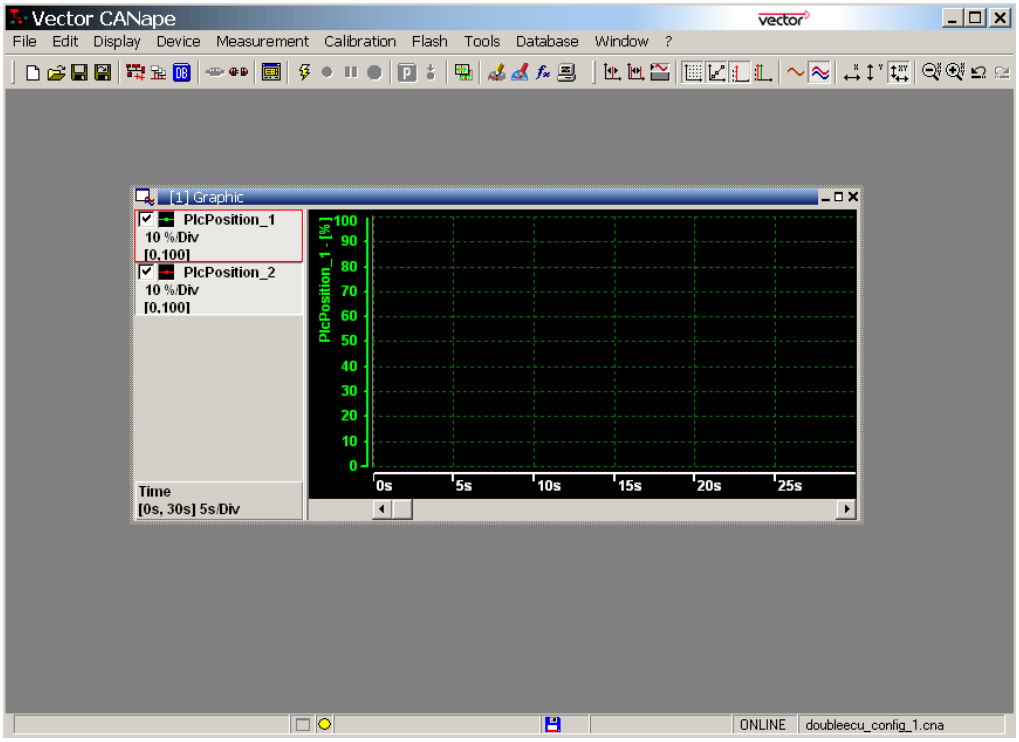


Fig. 3.42: finestra grafica inserita nella configurazione CANape

Quello di figura 3.42 è solo un esempio dei diversi elementi che *CANape* consente di associare ad una specifica configurazione; se la configurazione viene salvata, al suo successivo caricamento il layout e le impostazioni definite in precedenza verranno completamente ritrovate. *CANape* dà la facoltà di creare molte configurazioni per lo stesso progetto: in questo modo sarà possibile conferirgli una veste grafica e funzionale differente a seconda dell'applicazione. Siccome si ritiene di aver illustrato sufficientemente nel dettaglio le nozioni fondamentali per creare una generica interfaccia grafica su *CANape*, la figura 3.43 presenta la configurazione del progetto '*DoubleECU_project*' costruita per gestire appropriatamente le prove di fatica del *park-lock*.

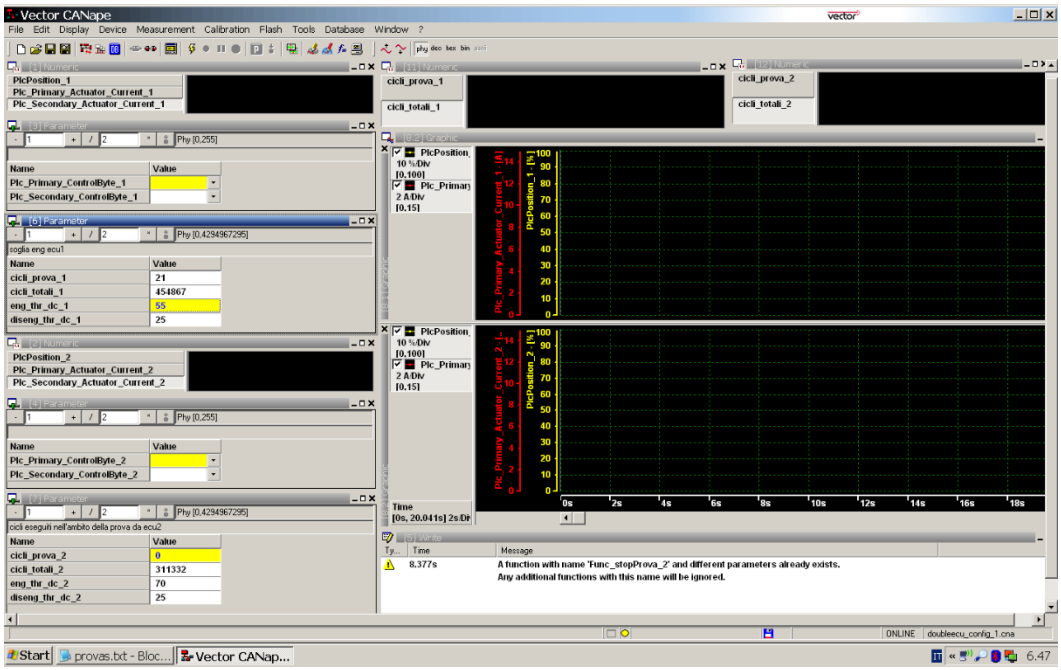


Fig. 3.43: interfaccia grafica per il controllo delle prove di fatica park-lock

Le finestre che caratterizzano la configurazione precedenti sono:

- Due finestre numerica ('*Numeric*') che presentano i valori attuali di posizione, corrente all'attuatore principale e corrente all'attuatore ausiliario per ciascuna delle due ECU (figg. 3.44 e 3.45).

Fig. 3.44: finestra numerica per ECU1

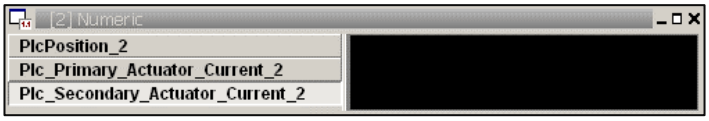


Fig. 3.45: finestra numerica per ECU2

- Due finestra di calibrazione ('*Parametric*') che consentono di specificare i valori dei byte di controllo degli attuatori collegati alle ECU 1 e 2, anche se in realtà nell'applicazione di interesse sono presenti solo gli attuatori MAIN (figg. 3.46 e 3.47).

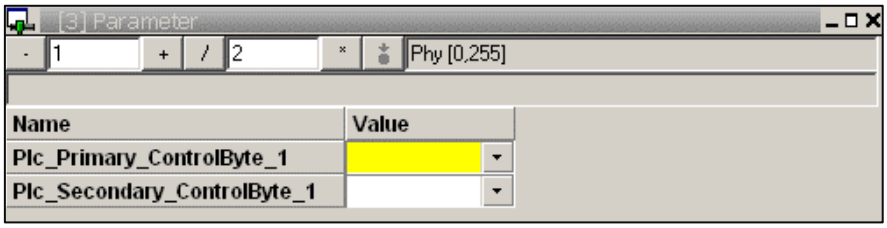


Fig. 3.46: finestra di comando per ECU1

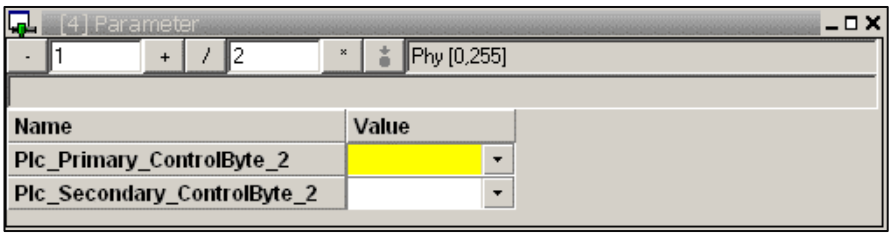


Fig. 3.47: finestra di comando per ECU1

- Due finestre di calibrazione ('*Parametric*') per la definizione manuale dei cicli prova, dei cicli totali e delle soglie di innesto e disinnesto che costituiscono il riferimento per il calcolo dei cicli (figg. 3.48 e 3.49). I 'cicli prova' sono quelli della singola registrazione, mentre 'cicli totali' identifica quelli complessivamente compiuti dal *park-lock* in prova, somma di tutti i cicli prova. I cicli prova sono azzerati automaticamente all'avvio di ogni registrazione e non dovrebbero mai essere modificati manualmente; i cicli totali, al contrario, necessitano dell'azione dell'utente per l'azzeramento. Anche le soglie vengono inizializzate in automatico quando la prova è avviata, però può essere opportuno cambiarle in corso prova in casi particolari.

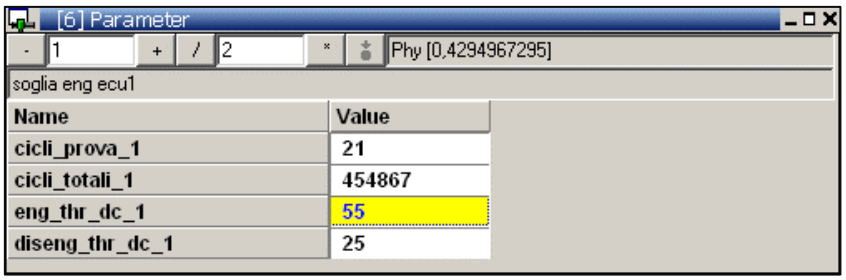


Fig. 3.48: finestra di calibrazione per ECU1

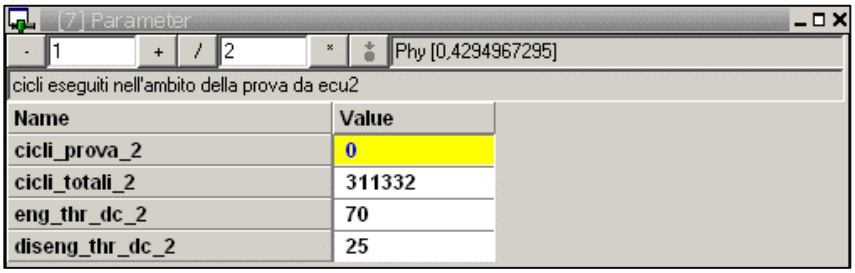


Fig. 3.49: finestra di calibrazione per ECU2

- Due finestre numeriche ('*Numeric*') affiancate per osservare il numero di cicli portati a termine dalle due centraline (fig. 3.50).

Fig. 3.50: finestre numeriche per l'osservazione del numero di cicli eseguiti

- Una finestra grafica ('*Graphic*') suddivisa in due sottofinestre: quella superiore permette di visualizzare gli andamenti di *dc* di posizione e di corrente assorbita dal primario per la ECU1, quella inferiore gli stessi andamenti per la ECU2 (fig. 3.51).

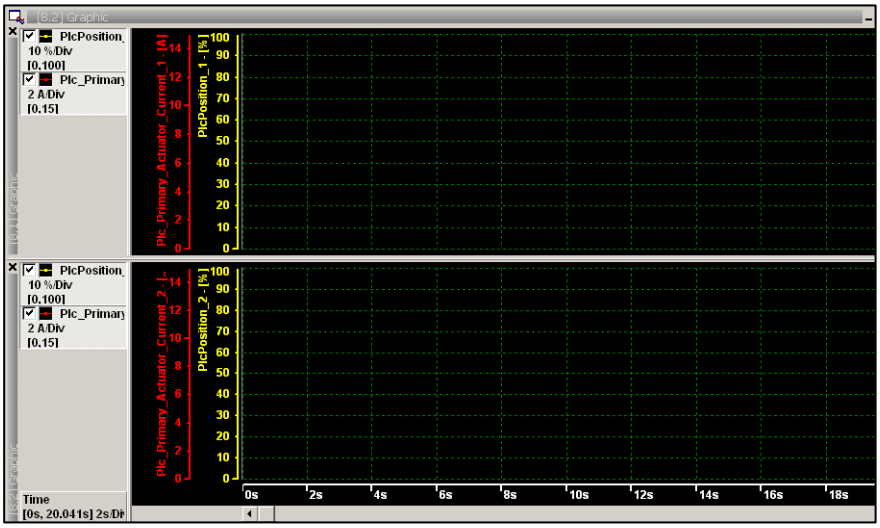


Fig. 3.51: finestra grafica per gli andamenti relativi ai due attuatori in prova

- Una finestra di scrittura ('*Write*') che consente di scambiare messaggi con il software *CANape* (fig. 3.52).

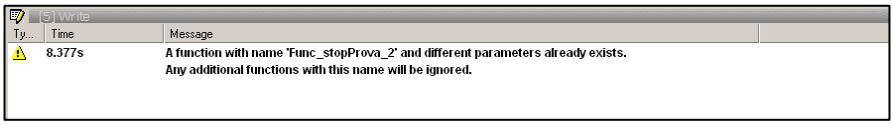


Fig. 3.52: finestra di scambio messaggi con CANape

Siccome la maggior parte delle attività di configurazione presentate nei capitoli successivi sono state eseguite su una centralina di controllo alla volta, si è ritenuto opportuno implementare un secondo progetto *CANape* predisposto per la comunicazione con un singolo dispositivo elettronico; questa azione è stata utile perché ha permesso di seguire il comportamento del *park-lock* nelle prove di calibrazione mediante un'interfaccia agile, priva soprattutto delle finestre relative al funzionamento di una seconda ECU. Il progetto ausiliario è stato realizzato partendo dal database originale delle centraline, mantenendo i nomi dei segnali inalterati (senza aggiunta di pedice numerico) e applicando le correzioni mostrate nei paragrafi precedenti; il *CANcase* è stato impostato per interfacciarsi con il dispositivo fisico collegato alla porta CAN1, a cui andranno connesse una alla volta le due centraline per il tempo necessario.

La figura seguente mostra, dunque, l'interfaccia che è stata impiegata per seguire le prove di calibrazione delle due ECU.

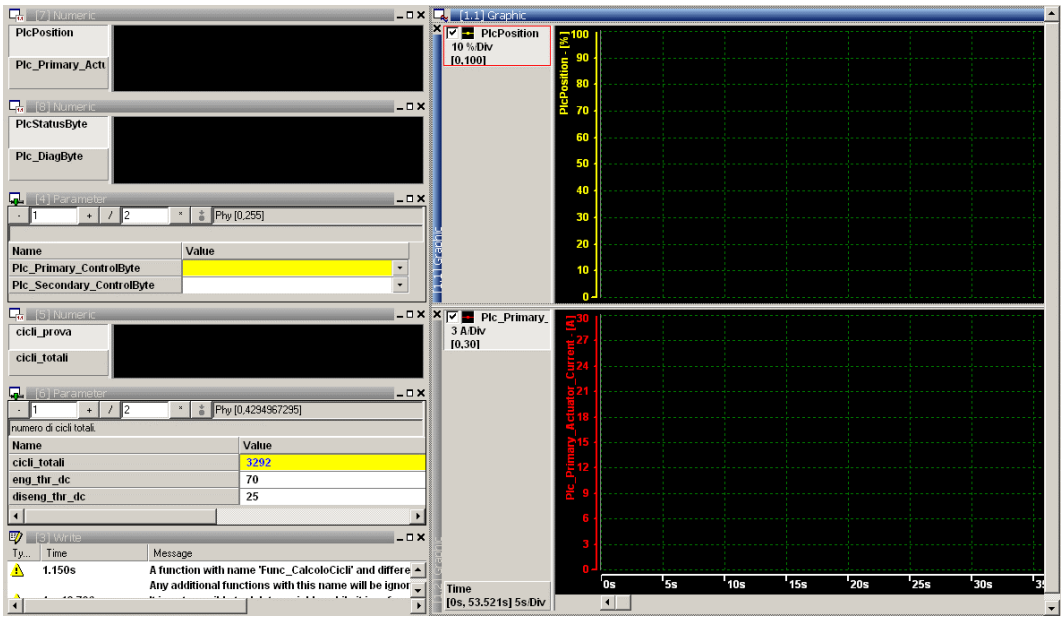


Fig. 3.53: interfaccia per prove su centraline singole

La configurazione di progetto di fig. 3.53 contiene tutte le informazioni che venivano già mostrate nell'interfaccia di '*DoubleECU_project*' declinate, però, per una singola centralina: dalla finestra grafica per gli andamenti di posizione e corrente a quella numerica per i cicli di prova e globali. A questi elementi già visti si aggiunge la seconda finestra numerica in alto a sinistra, la quale riporta lo stato dell'applicazione e lo stato di errore del sistema controllato.

CAPITOLO 4

Il controllo del moto del park-lock nelle prove di fatica

Il seguente capitolo inizierà con una breve panoramica sui motori elettrici a spazzole e, in particolare, sulle modalità tipiche di pilotaggio. Di seguito, si procederà alla presentazione delle leggi di spostamento da imitare al fine di realizzare delle prove di fatica adeguate e alla descrizione delle loro caratteristiche interessanti. Quindi, verranno presentati i due algoritmi di controllo considerati: prima quello originale (*closed loop*) e poi quello modificato (open *loop*); rispetto a quest'ultimo, sarà descritto minuziosamente il processo di calibrazione dei parametri caratteristici e verranno mostrati i relativi risultati. In ultimo, i due algoritmi di controllo saranno confrontati e si giustificherà la decisione di implementare il secondo sui *driver* di gestione del *park-lock*.

4.1 Il controllo dei motori elettrici a spazzole

Il *park-lock Oerlikon Graziano*, come precedentemente illustrato, è azionato mediante un piccolo motore elettrico a spazzole alimentato in corrente continua. Come la maggior parte dei motori elettrici DC, è caratterizzato da una carcassa esterna (lo statore) dotata di magneti permanenti e da un elemento rotante interno (il rotore); quest'ultimo è solcato da cave che costituiscono la sede di un insieme di avvolgimenti elettrici (detti di armatura). La figura successiva mostra il rotore e lo statore di un piccolo motore elettrico a spazzole disponibile in commercio.

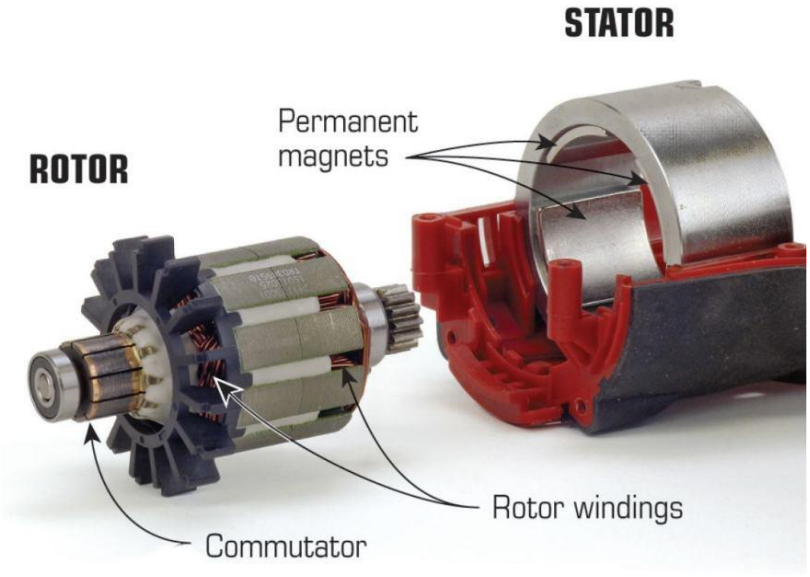


Fig. 4.1: struttura di un motorino brushed commerciale

Per quanto riguarda il funzionamento, gli avvolgimenti vengono alimentati in successione mediante il contatto tra una coppia di spazzole striscianti solidali allo statore e un collettore a lamelle posto sul rotore. Quando una bobina viene alimentata, la corrente che la percorre determina un campo magnetico di orientamento pressoché ortogonale rispetto al piano della stessa; l'interazione tra il suddetto campo e quello generato dai magneti permanenti dello statore produce una coppia meccanica (data dal disallineamento tra i campi) sull'albero rotorico che può mettere in rotazione un sistema meccanico ad esso accoppiato: in questo modo il motore elettrico trasforma la potenza elettrica (input) in potenza meccanica (output). Nel corso della rotazione, il sistema costituito da spazzole e collettore commuta la tensione di alimentazione degli avvolgimenti di armatura, impedendo il raggiungimento dell'allineamento tra i campi magnetici e assicurando, quindi, la continuità nell'erogazione della coppia motrice [34].

Per quanto riguarda il pilotaggio dei motori *brushed*, la semplicità di controllo costituisce uno dei loro maggiori punti di forza e uno dei motivi principali per cui, per certe applicazioni, vengano preferiti ad altre tipologie di motori elettrici. Infatti, mentre il *driver* di un motore *brushless* (rimanendo nell'ambito DC) deve essere fornito, sia a livello hardware che software, di sistemi adatti ad alimentare gli avvolgimenti nell'istante corretto e con corrente di forma adeguata, quello di un

motore a spazzole può realizzare il controllo gestendo un unico e semplice circuito elettronico: il ponte-H (o *H-Bridge*). Come evidenziato nel secondo capitolo, ogni modulo di valutazione (*DRV8701EVM*) è dotato di un *driver* (*DRV8701*) adatto a comandare un *H-Bridge* simile a quello di fig. 4.2.

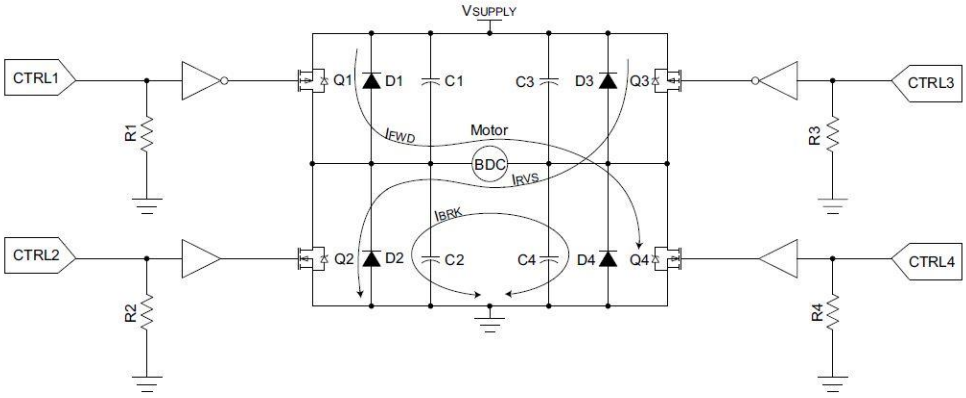


Fig. 4.2: esempio di H-Bridge per driver di motori a spazzole

Le fasi del controllo di un ponte H sono già state descritte nel capitolo dedicato alla centralina e ai sorgenti. In breve, il microcontrollore della scheda esegue un certo algoritmo che porta alla definizione di un segnale di riferimento per il *driver*. Il dispositivo elettronico converte il riferimento in un treno di impulsi PWM con *dc* ad esso proporzionale e che comanda l'attivazione di uno o più MOSFET; le proprietà induttive e capacitive del circuito elettrico fanno percepire agli avvolgimenti una tensione di comando nel range 0-12V direttamente proporzionale al *dc*. Naturalmente, se il motore è parte dell'attuazione di un servomeccanismo di cui si desidera controllare una certa grandezza (ad esempio posizione o velocità), il *duty cycle* di comando deve essere valutato secondo una logica opportuna, implementata da un software realizzato ad hoc per l'applicazione.

Di solito, i motori elettrici a spazzole sono controllati in velocità [35] e le applicazioni in cui ne viene richiesto il controllo in posizione (come nel caso del *park-lock*) sono piuttosto rare: in questi casi si preferisce utilizzare altre tipologie di motore come, ad esempio, lo *stepper*. L'algoritmo di regolazione più diffuso è basato su un unico *loop* PID, il quale riceve in ingresso l'errore sulla grandezza soggetta a controllo e restituisce in uscita un valore di riferimento proporzionale

alla tensione di comando; questo tipo di controllore garantisce ottimi risultati (dal punto di vista della precisione, della stabilità, del tempo di risposta,…) ed è relativamente semplice da calibrare. La struttura e le proprietà del PID non sono descritte in questo capitolo introduttivo in quanto si ritiene più efficace presentarle in seguito durante la trattazione degli algoritmi di controllo.

Nei paragrafi successivi verranno descritti due algoritmi di controllo, di cui è stata analizzata o realizzata l'implementazione nei sorgenti dei *driver*:

- Quello originale, progettato dall'azienda che ha fornito la centralina.
- Quello da me inserito nei codici, impiegato per realizzare tutte le prove di fatica (leva e molla).

Le trattazioni sopra riportate saranno anticipate da un breve resoconto delle caratteristiche della legge di posizione che ho dovuto riprodurre; infine, il capitolo sarà concluso da un paragrafo di considerazioni che mira a confrontare le due strategie considerate e a giustificare la scelta presa nella realizzazione del controllo.

4.2 Le leggi di posizione di riferimento

Una volta terminate le attività di taratura (esposte nel capitolo delle attività complementari) sul sistema costituito da centralina e progetto *CANape*, è stato possibile procedere verso l'adempimento della richiesta più importante che mi è stata fatta: fare sì che la ECU di controllo realizzasse delle leggi di posizione il più simili possibile a quelle a quelle presentate dalla ECU di riferimento, per entrambe le fasi di attuazione. Le figure successive mostrano nel dettaglio gli andamenti di posizione da riprodurre, così da poter successivamente esprimere delle considerazioni al riguardo.

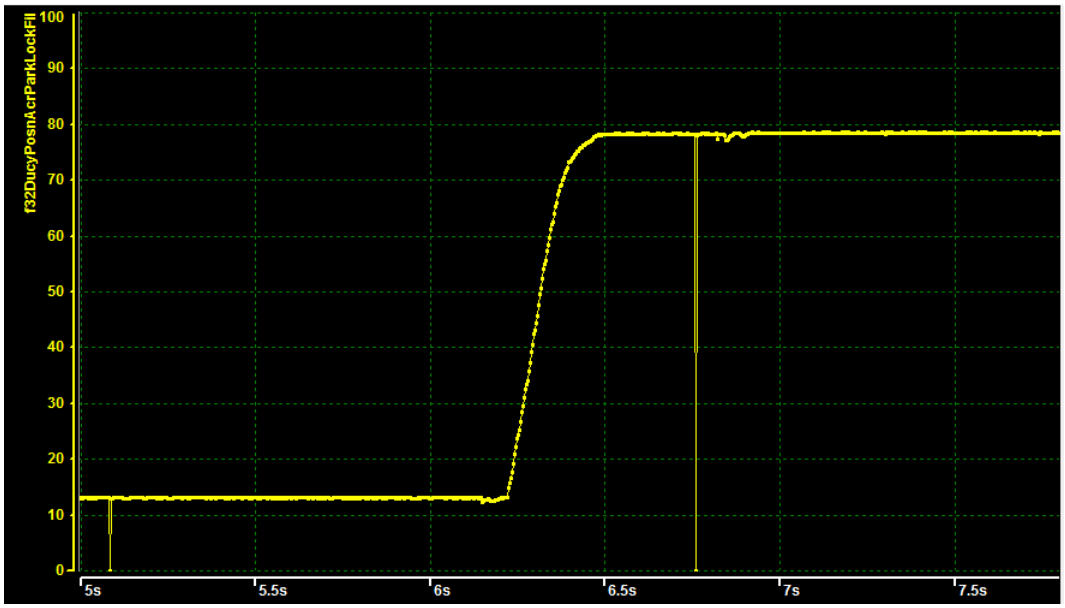


Fig. 4.3: legge di riferimento per la corsa di innesto

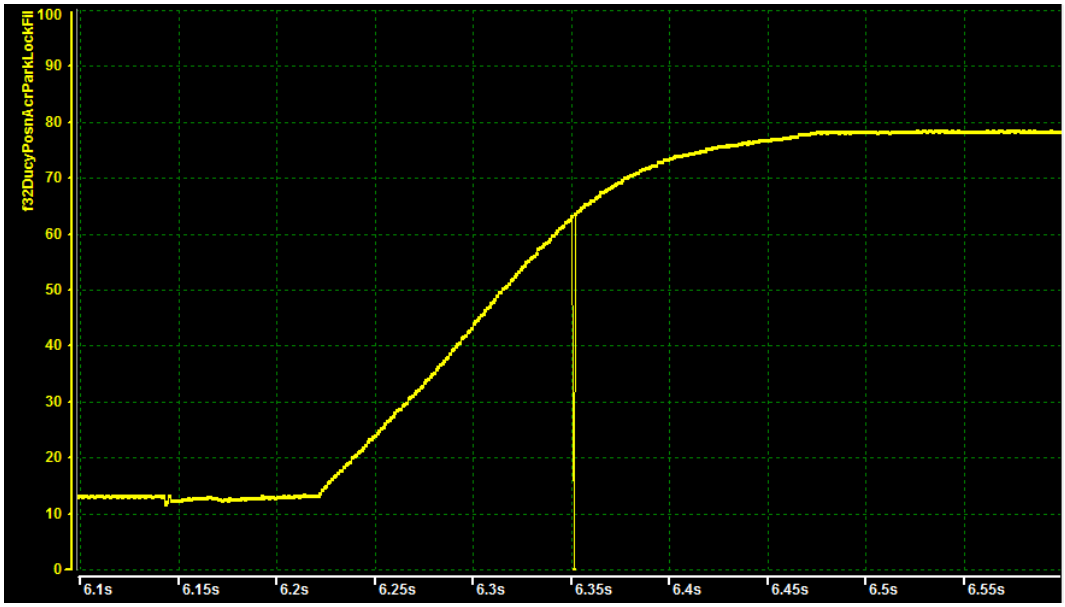


Fig. 4.4: legge di riferimento per la corsa di innesto, ingrandimento

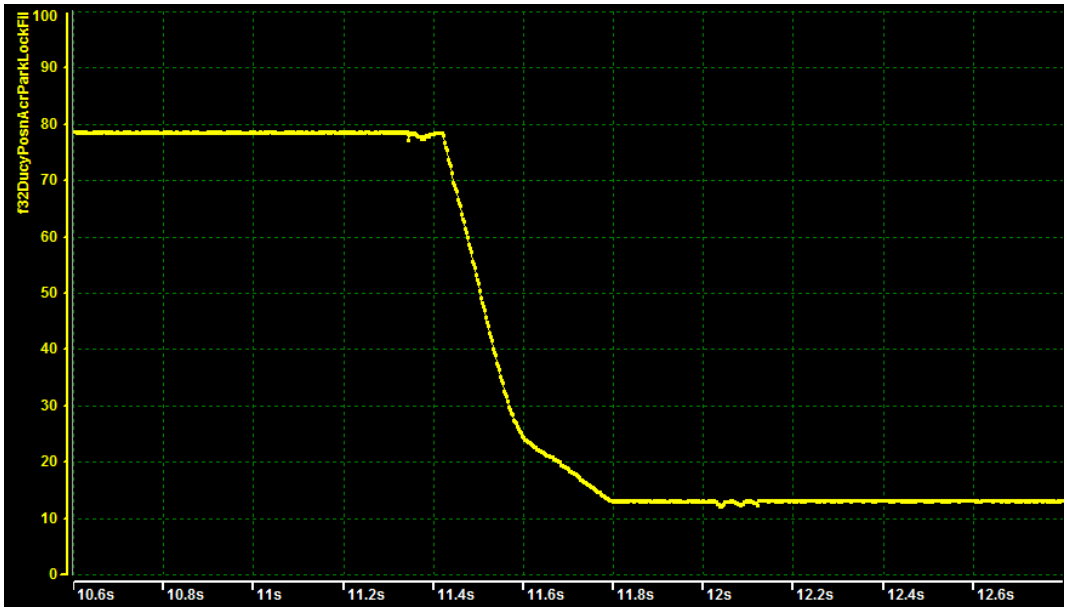


Fig. 4.5: legge di riferimento per la corsa di disinnesto

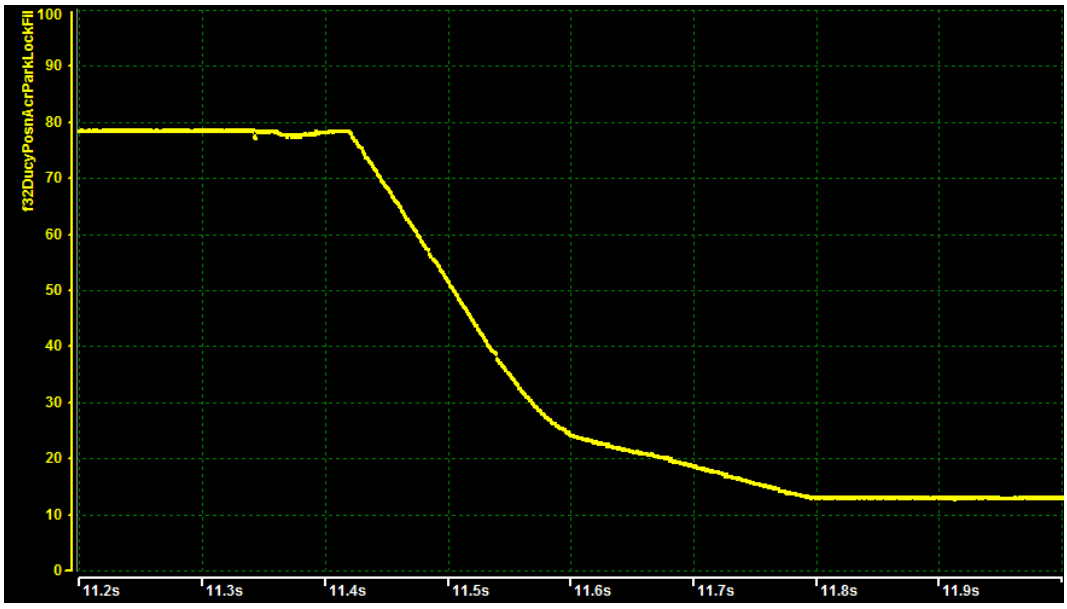


Fig. 4.6: legge di riferimento per la corsa di disinnesto, ingrandimento

Innanzitutto, è necessario individuare gli aspetti caratterizzanti della legge del moto per ciascuna fase:

- **<u>Moto di innesto</u>** (figg. 4.3 e 4.4)

  Il movimento di *engage* inizia con velocità (corrispondente alla pendenza della curva) molto elevata e circa costante; quando il *dc* di posizione

raggiunge circa il 60% il motore rallenta, raggiungendo nuovamente una velocità pressoché costante fino al raggiungimento della posizione di finecorsa. L'andamento della velocità (e quindi della posizione) si possono facilmente giustificare considerando che, in generale, si desidera che l'innesto del *park-lock* sia il più rapido possibile e che, allo stesso tempo, la leva porta-magnete non impatti contro il finecorsa con velocità eccessiva; un urto troppo violento potrebbe, infatti, determinare il danneggiamento di uno o di entrambi i componenti coinvolti, il quale potrebbero rendere inutilizzabile il dispositivo di sicurezza.

- **Moto di disinnesto** (figg. 4.5 e 4.6).

Anche il movimento di *disengage* parte rapidamente e con pendenza costante, per poi essere frenato quando il *dc* raggiunge circa il 25% e raggiungere il finecorsa di disinnesto a velocità bassa e costante. Le motivazioni sono le stesse di prima: minore è la velocità con cui, in questo caso, l'arpione raggiunge la condizione di finecorsa (data dal contatto con la camma) e minore è il rischio di danni al dispositivo.

Si ritiene interessante sviluppare alcune considerazioni sulla durata delle due fasi di attuazione rispetto al caso di riferimento. Innanzitutto, si possono distinguere almeno due differenti intervalli di tempo che possono essere utilizzati per identificare la durata di una fase:

- Quello che intercorre tra l'inizio del moto e il raggiungimento della posizione di finecorsa complementare.
- Quello che sussiste tra l'istante in cui viene prodotto il comando di innesto/disinnesto e quello in cui si perviene alla posizione desiderata.

Il primo tipo di durata è utile per comprendere la sola risposta cinematica del sistema e si può determinare esaminando gli andamenti di posizione delle figure 4.4 e 4.6: per l'innesto risulta essere di circa 280ms, per il disinnesto di circa 380ms. Invece, il secondo tipo di durata è importante per comprendere l'entità delle resistenze (attriti, rigidezze, inerzie) che si oppongono all'avvio del moto del *park-lock* e, in particolare, per definire quanto tempo impiega l'algoritmo utilizzato per

generare (indirettamente) la coppia di spunto necessaria per muovere il dispositivo; questo intervallo di tempo si può apprezzare esaminando in parallelo gli andamenti della posizione e della corrente richiamata dagli avvolgimenti (figg. 4.7 e 4.8).

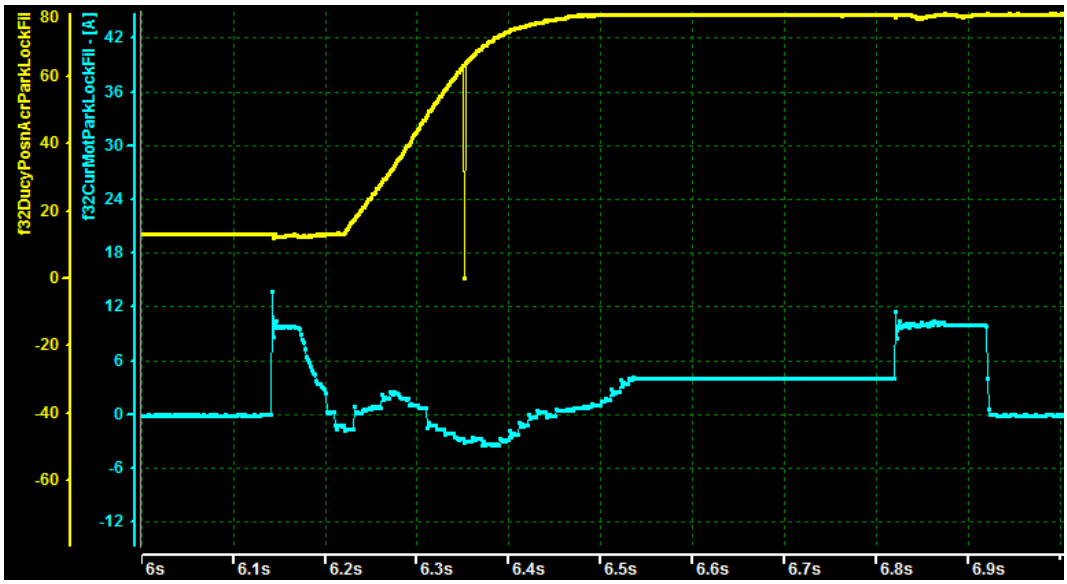


Fig. 4.7: andamenti di posizione e di corrente di riferimento per engage

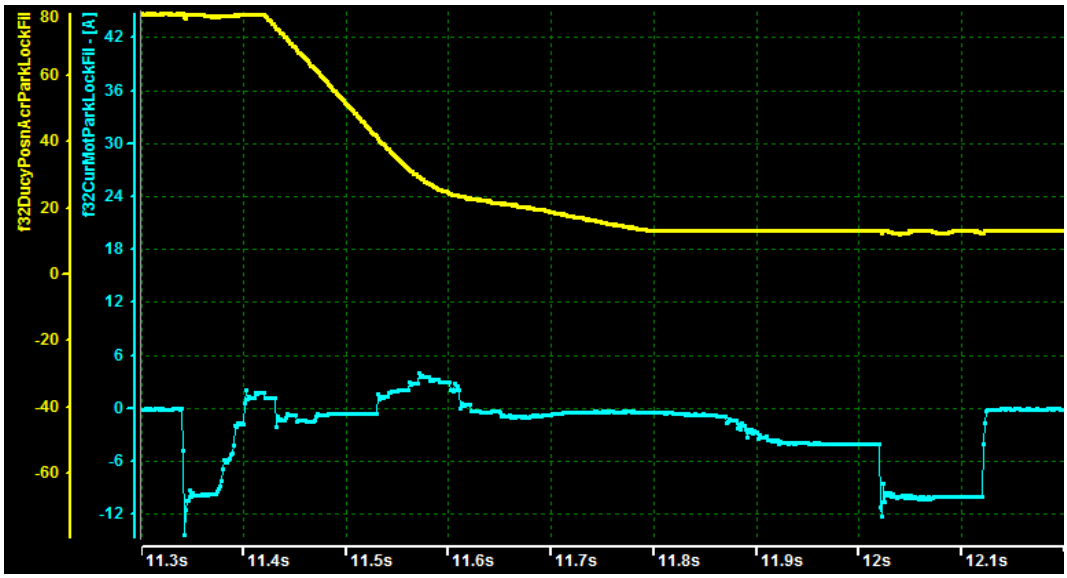


Fig. 4.8: andamenti di posizione e di corrente di riferimento per disengage

In riferimento alle due figure precedenti, è opportuno osservare che la visualizzazione della corrente (mostrata in azzurro) consentita dal sistema di riferimento è sensibile al suo verso di percorrenza: se la corrente produce coppia motrice nel verso di *engage* allora è positiva, altrimenti è negativa. In entrambi i

casi, al ricevimento del comando di attuazione, l'assorbimento della corrente inizia con un gradino di modulo 10A per poi mantenersi costante per qualche decimo di secondo e ridursi (sempre in modulo) quando l'alberino di ingresso *park-lock* inizia a muoversi. L'istante in cui il motore inizia a richiamare corrente (perché viene applicata un tensione di comando non nulla) corrisponde approssimativamente a quello in cui il *driver* riceve il comando di innesto/disinnesto prodotto dall'utente. Superati gli attriti e le inerzie, la coppia sull'alberino permette di avviare il moto del sistema: l'algoritmo di controllo determina un andamento del *dc* di comando e, quindi, della corrente assorbita e della coppia motrice tale da movimentare il *park-lock* secondo la legge di posizione desiderata.

Tornando ai grafici di corrente, si evidenzia che il controllo attuato dalla centralina di riferimento prevede che, in una stessa fase, la corrente possa attraversare gli avvolgimenti in entrambi i versi, a seconda che si desideri produrre una coppia motrice oppure frenante rispetto al movimento previsto dalla fase. Analizzando gli andamenti delle figure 4.7 e 4.8, si nota che, dopo il gradino di corrente di partenza a 10A, la corrente diminuisce in modulo per poi cambiare segno (quindi frenare) per poi, poco dopo, andare a zero: a ciò è dovuto il repentino rallentamento osservato per *duty cycle* pari a 60% per *engage* e 25% per *disengage*. Si nota ancora che subito dopo la soglia di innesto/disinnesto la corrente è mantenuta costante prima a 4A e poi a 10A; la prima azione è volta, teoricamente, a limitare a 4A la corrente nella fase di urto, mentre lo scopo della seconda verrà spiegato nel capitolo dedicato alle attività di configurazione complementari.

Ritornando al discorso precedente, i tempi di aggancio e sgancio compresi i ritardi provocati da attriti ed inerzie corrispondono a circa 360ms per *engage* e circa 470ms per *disengage*. La tabella seguente riassume le tempistiche caratteristiche della legge di riferimento da copiare.

	Tempo di attuazione netto [ms]	**Tempo di attuazione + ritardo di spunto [ms]**
Innesto (*engage*)	280	360
Disinnesto (*disengage*)	380	470

Tab. 4.1: tempi di attuazione caratteristici della legge di riferimento

Si ricorda ancora che lo svolgimento delle attività di tesi ha comportato la riprogrammazione parziale delle schede di due centraline (ciascuna dotata di un *core* e due *driver*). Per i quattro *driver* è stato elaborato un unico progetto sorgente, adatto a generare un singolo *firmware* con caratteristiche adeguate per l'applicazione (il controllo di un *park-lock* non ridondato) e caricabile su tutte le schede. L'eseguibile in questione si ottiene compilando il progetto sorgente nella configurazione MAIN, mentre l'altra configurazione (AUX) è rimasta praticamente intatta ed inutilizzata. Si sottolinea questo per far comprendere a priori il motivo per cui, nella successiva descrizione delle tattiche di controllo, si farà riferimento solo alla configurazione MAIN e non a quella AUX.

4.3 L'algoritmo di controllo originale: PI di posizione, PID di corrente e integratore puro

Si ritiene opportuno, a questo punto, procedere con la descrizione dell'algoritmo di controllo originale del *park-lock*, ossia quello implementato nei sorgenti di partenza forniti con la centralina. La comprensione accurata della logica di controllo originale costituisce un punto molto importante per il lavoro di tesi e, in generale, per l'arricchimento della mia esperienza accademica e aziendale per almeno due motivi:

(1) Perché rappresenta una struttura di controllo tipica nell'ambito della regolazione dei motori elettrici *brushed* DC; dunque, si ritiene che presenti un certo valore dal punto di vista didattico, soprattutto rispetto alla mia scelta di seguire l'orientamento di automazione.

(2) Perché tale architettura di controllo, benché non sia stata utilizzata per soddisfare le richieste ricevute inerenti la configurazione del sistema per le prove di fatica, potrebbe essere molto utile in applicazioni (o addirittura sistemi) differenti; ad esempio, potrebbe in futuro essere impiegata per controllare pompe, valvole o altri dispositivi *park-lock* azionati in DC.

Passando oltre, la figura 4.9 schematizza con un diagramma a blocchi l'algoritmo seguito da una scheda caratterizzata con un *firmware* MAIN originale; in seguito, la figura 4.10 non riporta solo la logica di controllo ma presenta anche tutta la sequenza di fasi che parte dalla generazione di un comando ENG MAIN o DISENG MAIN e termina con la produzione della legge di rotazione desiderata.

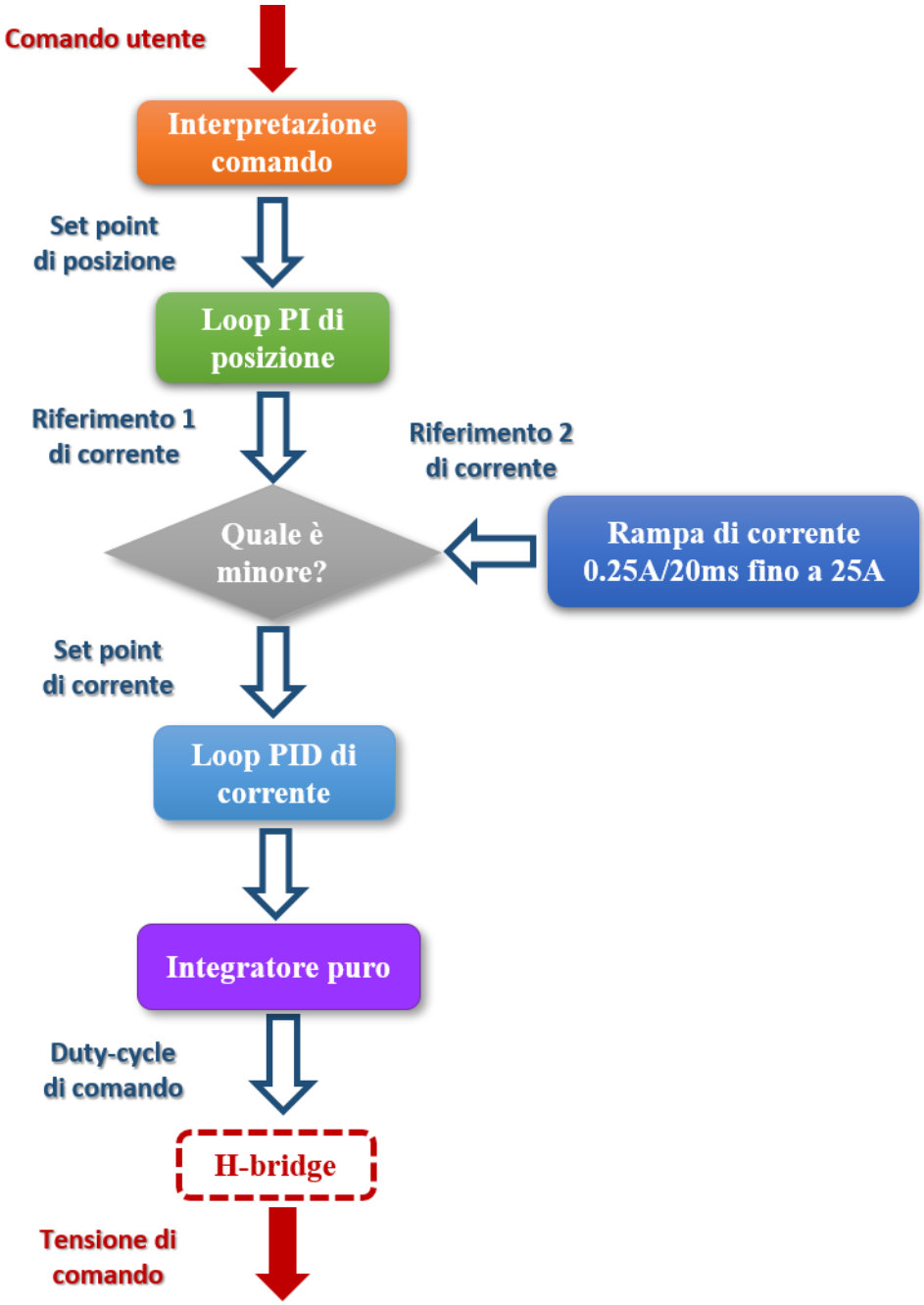


Fig. 4.9: diagramma contratto dell'algoritmo di controllo originale

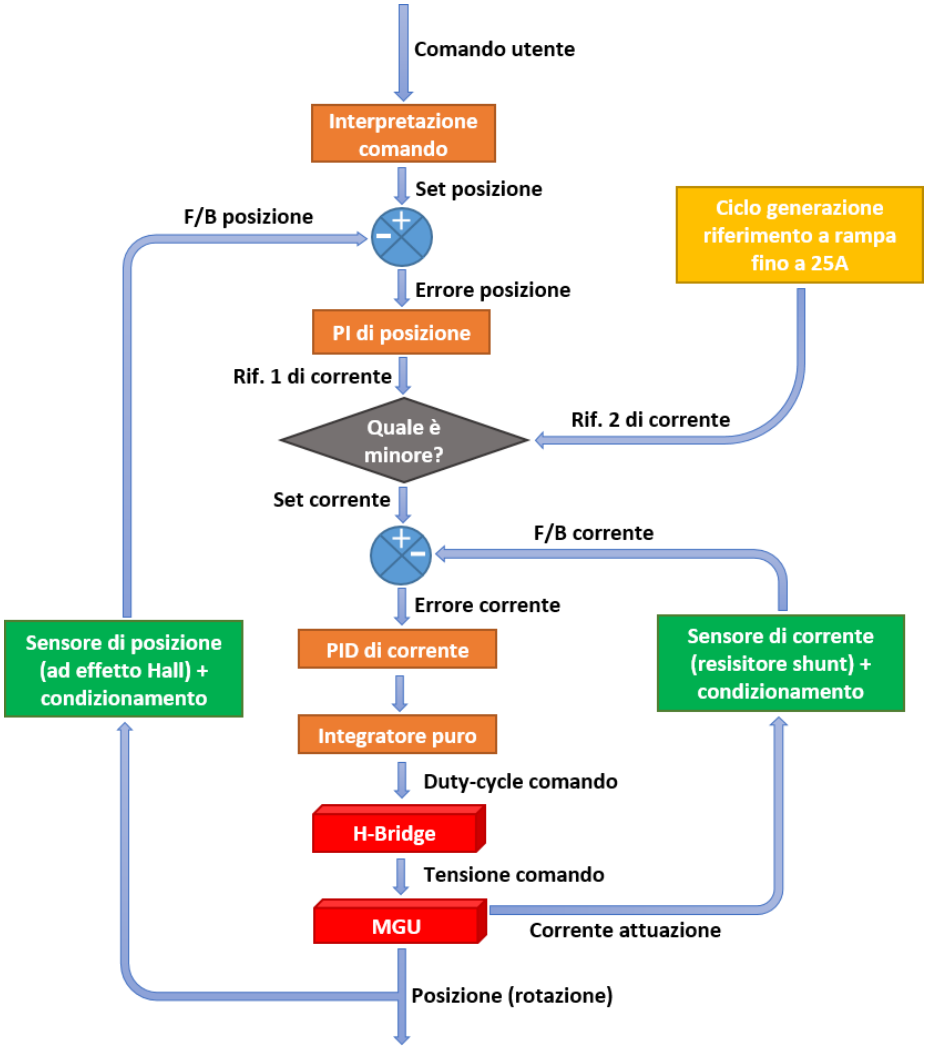


Fig. 4.10: diagramma esteso dell'algoritmo di controllo originale

Andando per ordine rispetto all'immagine precedente, quando uno dei due pulsanti di ENG MAIN o DISENG MAIN viene premuto è prodotto un comando che prima raggiunge il *core* e poi ad una o entrambe le schede *driver* (dipende dal sorgente e dal comando) mediante I2C. Quindi, il comando viene interpretato dal *driver* dentro *app_fsm.c* e ciò conduce all'esecuzione di una serie di istruzioni in sequenza; la più importante di queste definisce la soglia di posizione di innesto/disinnesto: si tratta del valore di *dc* (nella scala utilizzata dal *driver* per manipolare la posizione) che deve essere rilevato prima che scatti il *timeout* che porta poi all'interruzione della fase di controllo motore.

La soglia di posizione diventa quindi il *set point* (ossia il valore da raggiungere) che viene inseguito attraverso un primo anello (*loop*) di controllo di posizione operante con logica PI. L'uscita del controllo PI è un primo riferimento di corrente (nella scala di corrente del *driver*) che viene confrontato con un secondo valore di riferimento dato da un andamento che parte a rampa con pendenza 10A/s e poi finisce costante (come RELOAD) a 25A: il minore dei due numeri diventa il set point di un secondo anello di controllo, interno al primo, questa volta di corrente. Il secondo *loop* di controllo agisce con logica PID e genera un valore che viene integrato nel tempo per produrre, finalmente, il *duty cycle* di comando (0-100%), il quale definisce la tensione di comando (0-12V) del motore.

La tensione di alimentazione, insieme alla forza controelettromotrice data dalla rotazione e alle cadute di tensione dovute ad effetti resistivi, induttivi e capacitivi, determina la corrente richiamata dagli avvolgimenti della MGU. A sua volta la corrente produce una coppia motrice (di origine magnetica) agente sul rotore e l'equilibrio meccanico del dispositivo determina la legge di rotazione dell'alberino e, quindi, della leva. Il sensore Hall produce un PWM con *dc* proporzionale alla rotazione, che viene convertito (e condizionato) in tensione analogica dalla scheda filata e quindi trasferito al convertitore A/D del *driver*: il valore convertito e mediato (*POS_AVG_VAL*) costituisce il feedback del *loop* di posizione. Il sensore di corrente produce un segnale analogico di tensione che, una volta condizionato, è passato al convertitore A/D e il risultato della conversione mediato (SO_MCU_AVG) diventa il feedback dell'anello di corrente.

Il sistema costituito da centralina e *park-lock* risulta dunque essere un tipico servosistema elettromeccanico comandato elettronicamente, che dal punto di vista fisico si può schematizzare con i seguenti blocchi (fig. 4.11).

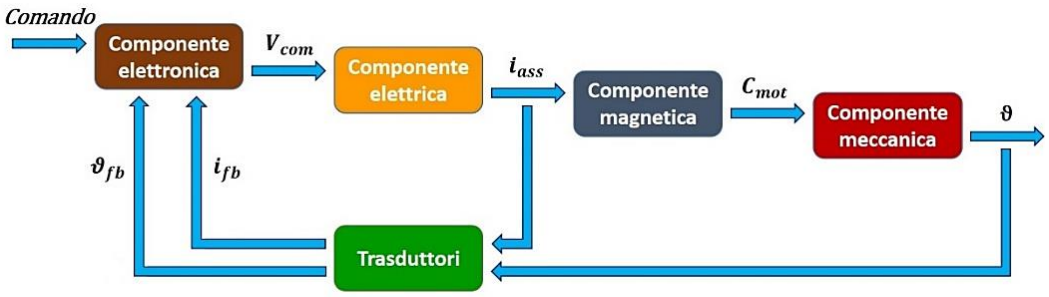


Fig. 4.11: schema del servosistema elettromeccanico a controllo elettronico digitale

Per i motivi di natura didattica precedentemente evidenziati, si ritiene opportuno offrire qualche delucidazione riguardo ai due *loop* di controllo che caratterizzano la logica originale. Si anticipa che le osservazioni successive faranno riferimento a sistemi operanti in modo continuo nel tempo, mentre la centralina, essendo un dispositivo digitale, non può che agire in modo tempo-discreto; ciononostante, i ragionamenti presentati nelle righe successive si possono generalmente considerare validi anche per i sistemi a tempo discreto e se ciò non fosse vero verrà naturalmente specificato.

L'anello chiuso di posizione si basa su una sequenza di istruzioni che implementano un controllore di tipo PI, ossia Proporzionale-Integrativo, la cui forma generica viene schematizzata a blocchi nella figura successiva [31].

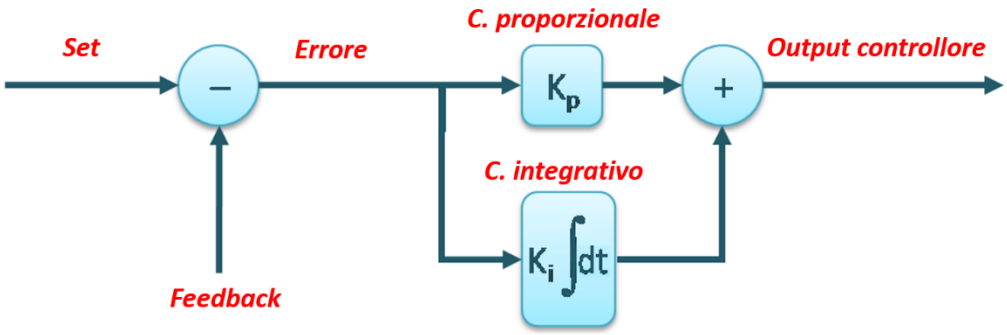


Fig. 4.12: schema a blocchi di un controllore PI

Questo tipo di controllo prevede, innanzitutto, il calcolo (nella scala di lavoro utilizzata) dell'errore che sussiste tra un valore di set e uno di feedback, associati alla medesima grandezza fisica che si desidera controllare. Il set (o *set point*) corrisponde al valore che si vorrebbe presentasse, in un certo istante, il parametro

controllato; invece, il feedback (o retroazione) è la misura del valore reale che la grandezza presenta nel medesimo istante. Questo errore viene dunque utilizzato per calcolare due contributi:

- Termine proporzionale (P): si ottiene moltiplicato semplicemente l'errore per una costante (K_p).
- Termine integrativo (I): è calcolato valutando l'integrale dell'errore a partire dall'istante di avvio del controllo moltiplicandone il valore per una costante (K_i).

Quindi, la somma dei due contributi definisce un valore di riferimento che costituisce l'output del controllore; esso può essere impiegato per generare un segnale di comando (ad esempio una tensione) ad esso spesso proporzionale oppure, come in questo caso, può diventare il *set point* di un secondo anello di controllo. Si evidenzia che i parametri che caratterizzano il comportamento statico e dinamico di un controllore PI sono le costanti proporzionale ed integrativa; da esse dipende il modo in cui il sistema agisce per compensare l'errore tra set e F/B, in particolare perché definiscono l'importanza assoluta e relativa dei due termini ad esse associati.

L'anello di controllo della corrente si serve di un nucleo di controllo basato sulla logica PID, che sta per Proporzionale-Integrativo-Derivativo; l'azione di un controllore PID non è molto dissimile da quella di un PI: si differenzia per l'aggiunta di un terzo contributo, quello derivativo, calcolato derivando l'errore nel tempo e moltiplicando il risultato per la costante derivativa K_d (fig. 4.13) [31].

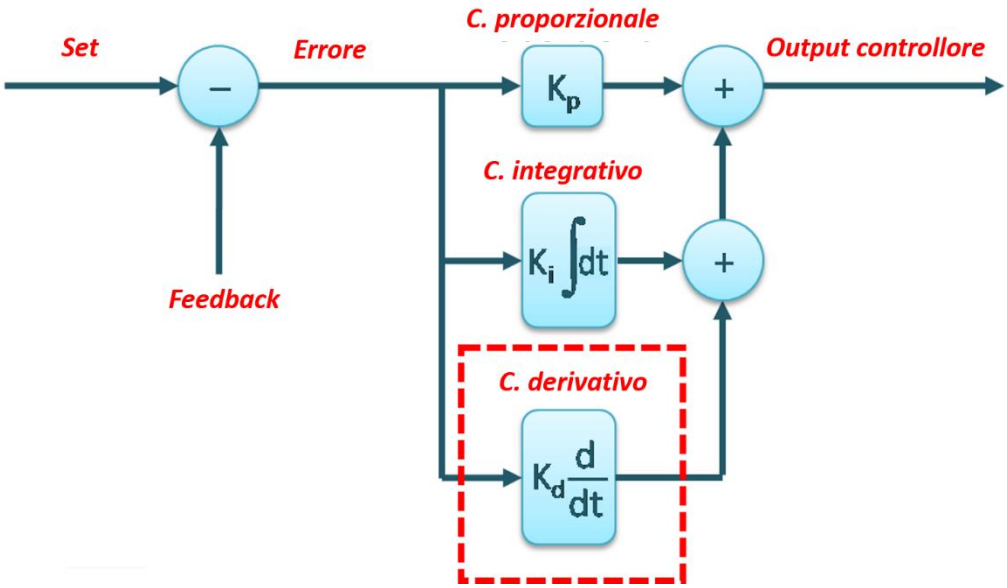


Fig. 4.13: schema a blocchi di un controllore PID

In realtà, il PID utilizzato per il controllo della corrente è 'fittizio'; la struttura implementata è, infatti, quella di un PID e prevede il calcolo dei tre termini, però la configurazione originale pone a zero la costante derivativa e dunque anche il termine relativo, trasformando così il PID in un PI. Per comprendere questa decisione conviene ricordare i principali effetti dei contributi di un PID [31]:

- *Proporzionale*: l'aumento del K_p porta al miglioramento della reattività del sistema controllato e alla riduzione dell'errore statico (in caso di assenza dell'azione integrativa); di contro riduce il margine di stabilità.
- *Integrativo*: azzera l'errore a regime, tanto più velocemente quanto maggiore è la costante K_i; di contro l'incremento del parametro determina riduzione della reattività e avvicinamento all'instabilità.
- *Derivativo*: in riferimento ad un comando a gradino, riduce la sovraelongazione (*overshoot*) e la durata del transitorio (*settling time*), inoltre aumenta la reattività e la stabilità; il rovescio della medaglia è che aumenta il rumore ad alte frequenze.

I controllori PID rappresentano circa il 95% dei controllori continui utilizzati in ambito industriale ed automotive [31]; tra di questi, la grande maggioranza

implementa in realtà la logica PI, senza contributo derivativo, per determinare il riferimento di uscita. Questo si collega al fatto che, quando si controlla un servosistema, si desidera che l'ampiezza della risposta decada il più rapidamente possibile quando il comando supera una certa frequenza (quella della banda passante). Le variazioni degli input caratterizzate da alta o altissima frequenza sono, infatti, spesso create da disturbi elettrici, da vibrazioni, etc., e quindi non sono collegate ad un esigenza dell'utente e quindi si desidera che il sistema ne rimanga immune; la reattività ad alte frequenze pregiudica il raggiungimento di una configurazione stabile di funzionamento e conduce alla perenne oscillazione dell'uscita. È pertanto evidente il motivo per cui l'azione derivativa viene tipicamente cancellata, preferendo giocare con le costanti degli altri due termini per ottenere il comportamento (reattività, stabilità, cancellazione dell'errore statico) desiderato.

4.4 L'algoritmo di controllo modificato: open loop con interpolazione del comando dal feedback di posizione

Al fine di imitare la legge di posizione di riferimento, è stata presa la decisione di abbandonare la struttura di controllo originale, complessa e difficile da calibrare, per passare ad un'architettura differente e decisamente più semplice; è stato, infatti, implementato un sistema di controllo in anello aperto (*open loop* o OL) molto particolare e basato sull'interpolazione. Normalmente si parla di controllo in anello aperto quando l'uscita dell'unità di controllo è indipendente dalla variabile di processo controllata (fig. 4.14); un classico esempio di sistema controllato in OL è l'asciugatrice del bucato: essa agisce sul carico per un intervallo di tempo fissato a priori dall'utente, senza l'utilizzo di un feedback sull'effettivo grado di asciuttezza del bucato.

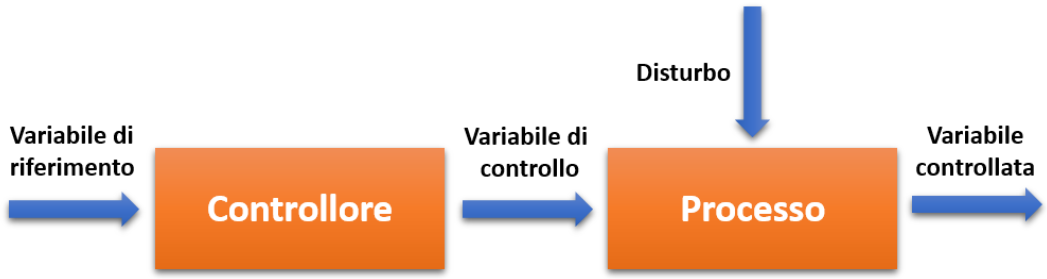


Fig. 4.14: struttura di un controllo OL

In riferimento all'applicazione analizzata, ossia l'attuazione di motori *brushed* in corrente continua, un controllo in OL tradizionale vorrebbe la definizione all'interno del software di una legge temporale per il riferimento di comando e l'esecuzione della stessa in modo svincolato dall'andamento reale della variabile di processo soggetta a controllo (la posizione angolare dell'alberino della MGU). Evidentemente, nel caso di una struttura di comando come quella descritta, la presenza di un trasduttore di posizione elaborante un feedback per l'unità di controllo sarebbe inutile, a meno di eventuali funzioni di sicurezza (ad esempio, monitorare che il *duty cycle* rimanga in un certo limite).

La nuova struttura di controllo non corrisponde a quella appena presentata ma si differenzia per il fatto che il valore di feedback della posizione viene utilizzato per calcolare il valore di riferimento per il *driver* del ponte H, ossia il *dc* del segnale PWM. L'algoritmo si può interpretare, in realtà, come una via di mezzo tra un *open loop* e un *Feed-Forward* (FF); questo secondo tipo di regolazione determina l'uscita del controllore sulla base della misura diretta di un disturbo agente sul processo controllato. Quindi, si è optato per l'identificazione della nuova strategia di controllo mediante la locuzione '*open loop*' non per l'assenza di rami di retroazione ma per il fatto che l'ingresso dell'algoritmo di controllo è costituito dal feedback stesso (fig. 4.15) e non dall'errore tra un valore di set e uno di feedback.

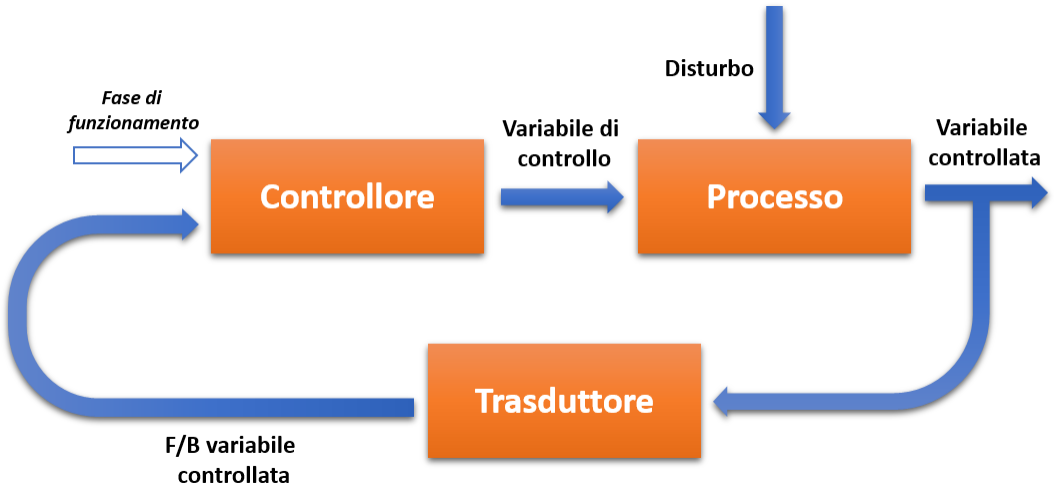


Fig. 4.15: struttura del controllo OL implementato

L'output del controllore è definito, come già anticipato, mediante un algoritmo imperniato sull'interpolazione. Dal punto di vista software, per ciascuna delle due fasi di attuazione (innesto e disinnesto) vengono definiti due vettori costanti: uno per i *dc* di posizione e l'altro per i *dc* di comando; essi contengono il medesimo numero di valori interi e le coppie di elementi accomunate dallo stesso indice di posizione all'interno dei vettori identificano una serie di punti di riferimento nel piano *dc* posizione – *dc* comando, i quali costituiscono la base per l'interpolazione attuata dal controllo.

In poche parole, ad ogni ciclo di esecuzione dell'algoritmo di controllo (implementato in *motor_control*) il F/B di posizione (*POS_AVG_VAL*) viene convertito nel *dc* corrispondente attraverso una macro, descritta in seguito nel capitolo dedicato alle attività complementari. Quindi, viene identificata la coppia di punti di riferimento adiacente tra le cui ascisse è compreso il *dc* di posizione attuale. Infine, mediante una semplice formula di interpolazione lineare tra i due punti individuati, viene calcolato il *dc* di comando dell'*H-Bridge* e quindi il valore della tensione di comando del motorino (fig. 4.16).

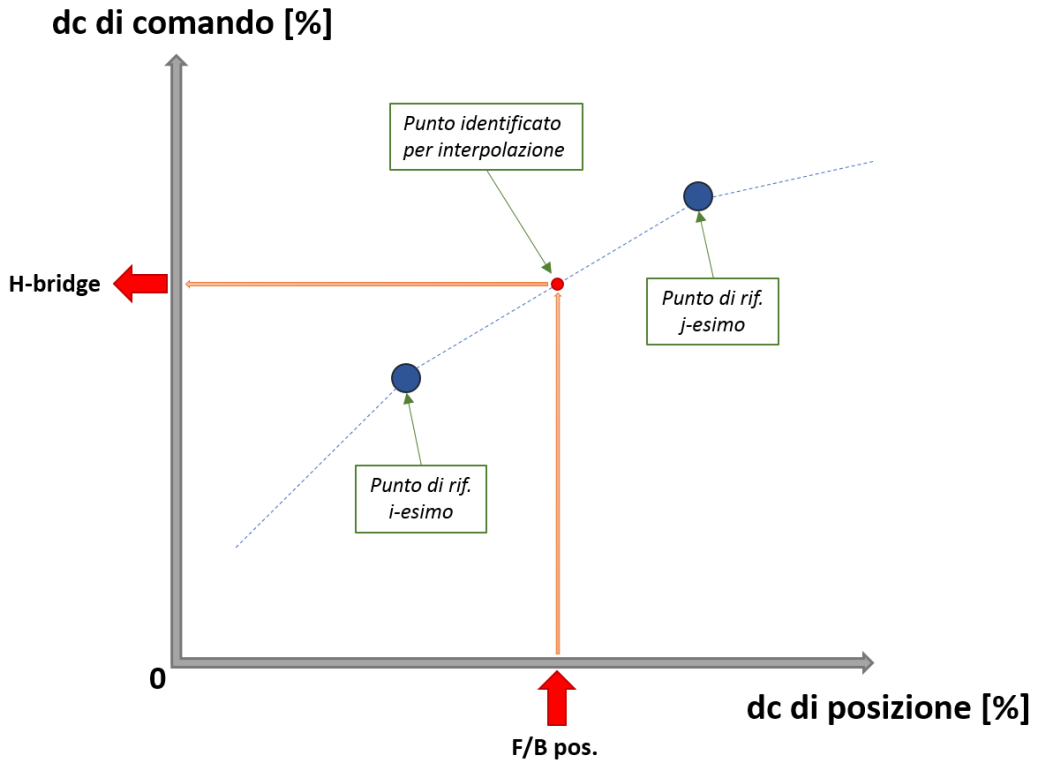


Fig. 4.16: rappresentazione grafica del processo di interpolazione lineare

È opportuno fare un paio di precisazioni sull'implementazione della procedura di interpolazione lineare.

1) Innanzitutto, non è stato possibile utilizzare lo stesso insieme di punti di riferimento per le fasi di *engage* e *disengage*, ossia prevedere per entrambe la stessa legge di comando in funzione della posizione. Infatti, per ciascuna delle due fasi è richiesta l'imitazione di una diversa legge di posizione e la coppia che il motore deve fornire (e quindi la corrente da esso assorbita) non dipende solo dalla posizione ma anche dalla velocità e da altri fattori (attrito, giochi, ecc.) inevitabilmente legati al verso di rotazione dell'attuatore e, quindi, alla specifica fase di funzionamento. Per tutti questi motivi è stato necessario definire una coppia di vettori di riferimento differenti per ciascuna fase.
2) In secondo luogo, i due algoritmi di esecuzione dell'interpolazione non sono esattamente identici. Infatti, nel caso dell'innesto, la ricerca della coppia di punti tra cui eseguire l'interpolazione parte dal primo elemento (indice i =

0) del vettore dei *dc* di posizione riferimento: se tale valore è maggiore di quello calcolato dal F/B allora si effettua l'interpolazione tra il punto i-esimo e quello (i-1)-esimo, altrimenti l'indice è incrementato di 1 e il calcolo viene ripetuto (mediante un ciclo for). Invece, nel caso del disinnesto, la ricerca parte dal penultimo elemento (indice j = end - 1) del vettore dei *dc* di posizione di riferimento: se è minore di quello di F/B viene realizzata l'interpolazione tra il punto (j+1)-esimo e quello j-esimo, altrimenti si verifica quello disposto nella posizione precedente.

Le figure successive mostrano, prima sinteticamente (fig. 4.17) e poi in forma estesa (fig. 4.18), i diagrammi a blocchi che mettono in luce il funzionamento di un *driver* su cui è stato caricato un *firmware* MAIN implementante il nuovo algoritmo.

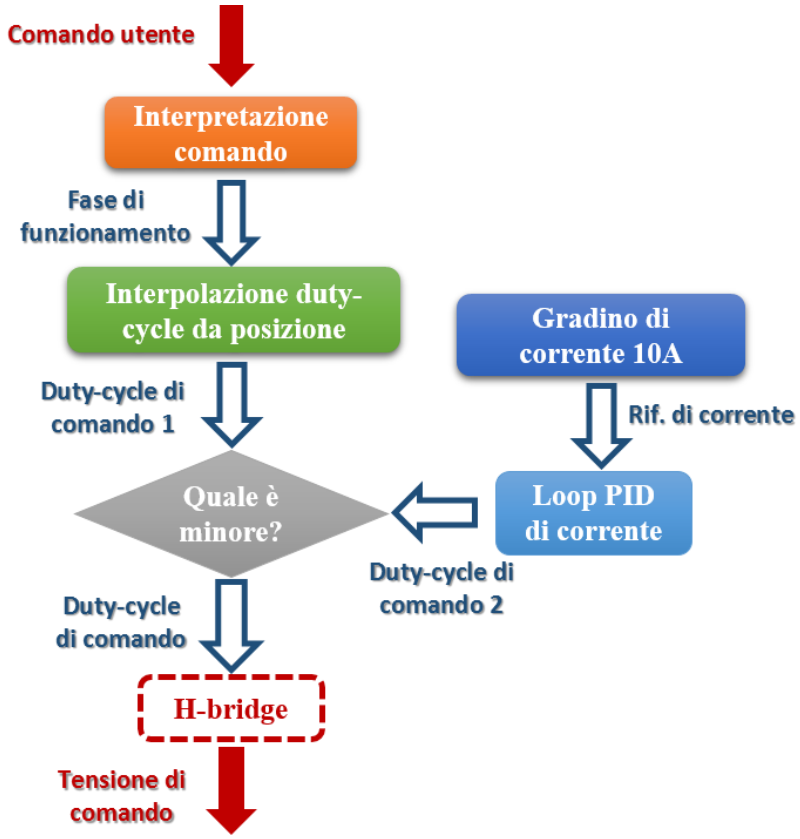


Fig. 4.17: diagramma contratto dell'algoritmo di controllo modificato

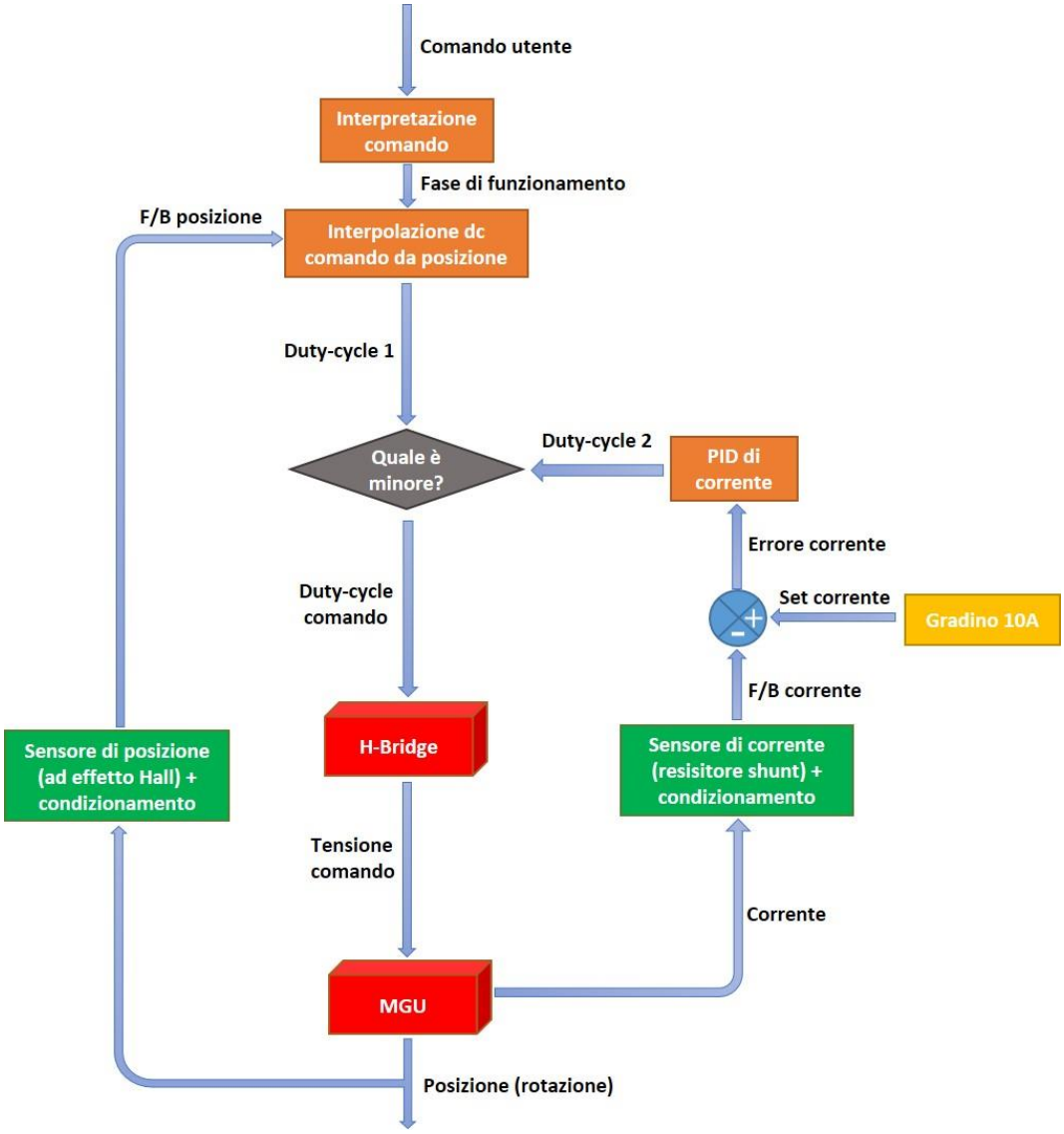


Fig. 4.18: diagramma esteso dell'algoritmo di controllo modificato

La fase di azionamento del *park-lock* inizia quando il *driver* a cui è collegato il motorino elettrico riceve e legge un comando di innesto o disinnesto, trasmesso dal *core* via I2C e derivante dall'azione di premuta del relativo pulsante sulla centralina da parte dell'utente. Quindi, l'interpretazione del comando porta alla definizione della fase di funzionamento (*DRV8701_phase*), la quale definisce la configurazione del ponte H e identifica il verso di applicazione della tensione di comando che rimarrà costante per tutta la fase di controllo. A questo punto, viene eseguita ciclicamente la funzione *motor_control*, che realizza il calcolo di

interpolazione che definisce mediante un ciclo for un *dc* di comando in funzione del valore attuale del *feedback* di posizione, come già ampiamente spiegato.

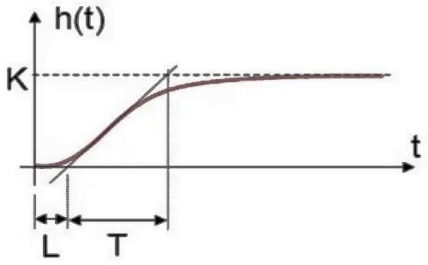
Inoltre, *motor_control* implementa un *loop* di tipo PID che riceve in input l'errore tra un set di corrente a gradino di ampiezza 10A (ossia costante) e produce come output un secondo *duty cycle* di comando, quello che sarebbe necessario applicare al ponte H per fare in modo che la tensione di comando spinga nella direzione di annullamento di tale errore. I due *dc* di comando calcolati vengono confrontati (per le motivazioni leggere il capitolo dedicato alla limitazione di corrente) e quello minore viene trasmesso all'*H-Bridge* e determina la tensione di alimentazione degli avvolgimenti della MGU.

Tale tensione, insieme alla forza controelettromotrice e alle varie cadute di tensione sugli avvolgimenti, determina la corrente richiamata dalla MGU; a sua volta la corrente produce una coppia motrice (di origine magnetica) agente sul rotore e l'equilibrio meccanico del dispositivo determina la legge di rotazione dell'alberino e, quindi, della leva. Per quanto riguarda la trasduzione delle grandezze di interesse, il sensore Hall con il suo sistema di condizionamento produce il feedback di posizione che viene usato per determinare il *dc* di comando interpolato; invece, il resistore shunt con il suo condizionamento genera il feedback per il *loop* di corrente adibito alla limitazione a 10A.

4.5 La calibrazione del controllo open loop

Una volta definita la struttura dell'algoritmo di controllo di un servosistema, è sempre necessario procedere alla sua calibrazione (o taratura), ossia alla definizione quantitativa dei parametri che lo caratterizzano. Qualora si fosse deciso di impiegare un sistema di controllo dotato di *loop* chiusi (come quello originale della scheda *driver*), la calibrazione avrebbe condotto alla definizione, per ogni *loop*, dei valori ottimali dei coefficienti nominati in precedenza: quello proporzionale, quello integrativo e/o quello derivativo. Nel caso venga utilizzato un controllore PID standard e il sistema controllato presenti un comportamento abbastanza lineare, esistono diversi metodi di calibrazione ampiamente validati: tra di questi i più famosi sono quelli di Ziegler-Nichols [31] basati sull'analisi della

risposta del sistema (approssimato con un primo ordine) ad un gradino di riferimento in ingresso (fig. 4.19) .



	K_P	K_I	K_D
P	T/L	0	0
PI	0.9 T/L	0.3/L	0
PID	1.2 T/L	0.5/L	0.5L

Fig. 4.19: formule per applicazione del metodo di Ziegler-Nichols in anello aperto

Al contrario, non esistono in letteratura metodi per calibrare un controllo in anello aperto come caratteristico dell'algoritmo nuovo: l'unica via possibile è stata andare per tentativi. Nello specifico, è stato necessario mettere in atto una campagna di prove sperimentali volta a definire opportunamente i vettori di riferimento per l'interpolazione, sia per la fase di innesto che per quella di disinnesto. La sequenza di fasi seguita per la calibrazione empirica del controllo (riassunta nelle righe successive) è stata ripetuta per entrambe le fasi di azionamento del sistema ed è stata portata avanti fino al raggiungimento di una corrispondenza ritenuta adeguata tra la legge di posizione reale e quella da riprodurre.

1) Due coppie di vettori di primo tentativo (ciascuna caratterizzata da elementi di identica lunghezza) sono inserite nel codice sorgente del *driver*.
2) Il progetto sorgente è compilato e quindi caricato come eseguibile sulla scheda *driver* che è collegata al *park-lock*.
3) Viene comandata l'attuazione di interesse mediante il pulsante di ENG MAIN o di DISENG MAIN.

4) Il movimento del dispositivo di sicurezza viene registrato mediante il progetto *CANape* associato alla centralina.
5) Il *datalog* di posizione salvato e quello di riferimento (la legge da imitare) vengono allineati nel tempo e confrontati.
6) Una o più coppie di *dc* (posizione-comando) vengono modificate oppure ne vengono aggiunte delle altre così che nella prova successiva la corrispondenza tra le due leggi aumenti.

Calibrazione per la fase di innesto (engage)

L'identificazione delle coppie di valori rappresentanti i punti di riferimento adeguati per l'interpolazione in fase di *engage* ha richiesto molti tentativi e sarebbe inutile riportarli tutti in questa sede; per questo motivo la tabella successiva riunisce i valori corrispondenti alle ultime prove fino alla presentazione della coppia di vettori definitivi (circondati da un bordo spesso nero), mentre la figura che la segue riporta le corrispondenti leggi *dc* di comando vs *dc* di posizione.

Tabella vettori di riferimento *engage*											
A	***dc rif. posizione***	0	50	55	60	61	80	81	100	/	/
	dc rif. comando	100	100	70	30	15	15	1	1	/	/
B	***dc rif. posizione***	0	50	55	60	61	80	81	100	/	/
	dc rif. comando	100	100	70	30	20	20	1	1	/	/
C	***dc rif. posizione***	0	50	55	60	61	70	74	80	81	100
	dc rif. comando	100	100	70	30	20	25	20	13	1	1

Tab. 4.2: tabella dei vettori di riferimento tentati per la fase di engage

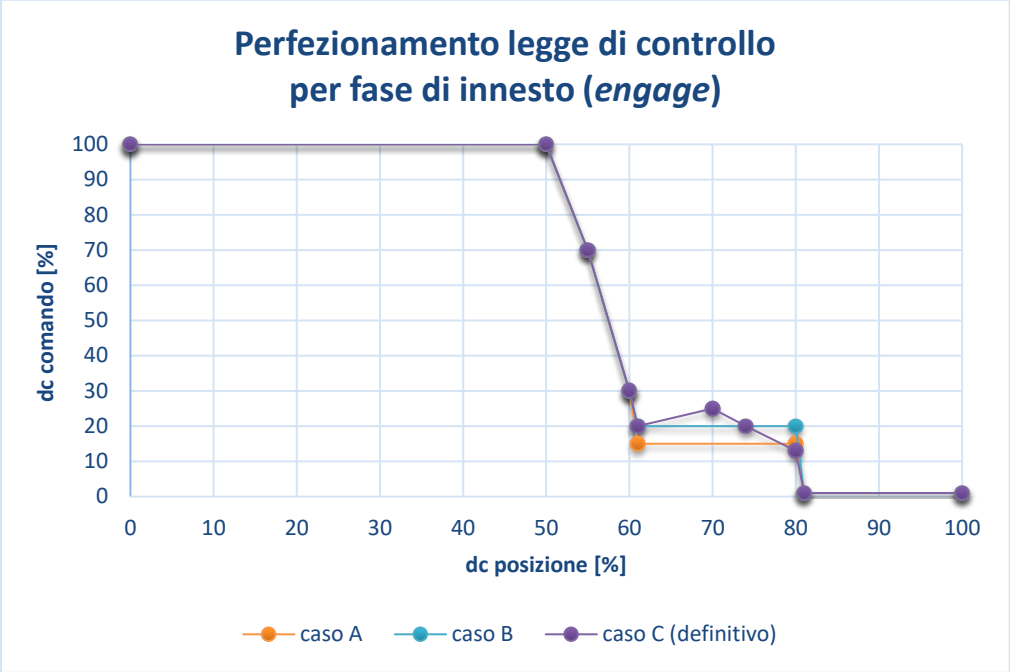


Fig. 4.20: grafico di perfezionamento della legge di comando per l'innesto

Le figure successive riportano i confronti di posizione e corrente per le tre prove considerate, così da poter effettuare alcune interessanti considerazioni.

Fig. 4.21: innesto, confronti relativi al caso A

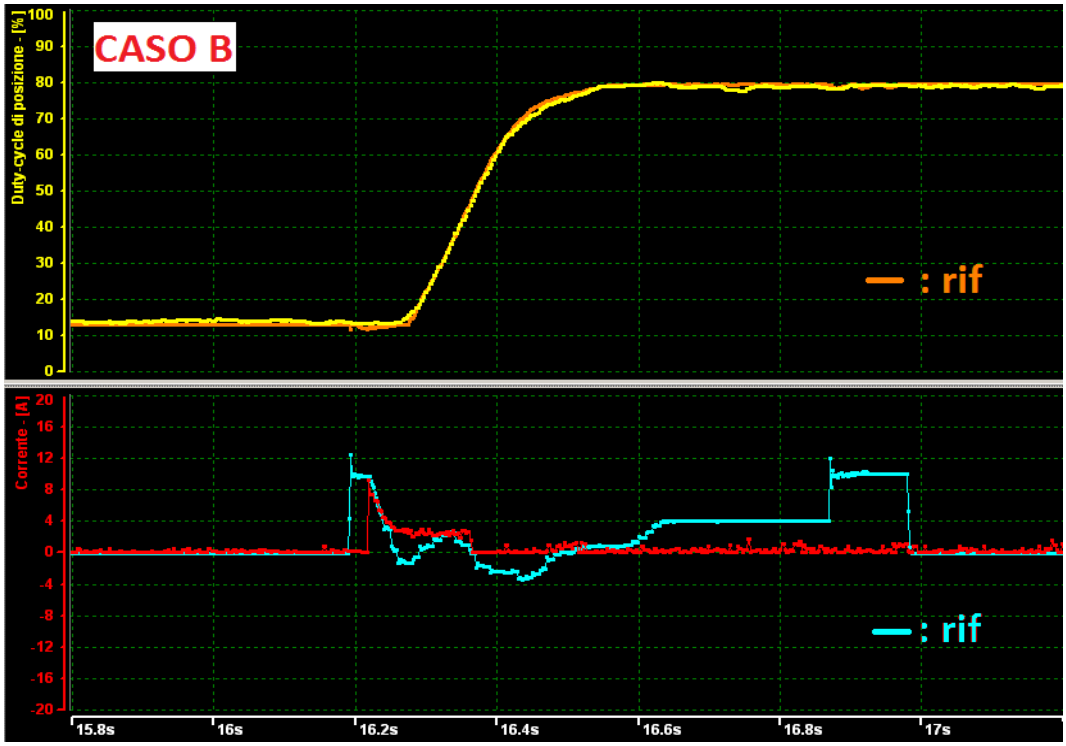


Fig. 4.22: innesto, confronti relativi al caso B

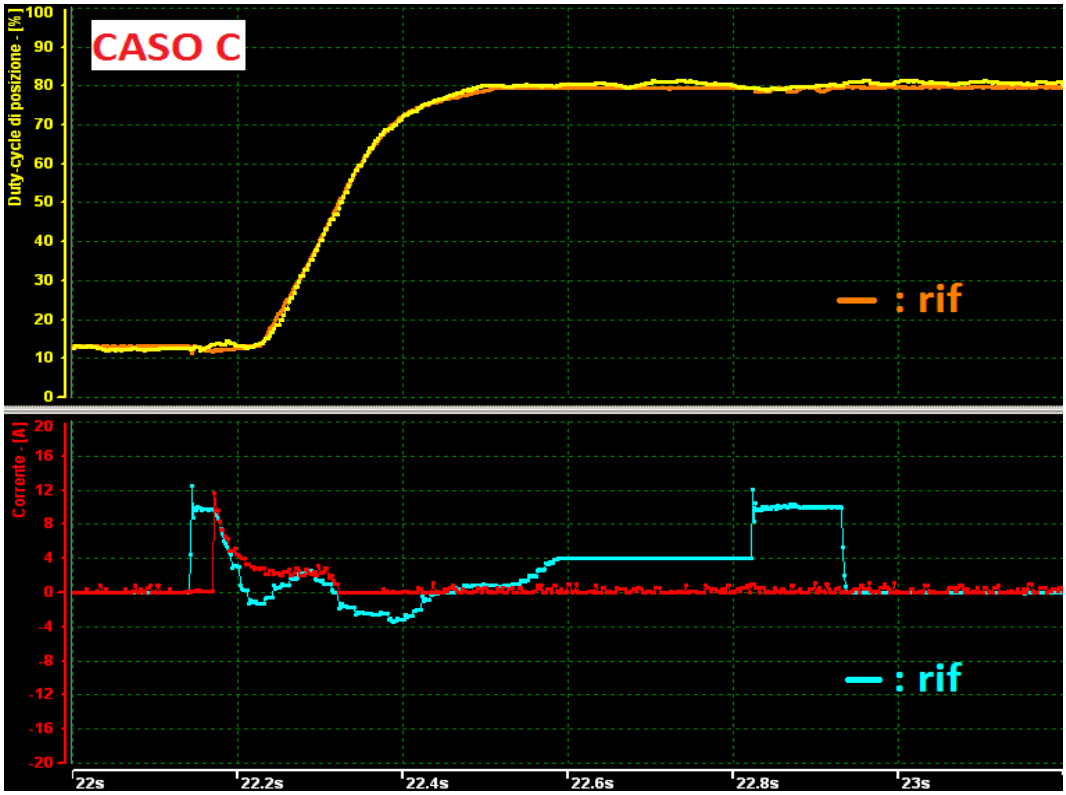


Fig. 4.23: innesto, confronti relativi al caso C (definitivo)

La figura 4.21 mostra come, utilizzando i valori di riferimento relativi al caso A, la sovrapposizione tra le due leggi di posizione non sia ottimale: le due curve corrispondono molto bene nel primo tratto caratterizzato da velocità elevata, fino ad un *dc* di posizione di circa il 65%, mentre per valori superiori lo scostamento tra le due curve aumenta fino ad un valore costante; in particolare, il *park-lock* rallenta più rapidamente di quanto dovrebbe e, addirittura, non viene raggiunta la configurazione di contato tra la leva porta-magnete e il finecorsa di innesto. Pertanto, nella transizione al caso B sono stati aumentati i *dc* di comando corrispondenti ai *dc* di posizione da 60% a 80% e questa azione ha portato (fig. 4.22) ad una riduzione dello scostamento tra le due curve in confronto. Per arrivare alla sovrapposizione pressoché perfetta del caso C (fig. 4.23) si è quindi proceduto ad aumentare ulteriormente i *dc* di comando corrispondenti al medesimo tratto di rotazione e il risultato rilevato ha permesso di confermare come definitivi i vettori di riferimento utilizzati.

Calibrazione per la fase di disinnesto (disengage)

Anche la determinazione dei punti di riferimento per la fase di *disengage* ha richiesto molti tentativi: la tabella successiva riporta i valori corrispondenti alle prove finali, inclusa la coppia di vettori definitivi (circondati da un bordo spesso nero), mentre la figura che la segue riporta le corrispondenti leggi *dc* di comando vs *dc* di posizione.

Tabella vettori di riferimento *disengage*																
A	***dc rif. posizione***	0	12	13	15	20	21	25	30	35	40	45	75	100		
	dc rif. comando	1	15	25	30	50	30	10	15	44	50	67	100	100		
B	***dc rif. posizione***	0	12	13	14	15	20	21	25	30	35	40	45	75	100	
	dc rif. comando	1	30	5	10	5	30	30	10	25	44	50	67	100	100	
C	***dc rif. posizione***	0	12	13	15	16	20	21	25	26	30	35	40	45	75	100
	dc rif. comando	1	30	5	20	10	23	20	1	10	27	50	53	65	100	100

Tab. 4.3: tabella dei vettori di riferimento tentati per la fase di disengage

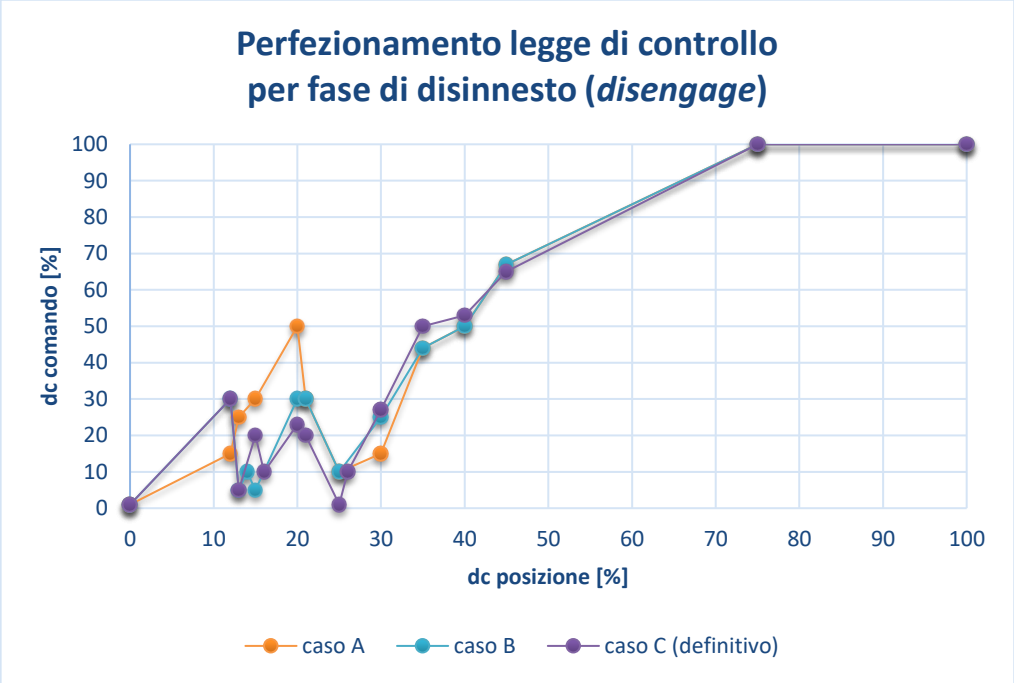


Fig. 4.24: grafico di perfezionamento per la legge di comando di disinnesto

Esattamente come nel caso dell'innesto, le figure seguenti riportano i confronti di posizione e corrente per le tre prove considerate e saranno accompagnate da osservazioni relative alle scelte di calibrazione.

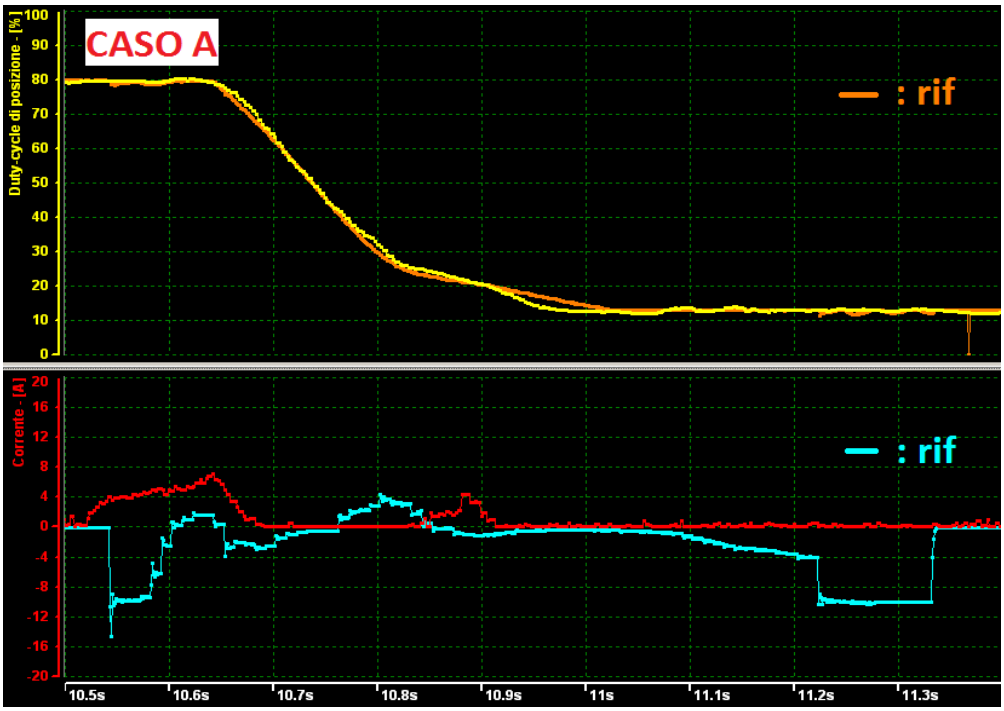


Fig. 4.25: disinnesto, confronti relativi al caso A

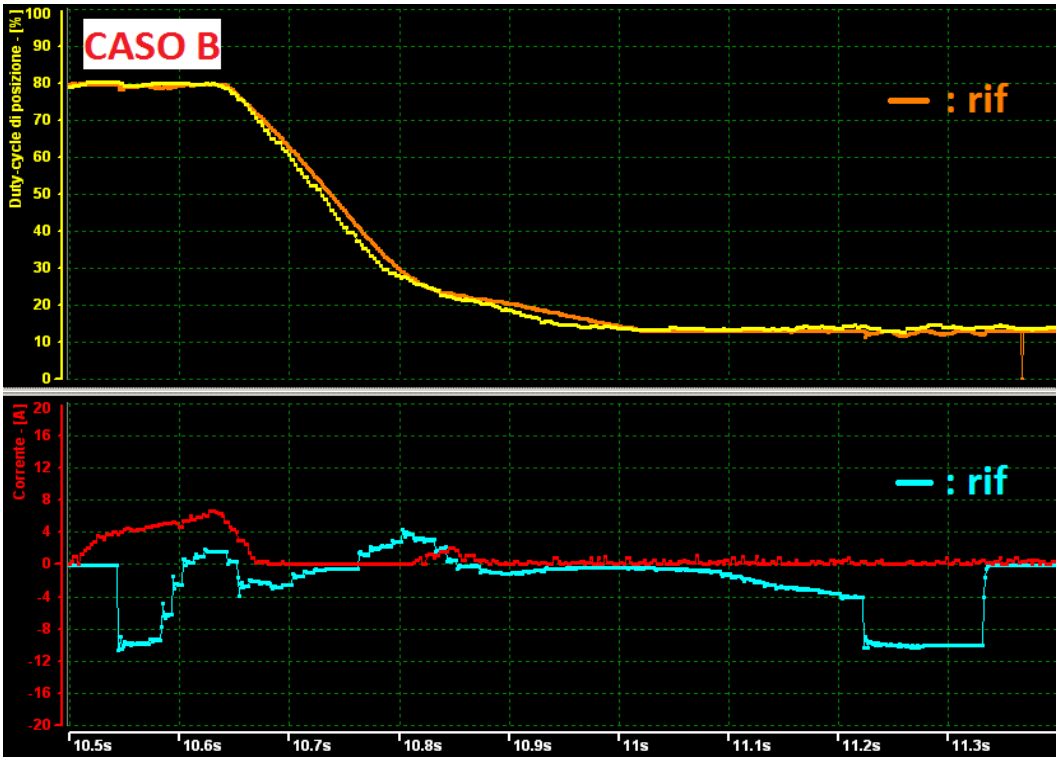


Fig. 4.26: disinnesto, confronti relativi al caso B

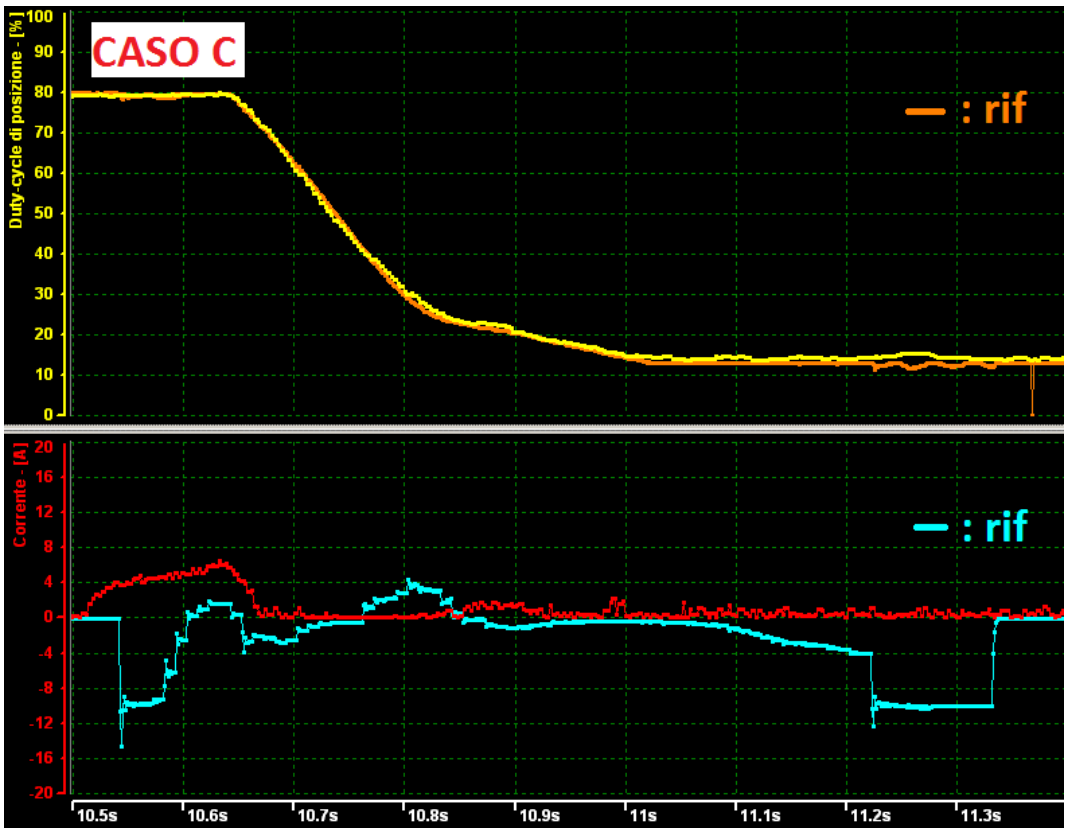


Fig. 4.27: disinnesto, confronti relativi al caso C (definitivo)

Per prima cosa è opportuno sottolineare un fatto che riguarda il confronto tra l'andamento della corrente relativo alla legge di riferimento e quello derivante dalla prova per la calibrazione. Si nota, infatti, che il primo *trend* di corrente è molto simile nella forma a quello della legge di innesto ma specchiato di 180° rispetto all'asse dei tempi; ciò implica che, mentre in *engage* la corrente misurata assorbita dal carico era prevalentemente positiva, in *disengage* diventa negativa per quasi tutta l'attuazione. Al contrario, sia in innesto che in disinnesto la corrente richiamata dal carico viene presentata da *CANape* come maggiore di zero, nonostante sia ovvio che la corrente di attuazione percorra gli avvolgimenti in versi opposti per i due casi.

Questa situazione è legata al fatto che il sistema di riferimento da cui proviene la relativa misura della corrente è configurato in modo più sofisticato di quanto non lo sia quello da me utilizzato per eseguire le prove. L'insieme centralina e progetto *CANape* che è stato utilizzato per rilevare la legge di posizione da imitare e la corrispondente legge di corrente può, infatti, far visualizzare all'utente sia il modulo che il verso della corrente richiamata dagli avvolgimenti mediante l'informazione del segno. Invece, il sistema ECU con interfaccia *CANape* che ho configurato può rappresentare a schermo solo l'intensità della corrente ma questo non è un problema perché, per ciascuna fase, la tensione di comando è applicata sempre nello stesso verso (stesso segno per tutti i *dc* di comando), indipendentemente dalla posizione e di conseguenza la corrente non può che percorrere gli avvolgimenti sempre nella stessa direzione.

Passando alla procedura di calibrazione, nel caso A (fig. 4.25) si osserva che la corrispondenza relativa al primo tratto di moto (*dc* da 80% a 20%) è accettabile, mentre sotto al 20% la posizione di finecorsa (13%) viene raggiunta più rapidamente del necessario. Dunque, per il caso B i *dc* di comando corrispondenti ai *dc* di posizione di riferimento inferiori al 20% sono stati ridotti, così da diminuire la spinta motrice applicata al *park-lock* in quel tratto (fig. 4.26). Siccome l'andamento di posizione ottenuto si discosta ancora nell'ultimo tratto da quello di riferimento (l'arpione raggiunge la posizione di finecorsa ancora troppo

velocemente), per il caso C i *dc* di comando finali sono stati ulteriormente ribassati, così da raggiungere la corrispondenza ottima desiderata (fig. 4.27).

4.6 Considerazioni sul tipo di controllo implementato

Precedentemente è stato osservato che esistono due principali strategie di controllo applicabili per regolare l'andamento di una certa variabile di interesse di un processo: si può operare in anello aperto o in anello chiuso. Ciascuno dei due possibili approcci racchiude, in realtà, una grande quantità di sotto-famiglie e varianti; ad esempio, in una struttura di regolazione in anello chiuso possono essere previsti dei sistemi atti a compensare automaticamente i disturbi sul processo o, ancora, possono venire utilizzate strategie di controllo adattativo per cui i parametri di un PID sono aggiornati a seconda del valore di determinati parametri operativi del sistema controllato [31].

È generalmente riconosciuto [31] che l'elemento che distingue un sistema di controllo in anello chiuso (CL) da uno in anello aperto (OL) è la presenza di un ramo di retroazione (di feedback), realizzato mediante un'opportuna catena di trasduzione. Un sistema OL opera, solitamente, ricevendo un input di riferimento e producendo un output di comando sul sistema senza che il reale andamento della variabile di interesse possa influenzarne il comportamento; al contrario, un sistema CL confronta un segnale di riferimento in ingresso con la misura attuale dell'uscita e procede per ridurre l'errore tra i due valori, avvicinando l'output all'input. La teoria dei controlli automatici [31] afferma che, a meno di costi eccessivi legati all'implementazione software e hardware, è sempre meglio seguire la strada del controllo CL; essa, infatti, presenta numerosi vantaggi rispetto a quella OL, tra i quali si annoverano i seguenti [36]:

- Maggiore precisione di raggiungimento del risultato desiderato.
- Compensazione automatica dei disturbi sul processo.
- Grande robustezza e affidabilità del controllo.
- Metodi di calibrazione dei parametri ampiamente validati.

Ciononostante, per la riproduzione della legge di spostamento desiderata ho deciso di passare ad una regolazione del processo di movimentazione del *park-lock* di tipo OL, con la peculiarità dell'interpolazione del segnale di comando sulla base della posizione misurata. L'utilizzo di una strategia OL per la gestione di un processo si può ritenere accettabile se sono verificate le condizioni elencate qui di seguito [36].

1) Il basso costo della soluzione di controllo è prioritario.
2) L'output del processo cambia raramente.
3) Non sono possibili misure quantitative delle grandezze di interesse.
4) Il processo è erratico, ossia ha caratteristiche che variano notevolmente nel tempo e in modo casuale.
5) I disturbi sul processo sono estremamente rari.

Le motivazioni per cui è stata presa la decisione di implementare la strategia di controllo precedentemente descritta sono principalmente quattro.

(1) Il sistema controllato (ossia MGU e *park-lock*) mediante la centralina è caratterizzato da un comportamento in fase di attuazione prevedibile e praticamente costante nel tempo. Gli attriti e i giochi che riguardano il moto dei componenti e lo spunto sono ridotti e poco influenzati dalle condizioni di contorno al funzionamento del sistema: temperatura, usura, inclinazione del prototipo, etc. Inoltre, la presenza della molla di torsione tra la leva e la camma consente all'attuatore di raggiungere la posizione di finecorsa di innesto mostrando scarsa sensibilità (a livello di coppia richiesta e di effetti dinamici) alla posizione del rocchetto nel momento dell'impatto.

(2) Le prime prove di fatica in cella avrebbero dovuto iniziare dopo un paio di mesi dal mio arrivo in azienda. Il ridotto tempo a disposizione per comprendere il funzionamento della centralina, configurare il sistema di misura e ottenere una legge di posizione accettabile mi ha condotto a scegliere, con il consenso del mio responsabile aziendale, la via della regolazione OL per la facilità di implementazione e la

velocità della calibrazione. Al contrario, l'utilizzo di un controllo CL adeguato avrebbe richiesto, come prima cosa, l'implementazione nei sorgenti di un *loop* per la produzione di un segnale di set adeguato (problema tutt'altro che banale); inoltre, sarebbe stato necessario calibrare i parametri del controllo PI o PID, processo che può anche richiedere molto tempo.

(3) Nel caso dell'utilizzo di un controllo ad anello chiuso, il *duty cycle* di comando viene determinato in modo autonomo dall'unità di controllo, sulla base dell'errore tra set e F/B. Nonostante l'implementazione (vedasi capitolo dedicato) di un algoritmo di limitazione della corrente, l'errore elevato che si registrerebbe, ad esempio, alla partenza dalla posizione di disinnesto condurrebbe ad un picco di corrente di ampiezza molto elevata; tale picco, benché transitorio, potrebbe danneggiare l'*H-Bridge* di comando ed è quindi una condizione da evitare. Invece, il tipo di controllo utilizzato ha consentito di controllare in modo più preciso la tensione di comando e la corrente assorbita (soprattutto allo spunto), risultando la soluzione preliminare migliore per un utilizzo sicuro del *driver*.

(4) Infine, è già stato messo in evidenza che l'algoritmo di controllo implementato è una via di mezzo tra un *open loop* ed un *Feed-Forward*: il riferimento per la funzione di controllo deriva dalla diretta misura della variabile controllata (la posizione). Quest'impostazione peculiare rende il sistema di controllo meno affidabile e reattivo di un FF ma sicuramente più di uno in OL, nel quale una variazione indesiderata dell'output non determinerebbe alcuna controreazione in termini di variazione dell'uscita del controllore.

CAPITOLO 5

Attività di affinamento del sistema configurato

Nel seguente capitolo verranno presentate le attività che sono state realizzate al fine di perfezionare sia le funzionalità del sistema di controllo che quelle dell'interfaccia utente. Le implementazioni descritte di seguito non sono di importanza secondaria rispetto alle precedenti; in mancanza di esse non sarebbe stato possibile, infatti, attuare le prove di fatica in modo affidabile e automatizzato e, inoltre, anche la registrazione e la visualizzazione delle grandezze mediante *CANape* sarebbe stata inadeguata e inutile.

5.1 Constatazioni preliminari

La maggior parte delle attività di perfezionamento verrà descritta seguendo la seguente successione di passaggi logici e operativi:

1) Viene presentato un *datalog* (ossia una registrazione di dati) ottenuto con *CANape* che mette in evidenza un determinato problema da risolvere o un certo aspetto da migliorare.
2) Dopo aver ragionato sulla causa del problema, si identifica la porzione di codice sorgente da modificare per superare la criticità.
3) Il codice sorgente viene adeguatamente modificato e caricato sulla relativa scheda elettronica.
4) Si presenta un nuovo *datalog CANape* per dimostrare che l'elaborazione dei sorgenti ha risolto la criticità osservata.

Tutte le prove relative a questo capitolo sono state effettuate attuando il motore elettrico un *park-lock* mediante il *driver* MAIN, quindi non è stato necessario comandare due attuatori contemporaneamente.

Le operazioni di calibrazione del sistema di lettura delle grandezze di interesse sarebbero dovute essere realizzate per entrambe le centraline a disposizione per l'esecuzione delle prove di fatica. In realtà, la taratura ha riguardato unicamente la ECU numero 2, a cui faranno riferimento tutti i risultati sperimentali e le elaborazioni presentate di seguito, mentre per la ECU 1 non è stato possibile lo stesso trattamento. Infatti, nonostante tutte le implementazioni volte al miglioramento sostanziale dell'interfaccia utente, gli andamenti ottenuti dalle informazioni trasmesse dalla centralina 1 hanno sempre mantenuto l'aspetto mostrato in figura 5.1.

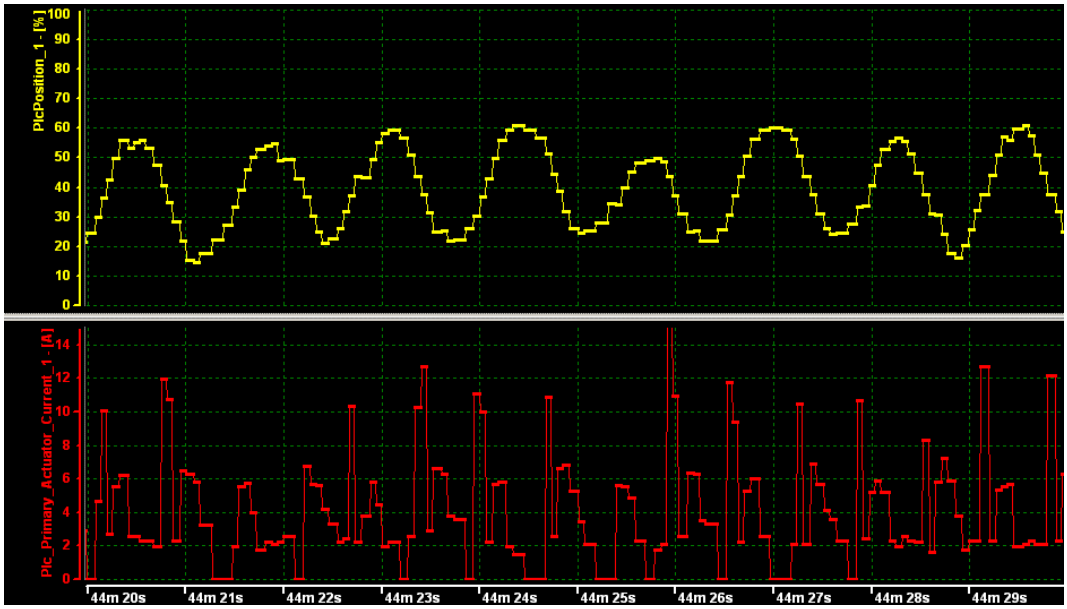


Fig. 5.1: andamenti di posizione e corrente ottenuti dalla ECU 1

I grafici riportati nell'immagine precedente non descrivono certamente il comportamento reale dell'attuatore del *park-lock*. Innanzitutto, nonostante le ottimizzazioni della risoluzione di lettura e del periodo di aggiornamento dei messaggi CAN che verranno spiegate nei paragrafi seguenti, i *trend* sono costituiti da sequenze di tratti orizzontali costanti e ciò li rende inutilizzabili per qualsiasi genere di considerazione e rilevazione quantitativa di dati. In secondo luogo, è stato notato che i segnali CAN trasmessi dalla ECU 1 sono in ritardo (per quanto riguarda il valore indicato) rispetto alla quantità fisica che dovrebbero descrivere. Ad esempio, la figura 5.1 mostra un *dc* di posizione che oscilla irregolarmente tra il

25% e il 60%, perciò da esso si potrebbe dedurre che l'attuatore non raggiunga le due posizioni di finecorsa: in realtà non è così, è colpa del segnale di posizione che non riesce a seguire correttamente la rapida inversione del moto.

Il malfunzionamento della ECU 1 è dovuto alla completa mancanza di funzionamento del *driver* AUX e, in particolare, all'assenza di partecipazione alla comunicazione I2C; il *core*, quando interroga il registro della scheda AUX nel *task* relativo, non riesce a leggere alcuna informazione e ciò gli impedisce di produrre dei dati CAN adeguati per la trasmissione. Pertanto è ovvio il motivo per cui la calibrazione del sistema di lettura e memorizzazione interna dei valori di posizione e corrente sia risultata impossibile per la ECU 1. Non è stato un problema, invece, tarare l'algoritmo di controllo in *open loop* del *park-lock* gestito dalla centralina 1, in quanto è stato sufficiente testarlo con il *driver* MAIN della ECU 1 e caricarlo successivamente su quello della ECU 2; si osserva, però, che quest'ultima pratica è basata sull'ipotesi che le due ECU presentino la stesse caratteristiche di trasduzione dei segnali dai sensori: essa è reputata accettabile in quanto sia la struttura hardware che quella software sono i medesimi.

Come già sottolineato, la centralina malfunzionante produce, tramite l'interfaccia *CANape*, grafici e registrazioni che non possono essere utilizzate per analisi di tipo quantitativo e quindi sembrerebbe un sistema inadatto per il controllo di una prova di fatica. La necessità di portare a compimento tali test in un lasso di tempo ristretto ha reso inevitabile l'utilizzo delle due centraline in parallelo per il controllo simultaneo di due dispositivi di sicurezza. Ciò è stato possibile perché le prove di fatica molla richiedono delle condizioni operative molto meno controllate di quelle della fatica leva; infatti, mentre per la verifica della durabilità della leva è fondamentala un adeguato controllo della velocità a fine innesto e della corrente assorbita dall'attuatore, la sollecitazione a torsione percepita dalla molla è determinata quasi unicamente dall'escursione angolare della camma. Quindi, verificata la condizione di contatto tra arpione e dente rocchetto alla fine della corsa di innesto (massima torsione della molla), la ECU 1 ha consentito adeguatamente di tener traccia del numero di cicli compiuti nelle prove di fatica relative alla molla.

5.2 Periodo di trasmissione dei messaggi CAN

Osservando il grafico di *CANape* di una grandezza fisica ricevuta su PC via CAN, nel caso in cui sulle schede siano stati caricati i sorgenti originali e non siano state fatte modifiche ai fattori di scala dei database, è evidente che l'andamento visualizzato è ottenuto dall'unione di una sequenza di punti molto radi (che rappresentano i valori effettivamente ricevuti via CAN) raccordati da tratti lineari. La figura 5.2 mostra, ad esempio, l'andamento del *dc* di posizione (*PlcPosition*) e della corrente assorbita dal motore (*Plc_Primary_Actuator_Current*) in seguito ad un comando di *engage* seguito da uno di *disengage*; inoltre in fig. 5.3 è possibile osservarne un ingrandimento.

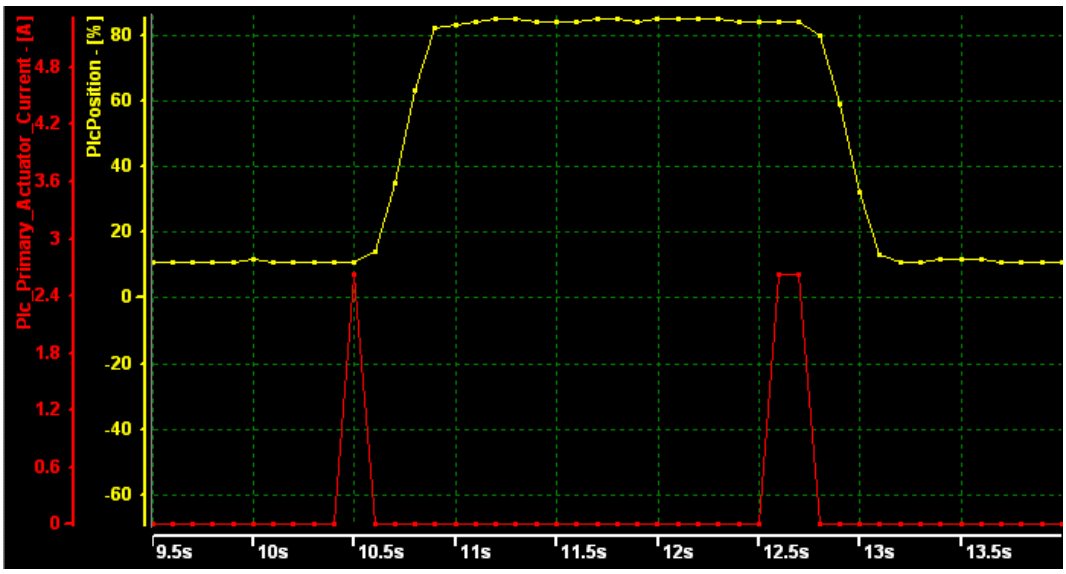


Fig. 5.2: periodo di visualizzazione dei segnali CAN paria a 100ms

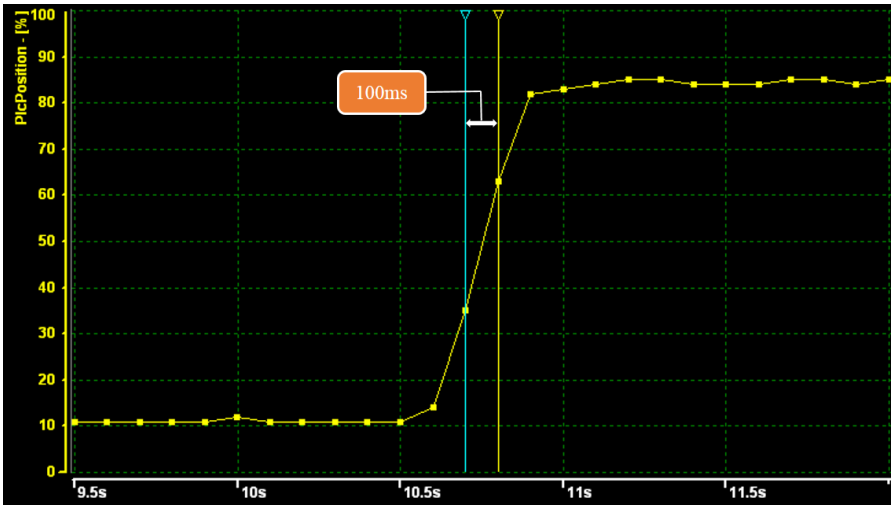


Fig. 5.3: periodo di visualizzazione dei segnali CAN paria a 100ms, ingrandimento

La seconda immagine indica che la distanza tra due punti contigui, ricevuti e letti dal progetto, è esattamente uguale a 100ms. Il grado di raffinatezza della visualizzazione su *CANape* di una grandezza ricevuta via CAN dipende da due fattori:

(1) Dalla frequenza di lettura del segnale, ossia quella con cui il PC sonda il bus CAN per registrare il contenuto di un determinato segnale.
(2) Dalla frequenza con cui il segnale di interesse (contenuto ovviamente in un messaggio) viene immesso sul bus dal nodo di pertinenza.

Come già sottolineato nel capitolo precedente, la frequenza di lettura dei segnali in input è stata impostata per adattarsi automaticamente a quella di trasmissione da parte della ECU. Invece, le modalità di trasmissione del messaggio dipendono da alcune impostazioni relative alla scheda *core* e gestibili attraverso opportune modifiche ai sorgenti.

Le proprietà relative alla trasmissione del *Data Frame* di uscita della ECU sono determinate dai seguenti due parametri:

- La frequenza con cui opera il *transceiver* che serve *core*, ossia il ritmo temporale con cui il messaggio viene diffuso sul bus CAN.
- La frequenza con cui il contenuto del messaggio viene aggiornato per rispecchiare lo stato attuale delle schede e del *park-lock*.

Il valore del primo parametro è definito all'interno della struttura che identifica le proprietà generali del messaggio di uscita, la quale presenta il nome di *can_message_map_PARKLOCK*; in particolare, il periodo di funzionamento del *transceiver* è specificato nel campo *Period* (fig. 5.4).

```
Data[8]                                  | Period     | Timeout    | Periodicity
0x11,0x00,0xff,0xff,0xff,0xff,0xff,0xff, 0x00000064, 0x00000000, CYCLIC
```

Fig. 5.4: periodo di trasmissione del Data Frame di uscita pari a 100ms

Il periodo in questione è originariamente impostato al valore esadecimale 0x00000064 che, convertito in decimale, corrisponde a 100; ciò indica che il messaggio CAN di output è trasmesso sul bus per raggiungere il *CANcase* e poi il PC ogni 100ms, coerentemente con quanto rilevato nel *datalog* di figura 4.1.1. Dunque, al fine di migliorare la visualizzazione degli andamenti temporali su *CANape*, il parametro appena evidenziato è stato posto a 1ms, ossia a 0x00000001 in notazione esadecimale (fig. 5.5).

```
Data[8]                                  | Period     | Timeout    | Periodicity
0x11,0x00,0xff,0xff,0xff,0xff,0xff,0xff, 0x00000001, 0x00000000, CYCLIC
```

Fig. 5.5: periodo di trasmissione del Data Frame di uscita posto a 1ms

L'immagine successiva consente di osservare l'effetto di tale modifica sugli andamenti di posizione e di corrente.

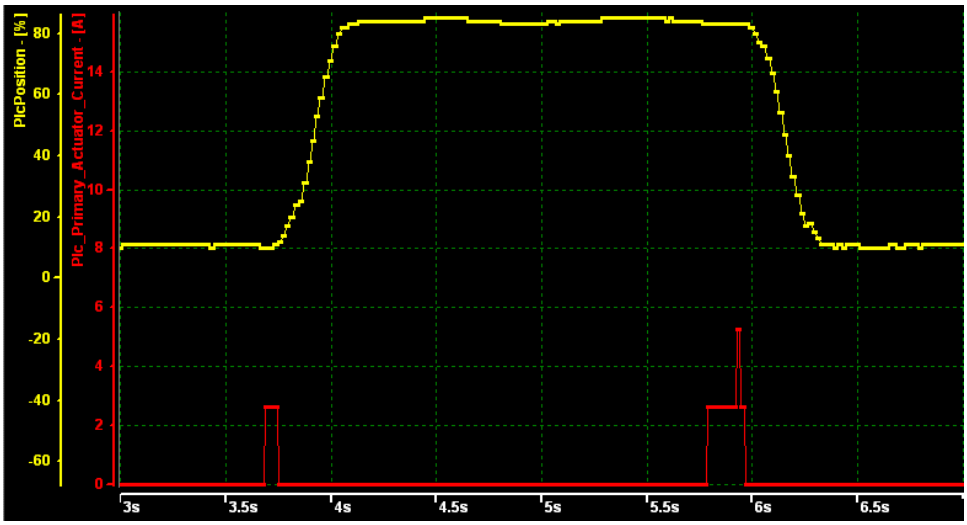


Fig. 5.6: effetto del periodo di trasmissione posto a 1ms

Confrontando la figura precedente con quanto mostrato in fig. 5.3 si osservano sicuramente dei miglioramenti ma insufficienti per poter considerare il risultato accettabile: il numero di punti utilizzati da *CANape* per creare il grafico è maggiore però l'andamento continua ad essere a gradini. Le figure 5.7 e 5.8 mostrano due ingrandimenti progressivi del grafico di fig. 5.6, utili per osservare meglio le caratteristiche dei gradini.

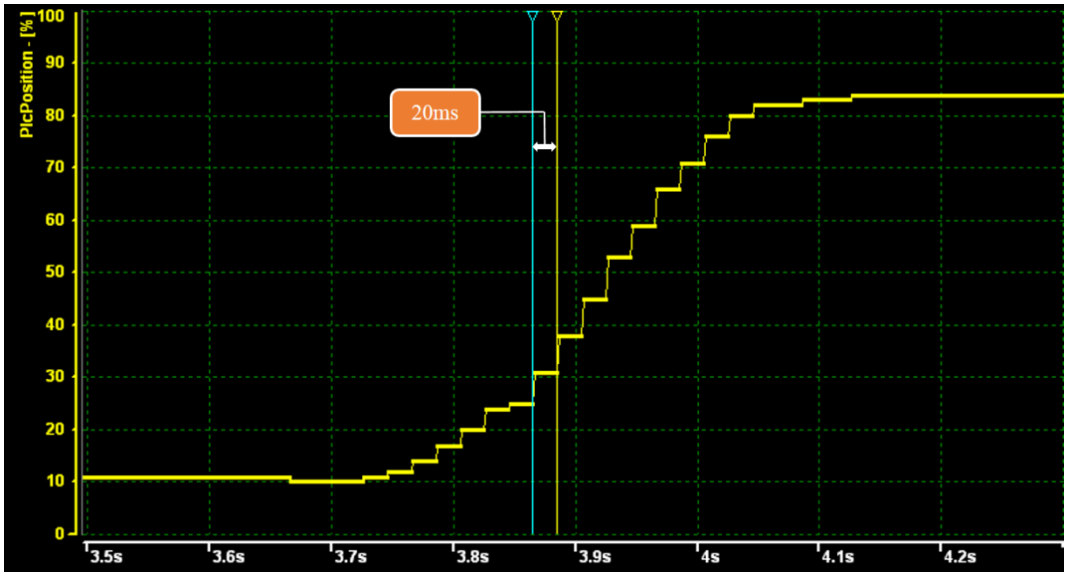


Fig. 5.7: effetto del periodo di trasmissione posto a 1ms, ingrandimento n.1

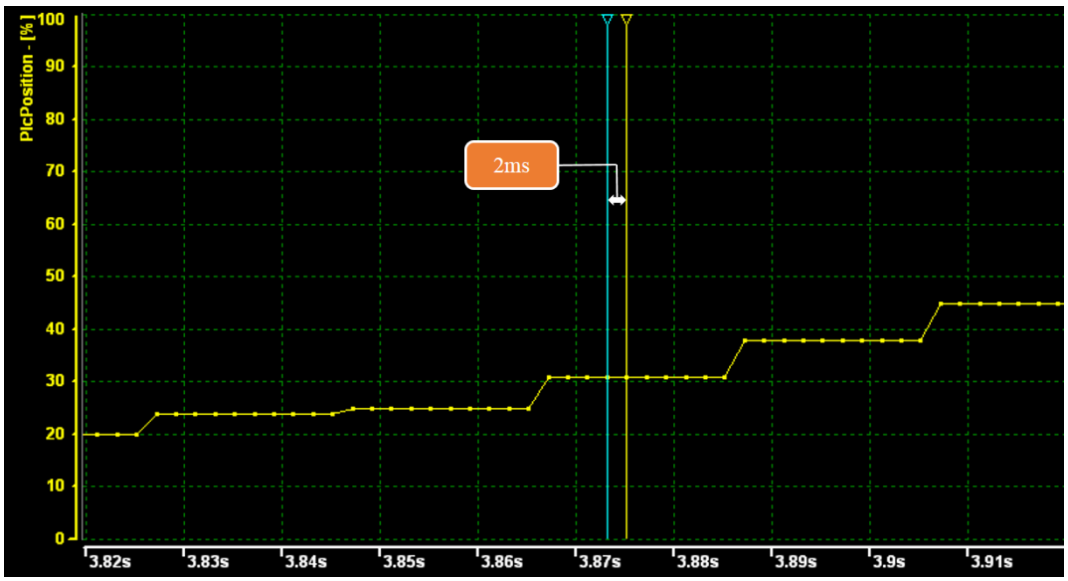


Fig. 5.8: effetto del periodo di trasmissione posto a 1ms, ingrandimento n.2

Le due immagini precedenti mostrano che;

1) Il periodo di trasmissione del messaggio di output vale 2ms e non 1ms (fig. 5.8) come desiderato.
2) Il periodo di aggiornamento del valore dei segnali è pari a 20ms (fig. 5.7), e ciò determina il mantenimento del *trend* a scalini.

Nel secondo capitolo è stato spiegato che due dei tre processi realizzati dal *core* sono strettamente legati alla comunicazione CAN: il *task* di comunicazione con i *driver* e quello di creazione dei messaggi CAN. Infatti, la lettura e la memorizzazione dei valori di corrente, stato e *fault* e la scrittura dei comandi relativi ai due *driver* viene eseguita tramite il primo dei due processi citati (*parklock_task*); invece, la costruzione del *Data Frame* di uscita (a livello di struttura) e la definizione delle proprietà della comunicazione CAN sono gestite dal secondo processo (*can_task*). Il periodo di aggiornamento del contenuto dei messaggi CAN è sicuramente legato a quello dei due *task* appena nominati: se il periodo di costruzione del messaggio di uscita o quello di lettura delle informazioni trasmesse dai *driver* sono elevati, anche il contenuto del messaggio varierà con frequenza bassa; in particolare, è facile dedurre che il periodo con cui si vede mutare il valore delle grandezze su *CANape* è il maggiore tra quelli dei due processi indicati.

Pertanto, è stato modificato l'*header taskConfig.h*, parte del progetto del *core*, che definisce i parametri di riferimento per lo *scheduler* del sistema operativo e, in particolare, i periodi di esecuzione dei processi; la figura che segue mostra i periodi previsti dall'*header* originale.

```
#define TASKCONFIG_GPIO_TIME_PERIOD          50  /* [ms] */
#define TASKCONFIG_CAN_TIME_PERIOD           2   /* [ms] */
#define TASKCONFIG_PARKLOCK_TIME_PERIOD      20  /* [ms] */
```

Fig. 5.9: periodi di esecuzione dei task originali

Il *task* di gestione degli input e degli output generici è realizzato ogni 50ms, quello della gestione della comunicazione CAN ogni 2ms e quello della comunicazione con i *driver* ogni 20ms: questo ultimo valore spiega l'ampiezza temporale dei gradini osservati su *CANape*. Anche se è evidente che i periodi di

scheduling non possono essere l'unica causa dell'andamento a gradini evidenziato in precedenza, si è proceduto ad impostare tutti i valori a 1ms (fig. 5.10), al fine di rendere la scheda *core* più performante e velocizzare le interazioni con gli elementi che la circondano.

```
#define TASKCONFIG_GPIO_TIME_PERIOD          1  /* [ms] */
#define TASKCONFIG_CAN_TIME_PERIOD           1  /* [ms] */
#define TASKCONFIG_PARKLOCK_TIME_PERIOD      1  /* [ms] */
```

Fig. 5.10: periodi di esecuzione dei task posti a 1ms

Il risultato della correzione effettuata su *taskConfig.h* si può osservare nelle tre immagini sottostanti.

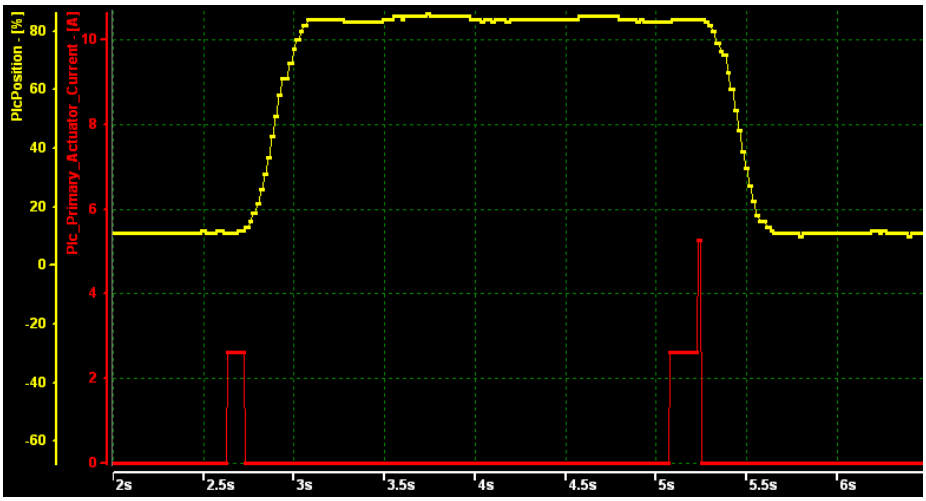


Fig. 5.11: effetto dei periodi di esecuzione dei task posti a 1ms

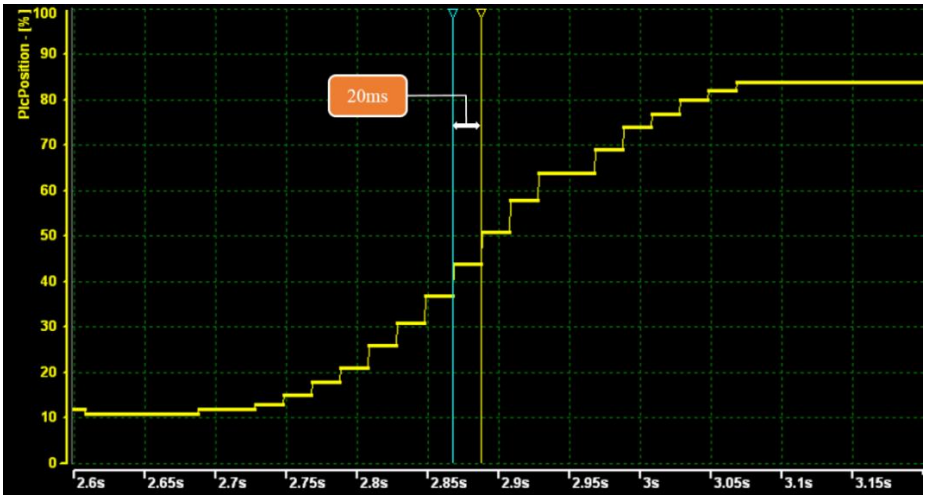


Fig. 5.12: effetto dei periodi di esecuzione dei task posti a 1ms, ingrandimento n.1

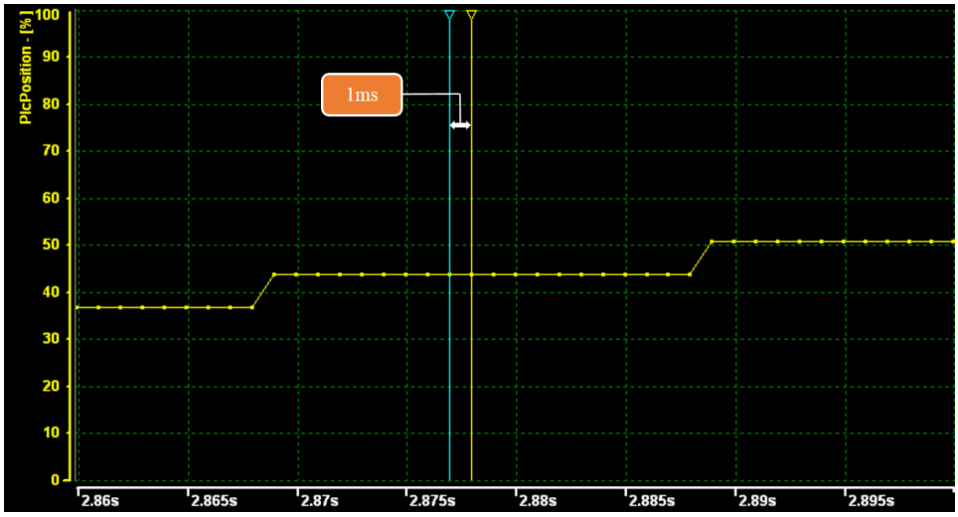


Fig. 5.13: effetto dei periodi di esecuzione dei task posti a 1ms, ingrandimento n.2

È evidente che il risultato non è quello auspicato: l'andamento rilevato è ancora a scalini e sempre di ampiezza 20ms. Analizzando attentamente il file *parklock.c* ho trovato un parametro di grande importanza, che definisce il tempo minimo necessario per terminare l'esecuzione di tutti e tre i processi del *core*: si tratta della costante intera *PARKLOCK_MUTEX_TIMEOUT* (fig. 5.14).

```
#define PARKLOCK_MUTEX_TIMEOUT      10      /* [ms] */
```

Fig. 5.14: PARKLOCK_MUTEX_TIMEOUT pari a 10ms

Questo valore viene chiamato due volte all'interno di *parklock_task* (la funzione principale di *parklock.c*), le cui istruzioni implementano la gestione delle interazioni tra *core* e *driver*. Il *task* in questione prevede la realizzazione di due fasi in successione:

(1) Lettura via I2C dei valori di stato, di *fault* e delle correnti relative ai *driver* e attribuzione di essi a variabili temporanee; inoltre, viene calcolato (come media mobile) e memorizzato il valore del *dc* di posizione; quest'ultimo, però, non proviene dalla comunicazione con i *driver* ma è ricavato dai 10 valori presenti nel *buffer* dedicato annesso al convertitore A/D del *core*.

(2) Scrittura via I2C in appositi registri di memoria dei *driver* di eventuali comandi di aggancio/sgancio/*reload* ricevuti tramite la pulsantiera

della ECU ed interpretati dal *core*. Si evidenzia che, nel caso di ricezione di un comando, la scheda *core* può trasmetterlo (scrivendo dei valori in opportuni registri associati alla comunicazione I2C) ad una sola scheda *driver* o ad entrambe, a seconda delle proprietà impostate nello stesso progetto sorgente.

Per evitare conflitti nella comunicazione *core-driver*, l'esecuzione del processo in questione prevede che ciascuna delle due fasi mantenga il bus I2C occupato (bloccando l'adempimento della fase successiva) per un intervallo di tempo in ms pari alla costante *PARKLOCK_MUTEX_TIMEOUT*. Il nome di questo parametro vede l'intervento di due termini fondamentali:

- ***MUTEX***: corrisponde a *MUTual EXclusion*, ossia mutua esclusione, in quanto nella situazione considerata ci sono due processi che possono bloccarsi a vicenda, quando è opportuno, l'accesso ad una determinata risorsa (il bus I2C).
- ***TIMEOUT***: indica l'esistenza di un parametro che segnala per quanto tempo una fase può bloccare la risorsa in questione; scaduto il tempo a disposizione, il processo deve essere interrotto per lasciare spazio a quello successivo.

Quindi, il fatto che il contenuto del messaggio CAN di fig. 5.12 venga aggiornato ogni 20ms è dovuto alla scelta di un valore di 10ms per il *timeout*, il quale determina una durata complessiva di 20ms per il *task* considerato. Si nota, quindi, che il periodo di esecuzione del processo per la comunicazione tra *core* e *driver* è stato originariamente posto a 20ms per coerenza con la sua durata effettiva; pertanto, aver ridotto il periodo a 1ms non poteva condurre a miglioramenti nell'andamento in quanto il *task* sarebbe continuato a durare complessivamente 20ms.

In definitiva, il valore di *PARKLOCK_MUTEX_TIMEOUT* è stato ridotto al minimo, ossia 1ms (fig. 5.15); inoltre, siccome il periodo di aggiornamento del CAN sarebbe diventato 2ms (il doppio del *timeout*), sia il periodo di esecuzione del *task* di costruzione del messaggio CAN che il suo periodo di trasmissione sono stati

alzati a 2ms, così da non caricare inutilmente di lavoro le risorse hardware della scheda *core*.

```
#define PARKLOCK_MUTEX_TIMEOUT      1       /* [ms] */
```

Fig. 5.15: PARKLOCK_MUTEX_TIMEOUT posto a 1ms

La tabella seguente riassume i valori definitivi che sono stati attribuiti ai parametri salienti per la qualità della visualizzazione dei segnali su *CANape.*

Parametro	Valore [ms]
Periodo trasmissione messaggio CAN	***2***
Periodo comunicazione I2C	***2***
Periodo creazione messaggio CAN	***2***
Periodo gestione GPIO	***10***
Timeout per mutex I2C	***1***

Tab. 5.1: valori che definiscono la qualità di visualizzazione dei segnali via CANape

Il risultato a livello grafico derivante dalla configurazione finale è presentato nelle figure sottostanti.

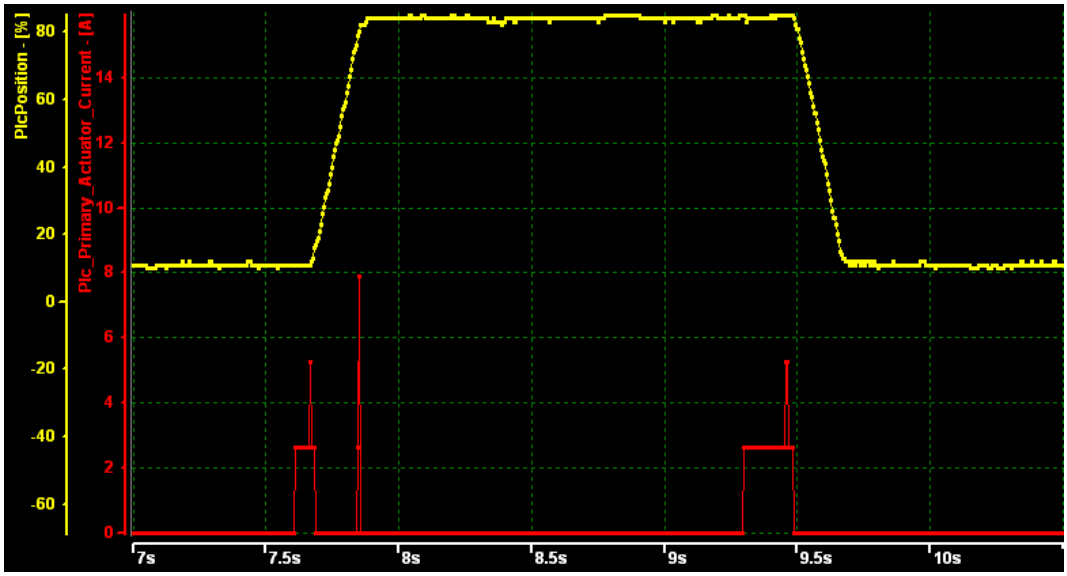


Fig. 5.16: effetto di PARKLOCK_MUTEX_TIMEOUT posto a 1ms

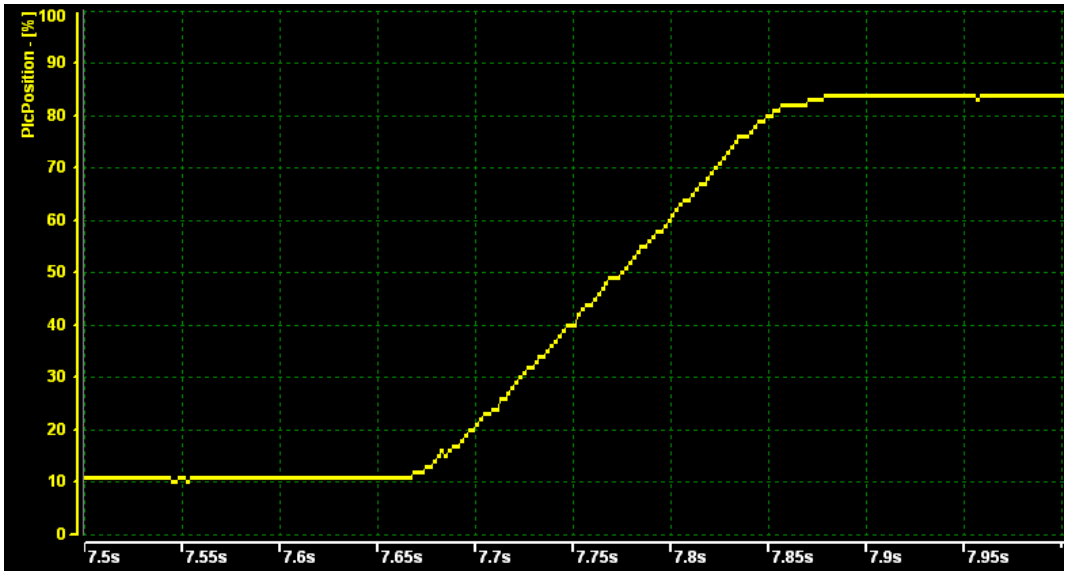


Fig. 5.17: effetto di PARKLOCK_MUTEX_TIMEOUT posto a 1ms, ingrandimento n.1

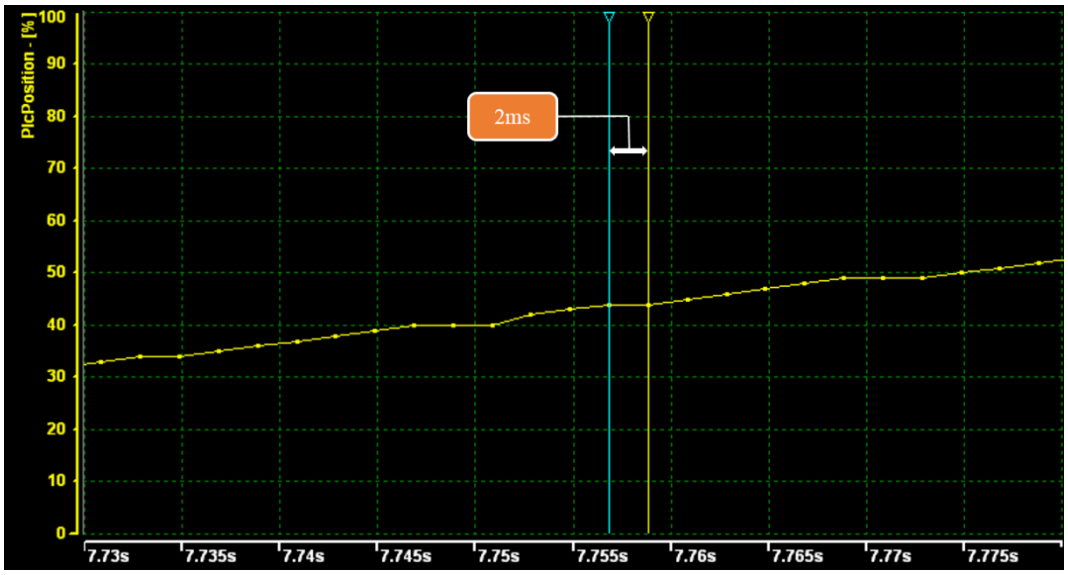


Fig. 5.18: effetto di PARKLOCK_MUTEX_TIMEOUT posto a 1ms, ingrandimento n.2

L'andamento di fig. 5.17 è notevolmente più regolare di quelli presentanti in precedenza: lo zoom di fig. 5.18 conferma il fatto che adesso il periodo con cui il computer riceve i segnali CAN e quello di aggiornamento del contenuto sono identici e corrispondono a 2ms.

Il risultato ottenuto si può ritenere soddisfacente e gli andamenti che verranno visualizzati in affiancamento alla descrizione delle attività successive potranno essere ritenuti attendibili e rappresentativi della reale evoluzione temporale dei

segnali in questione. Queste considerazioni non sono casuali ma si basano sul teorema del campionamento (o di Nyquist); questo teorema afferma che la frequenza minima con cui bisogna campionare una grandezza fisica al fine di rappresentarla in modo fedele e senza alterazioni è pari al doppio del massimo contenuto in frequenza della grandezza stessa; qualora si campioni ad una frequenza inferiore al limite appena definito (detto 'frequenza di Nyquist'), il fenomeno dell'*aliasing* crea dei contenuti in frequenza numerici, assenti nel segnale iniziale, e quindi l'andamento frutto del campionamento potrebbe non rispecchiare affatto quello di riferimento [37]. Tuttavia, l'acquisizione di un segnale ad una frequenza molto più elevata determina il riconoscimento di disturbi, legati soprattutto alla parte elettrica ed elettronica coinvolta nella trasmissione di dati, che sarebbe preferibile non visualizzare perché anche essi slegati dalla fisica del sistema osservato.

La figura 5.17 rappresenta il grafico del *duty cycle* di posizione quando la scheda *driver* che guida l'attuatore riceve il comando di ENG MAIN. La durata della fase di salita è pari a circa 200ms e si può verificare che lo stesso vale per quella di discesa; tali dati riguardano l'impiego di un *driver* che esegue l'algoritmo di controllo originale, mentre si è osservato che la legge del moto copiata prevede corse di innesto e disinnesto con durata media pari a circa 300ms. Quindi, assimilando le leggi di attuazione a quarti di solenoide, il massimo contenuto in frequenza delle leggi di posizione si può approssimare come:

$$f_{pos_{MAX}} = \frac{4}{0.3s} = 13.3Hz \tag{5.1}$$

Dunque, la frequenza di Nyquist della posizione corrisponde a:

$$f_{pos_{NYQ}} = 2 * f_{pos_{MAX}} = 26.6Hz \tag{5.2}$$

La frequenza di campionamento e visualizzazione dei segnali provenienti dalla centralina è:

$$f_{sampling} = \frac{1}{0.002s} = 500Hz \tag{5.3}$$

Il segnale di posizione è acquisito da *CANape* con una frequenza circa 19 volte superiore a quella di Nyquist e ciò consente di evitare l'*aliasing*. La buona norma dice che è sufficiente campionare a 10-20 volte la frequenza di Nyquist per ottenere una visualizzazione accettabile di un segnale; in questo caso si è deciso di mantenere il ritmo di acquisizione a 500Hz, il valore più alto ottenibile con il sistema a disposizione, così da poter osservare adeguatamente anche la dinamica della corrente, che presenta dei transitori decisamente più rapidi di quelli della posizione in entrambe le fasi di attuazione.

5.3 Risoluzione di lettura di posizione e corrente

La figura 5.19 mostra, in un'unica finestra grafica, gli andamenti di *dc* (giallo) e corrente (rosso) per le fasi di *engage* e *disengage* realizzate secondo l'algoritmo originale; la figura 5.20 ne presenta un ingrandimento focalizzato sull'innesto.

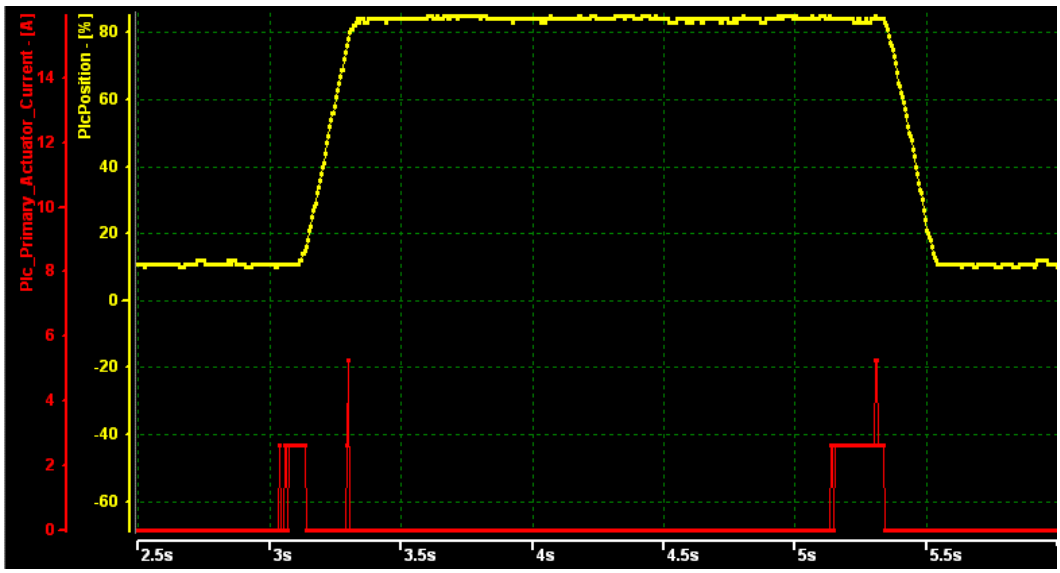


Fig. 5.19: andamenti di posizione e corrente

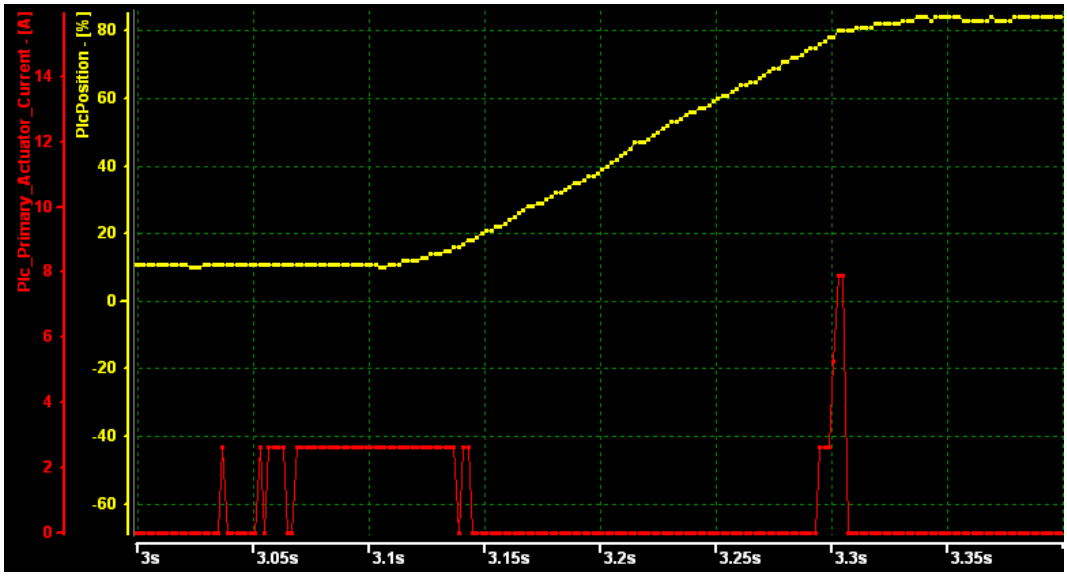


Fig. 5.20: andamenti di posizione e corrente, ingrandimento

L'andamento irregolare della corrente mette in evidenza un problema difficile da individuare dall'analisi del solo andamento della posizione: quello della risoluzione di lettura dei segnali CAN. Infatti, la posizione viene visualizzata con risoluzione di 1% (fig. 5.21) mentre la corrente con risoluzione di 2.625A (fig. 5.22), entrambe troppo basse per l'applicazione del controllo *park-lock*.

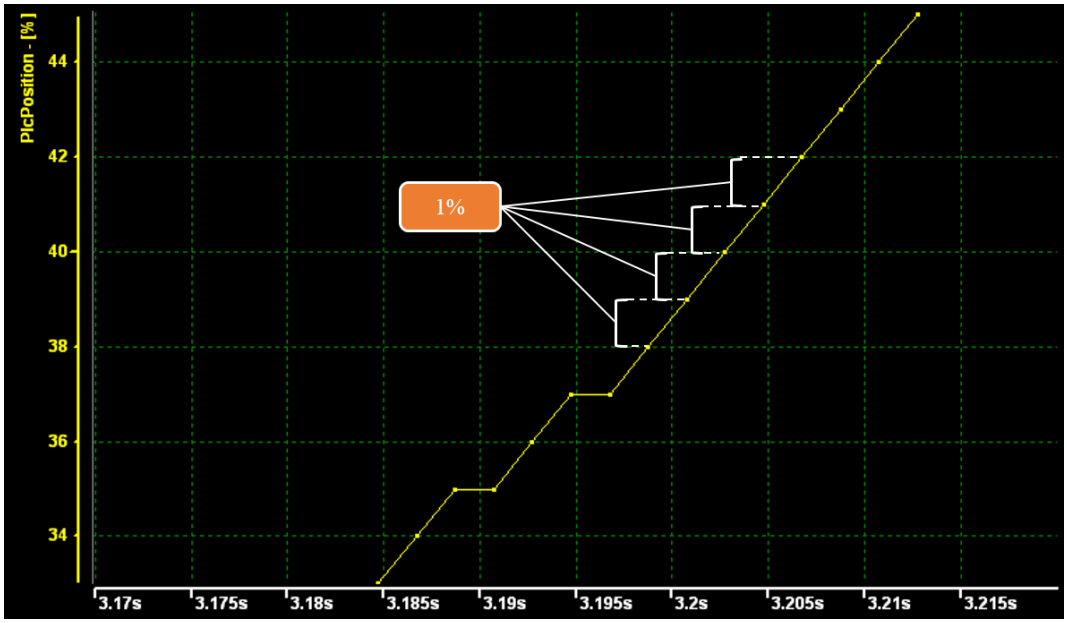


Fig. 5.21: risoluzione originale di lettura del duty cycle di posizione, pari a 1%

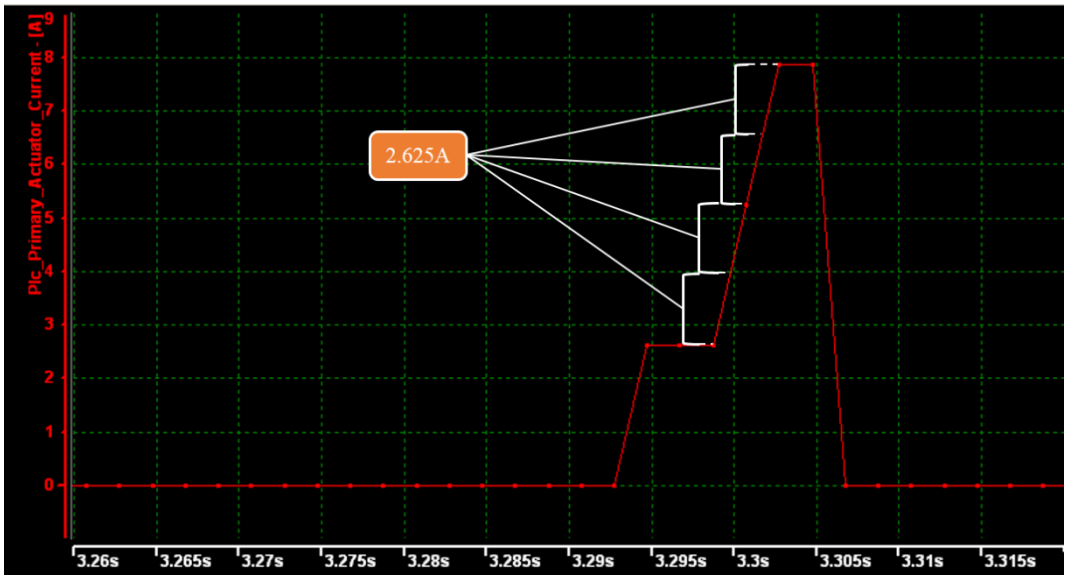


Fig. 5.22: risoluzione originale di lettura della corrente, pari a 2.625A

Il motivo della ridotta risoluzione dei due segnali è dovuto al fatto che questi vengono trasmessi dal *core* nel *Data Frame* di uscita come numeri interi e non come numeri razionali con una certa quantità di cifre significative. In particolare, entrambe le variabili sono passate come *unsigned int*, ossia interi senza segno, perché il *duty cycle* può essere solo positivo mentre la corrente è considerata positiva indipendentemente dal verso di percorrenza (in una singola fase viene consentito il passaggio della corrente in un unico verso).

Come già sottolineato nella descrizione del progetto *CANape*, uno delle funzioni di un database file è di definire i fattori di scala che permettono di passare dai valori interni indicati dalle sequenze di bit del *Data Frame* ai valori in unità fisiche, così da poter realizzare dei grafici utili alla comprensione dei processi che si desidera controllare. La figura 5.23 riporta alcune informazioni contenute nel database originale alla base della creazione del progetto sul software *CANape*.

Name	Star...	Len...	Factor	Offset	Mini...	Maxi...	Unit
PlcStatusByte	0	8	1	0	0	2	
Plc_DiagByte	8	8	1	0	0	0	
PlcPosition	24	16	1	0	0	100	%
Plc_Primary_...	42	14	2.625	0	0	15	A
Plc_Seconda...	62	12	2.625	0	0	15	A

Fig. 5.23: fattori di scala originali indicati nel database

La quarta colonna da sinistra riporta i fattori di scala per i cinque segnali del messaggio di ingresso e si osserva che quello per la posizione del *park-lock* è pari ad 1 mentre per le due correnti vale 2.625. Quindi, le istruzioni del database lasciano inalterato il valore originale della posizione, che rimane intero e quindi con risoluzione di 1%, mentre il valore intero di corrente viene convertito in un multiplo di 2.625A, che ne rappresenterà anche la risoluzione di visualizzazione.

Per risolvere il problema, si è deciso di operare una doppia modifica, sia sul database del progetto che sul sorgente del *core* che definisce i valori di posizione e correnti da inserire nel messaggio CAN, allo scopo di ottenere per tutti e tre i segnali una risoluzione di 100 volte migliore (quindi più piccola) rispetto a quella di partenza. Per prima cosa, l'immagine successiva espone le righe di codice di *parklock.c* nel quale viene determinato il contenuto del *Data Frame* di uscita.

```
siTmp = (uint8_t)((tmpsensor_position * 0.1286) / 5.00);
tmpprimary_current = (uint8_t)(tmpprimary_current * 0.032258);
tmpsecondary_current = (uint8_t)(tmpsecondary_current * 0.032258);
```

Fig. 5.24: codice con i calcoli originali dei valori da inserire nel Data Frame di uscita

La prima espressione calcola il valore del *dc* di posizione, la seconda quella della corrente primaria e la terza quella della corrente secondaria. Le variabili *tmpsensor_position*, *tmpprimary_current* e *tmpsecondary_current* sono dichiarate sempre in *parklock.c* come *int16_t*, ossia come interi con segno a 16 bit (il MSB porta il segno mentre gli altri il modulo). Ciascuna delle variabili temporanee è coinvolta in un calcolo con uno o più numeri razionali, il cui risultato viene salvato in una variabile intermedia di tipo *float*, non dichiarata; quindi, viene effettuato il *cast* (ossia una conversione di tipo) di ciascun risultato nel tipo intero senza segno a 8 bit (*uint8_t*) e i valori ottenuti vengono subito dopo inseriti nel *payload* (la sequenza di bit utili) del messaggio CAN che verrà trasmesso in uscita.

Per ottenere il perfezionamento della risoluzione, si è deciso di moltiplicare per 100 (fig. 5.25) il valore di ogni grandezza a monte del *cast* ad *unsigned int*; in questo modo, le informazioni delle prime due cifre decimali possedute dal risultato *float* intermedio non vengono perse nel troncamento della parte decimale effettuato dal *cast* ma sono trasferite nel numero intero ad 8 bit.

```
siTmp = (uint8_t)((tmpsensor_position * 0.1286) / 5.00 * 100.0);
tmpprimary_current = (uint8_t)(tmpprimary_current * 0.032258 * 100.0);
tmpsecondary_current = (uint8_t)(tmpsecondary_current * 0.032258 * 100.0);
```

Fig. 5.25: codice con i calcoli modificati dei valori da inserire nel Data Frame di uscita

Naturalmente, per lavorare su *CANape* con valori aventi significato fisico e non 100 volte più grandi della realtà è stato necessario correggere il file *.dbc*, riducendo il valore dei fattori di scala di un fattore pari a 100 (fig. 5.26).

Name	Star...	Len...	Factor	Offset	Mini...	Maxi...	Unit
PlcStatusByte	0	8	1	0	0	2	
Plc_DiagByte	8	8	1	0	0	0	
PlcPosition	24	16	0.01	0	0	100	%
Plc_Primary_...	42	14	0.02625	0	0	15	A
Plc_Seconda...	62	12	0.02625	0	0	15	A

Fig. 5.26: fattori di scala modificati indicati nel database

In questo modo, la risoluzione di visualizzazione del *duty cycle* di posizione è diventata dello 0.01%, mentre quella della corrente di 0.02625A; l'effetto di queste modifiche sui grafici di posizione e di corrente si possono osservare nella figura che segue.

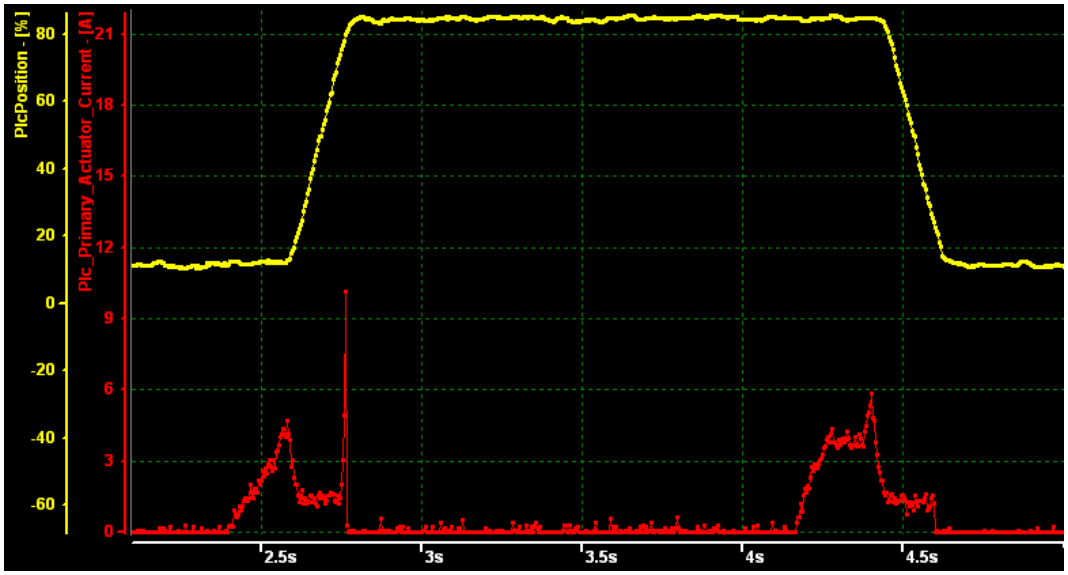


Fig. 5.27: miglioramento della risoluzione di lettura di posizione e corrente

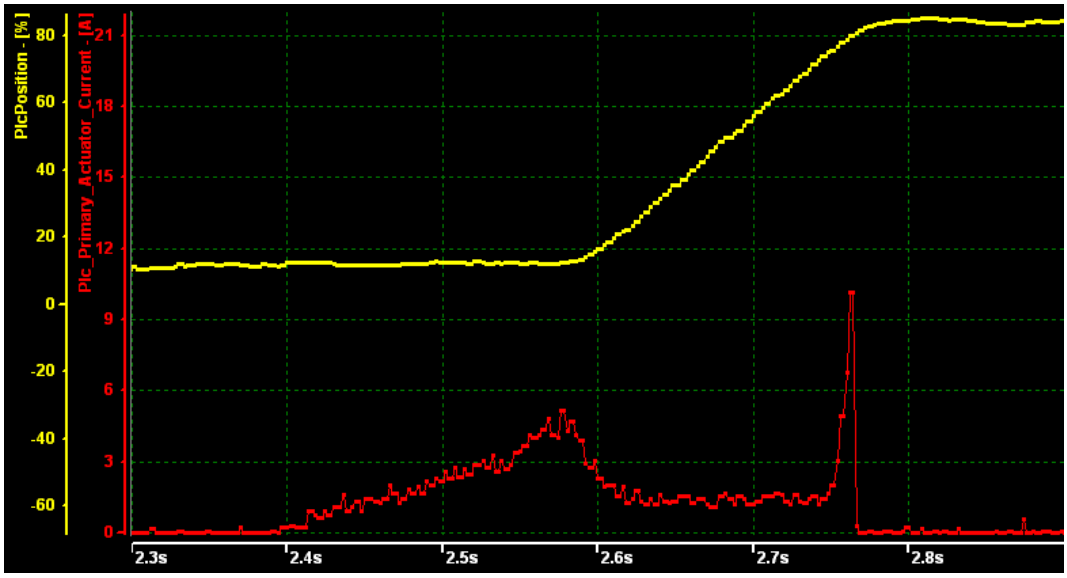


Fig. 5.28: miglioramento della risoluzione di lettura di posizione e corrente, ingrandimento

5.4 Taratura delle misure di posizione e corrente

Come affermato nell'introduzione di questo capitolo, i sorgenti e il progetto *CANape* sono stati modificati al fine di configurare adeguatamente un sistema per l'esecuzione controllata di prove a fatica coinvolgenti dispositivi *park-lock*. Le prove in questione, però, non dovevano essere applicate delle leggi di *engage* e *disengage* casuali ma mi è stato richiesto di modificare i programmi in modo da far compiere al dispositivo dei movimenti con caratteristiche specifiche; in particolare, ho dovuto tarare l'algoritmo di controllo attuato dal *driver* così da ottenere un andamento del *dc* di posizione molto simile a quello determinabile attraverso l'utilizzo di una terza centralina a disposizione dell'azienda e diversa da quelle con cui io ho avuto a che fare. Questa ECU è stata programmata da terzi per l'esecuzione di prove a fatica ed è interfacciata ad un banco prova (oltre che al *park-lock*) mediante un progetto *CANape* adeguato, che permette addirittura di fissare i parametri di controllo del *park-lock* attraverso la GUI del progetto.

Prima di tentare la riproduzione degli andamenti di *dc* di posizione prodotti dalla terza centralina, è stato fondamentale eseguire la taratura delle letture di posizione e di corrente restituite dal mio progetto *CANape*. Infatti, quando vi è la necessità di confrontare due andamenti visualizzati con due centraline e due progetti

CANape con caratteristiche differenti, è necessario verificare e garantire che, a fronte dello stesso valore reale della grandezza fisica esaminata, i due sistemi restituiscano lo stesso valore o comunque due numeri molto simili tra loro [37-38]. Senza le operazioni di taratura, tutte le considerazioni realizzate sulla base del confronto tra grafici e/o dati prodotti dai due sistemi di lettura non sono affidabili e, chiaramente, non è possibile riconoscere gli effetti positivi o negativi delle modifiche apportate all'algoritmo di controllo del motore.

I valori che si possono leggere mediante il progetto *CANape* della centralina analizzata derivano dal *Data Frame* CAN di uscita dal dispositivo; è quindi facile intuire che la taratura di cui si è appena disquisito porterà modifiche unicamente ai sorgenti del *core* che riguardano la determinazione del contenuto dei messaggi CAN e, in particolare, di *parklock.c*. Occorre evidenziare che, però, non è stata verificata solo la corrispondenza tra i valori restituiti dai due progetti di interfaccia: il tipo di applicazione, le caratteristiche delle strategie di controllo e le richieste che ho dovuto soddisfare hanno reso necessaria anche la calibrazione dei valori di posizione e corrente letti ed utilizzati dalle schede *driver*. Come si vedrà nel dettaglio più avanti, l'algoritmo di controllo originale applicato dal *driver* MAIN prevede due anelli chiusi: uno di posizione e un secondo di corrente posto più all'interno. In generale, i *loop* per il controllo di una grandezza come questi prevedono (a tempo continuo o discreto) il confronto tra:

- Un valore di riferimento (*set point* o set), che rappresenta il valore che la grandezza dovrebbe presentare in quel momento.
- Un valore retroazionato (feedback), che deriva da una catena di misura e corrisponde al valore attuale e reale della grandezza.

Affinché il confronto abbia un senso, i due termini devono essere sulla stessa scala di misura, ossia lo stesso valore della grandezza in questione deve essere rappresentato in modo analogo sia se è introdotto nel *loop* come riferimento sia come valore attuale. La definizione del *set point* istantaneo di un anello di controllo può avvenire in modo diretto o indiretto, ma in ogni caso è sempre necessario conoscere la relazione tra il valore indicato (che può essere in una scala arbitraria,

purché corrispondente a quella del feedback) e quello 'fisico' che si vuole raggiungere. Pertanto, sia per il *dc* di posizione che per la corrente, si è proceduto all'identificazione della caratteristica statica che correla il valore nella scala usata dal *driver* al valore ingegneristico.

Riassumendo, sono state realizzate sperimentalmente due attività di taratura distinte per ciascuna delle due grandezze coinvolte in anelli di controllo:

1) Taratura dei valori restituiti dai grafici di *CANape*.
2) Taratura dei valori usati dai *loop* di controllo applicati dai *driver*.

La taratura ha riguardato il *duty cycle* di posizione e la corrente e non altre variabili fisiche (velocità, tensione, etc.) per due semplici motivi:

1) Perché sono le uniche due grandezze che entrambe le centraline leggono dall'esterno tramite sensori (ad effetto Hall per la posizione e a resistore per la corrente) e, se trasmesse mediante messaggio CAN, non provengono da derivazione numerica o non sono imposte in fase di controllo: si tratta delle due grandezze principali che descrivono il comportamento del carico.
2) Perché sugli andamenti di queste grandezze sono state fatte delle richieste specifiche, volte all'allineamento tra quanto può realizzare la terza centralina e quanto si riesce ad ottenere con una delle due centraline riprogrammate.

La taratura della corrente è stata effettuata perché ho dovuto implementare nell'algoritmo del *driver* sia l'esecuzione di un breve gradino di 10A in corrispondenza di ciascun finecorsa, sia la limitazione della corrente applicata al carico di 10A durante tutto il corso di funzionamento; le motivazioni delle richieste sulla corrente verranno delucidate in seguito.

Taratura della lettura di posizione

Le operazioni di taratura riguardanti la posizione sono state sviluppate per ottenere dei movimenti in fase di *engage* e di *disengage* con caratteristiche

particolari. In particolare la richiesta che ho dovuto soddisfare è stata la seguente: a fronte dell'utilizzo dello stesso *park-lock*, l'andamento di *duty cycle* determinato dalla centralina nuova (indicata da ora in poi con la lettera "N") deve essere uguale a quello conseguito mediante l'uso della centralina di riferimento (indicata con la lettera "R"). Le due ECU vengono confrontate nell'azione sullo stesso *park-lock*; ciò indica l'uguaglianza non solo delle parti meccaniche di base (alberino, camma, molla, etc.) ma anche del sensore e della MGU; in particolare, il fatto che il sensore sia lo stesso svincola la calibrazione dal funzionamento del trasduttore: esso, infatti, non è parte del sistema di lettura ma dell'ambiente esterno alla ECU. In secondo luogo, è opportuno sottolineare che la legge del moto da riprodurre è data in *dc* e non in angolo di rotazione; se non fosse stato così sarebbe stato necessario determinare anche la caratteristica statica del sensore per associare il grafico del *duty cycle* all'andamento dell'angolo nel tempo.

La taratura della lettura di posizione ha avuto come scopo la verifica della corrispondenza tra i *dc* resi dalle due centraline e dai due relativi progetti *CANape* e, in caso di mancanza di questa corrispondenza, di modificare il sistema di lettura nuovo (quello della ECU "N") per ottenere il risultato voluto. Non è stato possibile confrontare i *dc* in modo diretto, ossia di fornire l'output del sistema "R" come input del sistema "N" per poi paragonare le uscite, ma si è ricorso ad una sorgente di segnali PWM considerata 'ideale', ovvero capace di produrre un treno di impulsi pwm con caratteristiche note. Quindi, il segnale generato con *dc* noto è stato letto prima con una centralina e poi con l'altra, per poi confrontare le due letture e controllare che il valore della ECU N corrispondesse a quello della ECU "R"; naturalmente, l'operazione è stata ripetuta per un numero di punti ritenuto sufficiente all'interno del massimo campo di lavoro delle centraline, ossia 0-100%.

Taratura della lettura di posizione su CANape

Vengono quindi presentate nel modo più sintetico e compatto possibile le fasi affrontate per tarare i valori di *duty cycle* riportati dal progetto *CANape* della centralina N rispetto a quelli presentati dal progetto *CANape* della centralina R.

Per prima cosa, è stato necessario identificare un dispositivo elettronico capace di generare segnali PWM con *duty cycle* noto: per realizzare questo scopo ho utilizzato una scheda programmabile *Arduino*-clone (fig. 5.29).

Fig. 5.29: scheda Arduino-clone

Quindi, servendosi dell'IDE specifico per *Arduino* è stato scritto un programma molto semplice che, una volta caricato sulla scheda dal PC mediante cavo USB, determina la generazione di un segnale PWM con *dc* che varia ciclicamente a gradini tra un minimo e un massimo. Per quanto riguarda lo step di crescita del segnale, occorre specificare che l'*Arduino*-clone non può far variare il *dc* con risoluzione elevata; infatti, il valore del *duty cycle* viene indicato nello script come numero da 0 a 255 e in uscita si ritrova, secondo una scala lineare, un valore reale tra 0 e 100: la risoluzione di produzione del *duty cycle* è quindi:

$$risoluzione\ dc\ prodotto = \frac{100\ \%}{255} \cong 0.392\ \% \qquad (5.4)$$

Per non dover considerare un numero eccessivo di punti di taratura, il *dc* è stato fatto variare in scala 0-255 con gradini di ampiezza 10 tra i valori 10 e 250 (estremi inclusi): in uscita il *dc* del PWM reale cresce, dunque, con gradini di ampiezza 3.92% tra gli estremi 3.92% e 98%, per un totale di 25 valori di riferimento. Siccome il segnale prodotto della scheda deve avere le stesse caratteristiche di quello del sensore montato sul prototipo, esso viene generato sul pin digitale numero 6, in quanto è configurato a livello hardware per produrre segnali PWM con portante a 1000Hz.

A questo punto, è stato necessario leggere e memorizzare il segnale prodotto dalla scheda prima con il sistema di riferimento ("R") e poi con quello nuovo ("N"). In entrambi i casi, la scheda è stata alimentata con una batteria esterna mentre l'interfacciamento con la catena di lettura è stato effettuato mediante il connettore del sensore, mettendo in comune le masse (*GND*) e collegando il pin digitale numero 6 con l'ingresso di lettura del PWM del connettore (fig. 5.30).

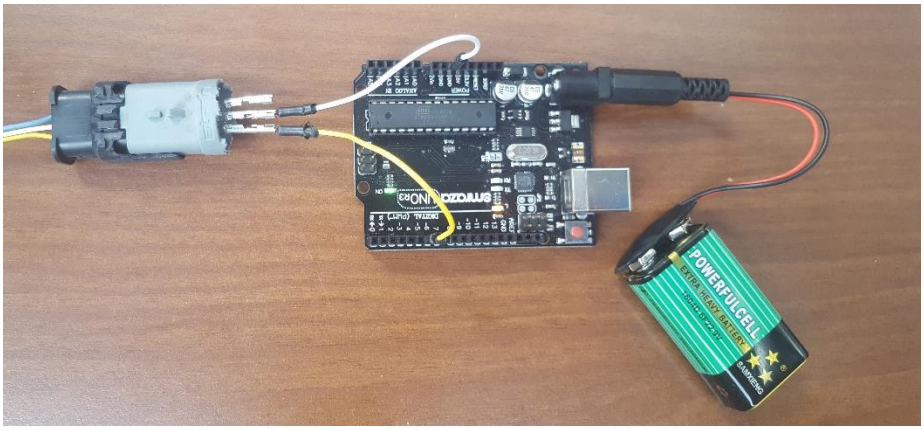


Fig. 5.30: collegamento tra scheda e connettore del sensore posizione lato centralina

La figura 5.31 rappresenta l'andamento del *duty cycle* di posizione rilevato dal sistema centralina con progetto *CANape* nuovo mentre la figura 5.32 mostra l'andamento rilevato con il sistema di riferimento.

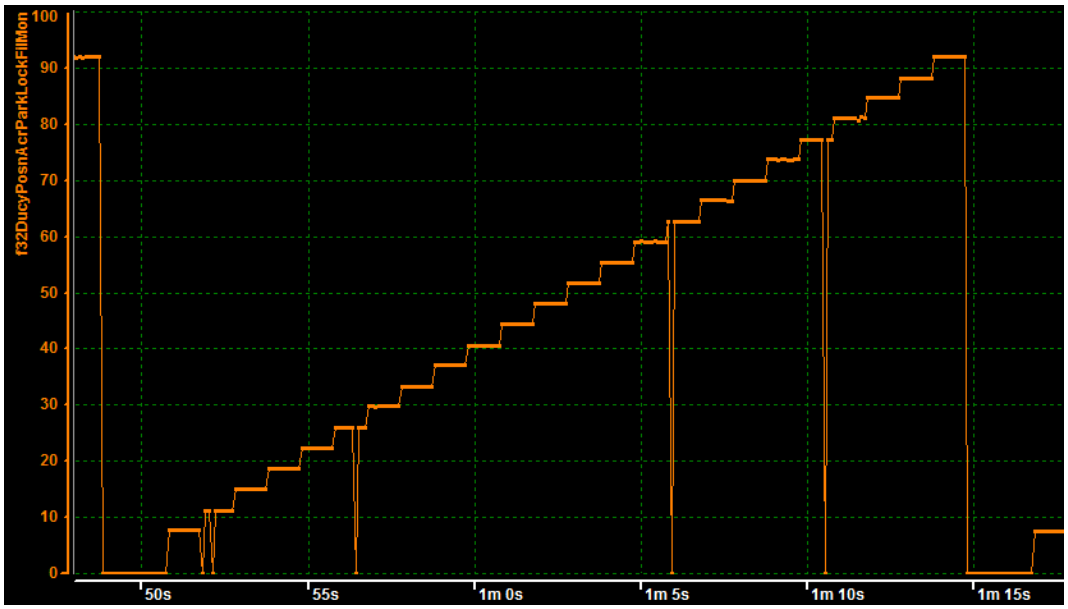


Fig. 5.31: lettura dei gradini di PWM con sistema "R"

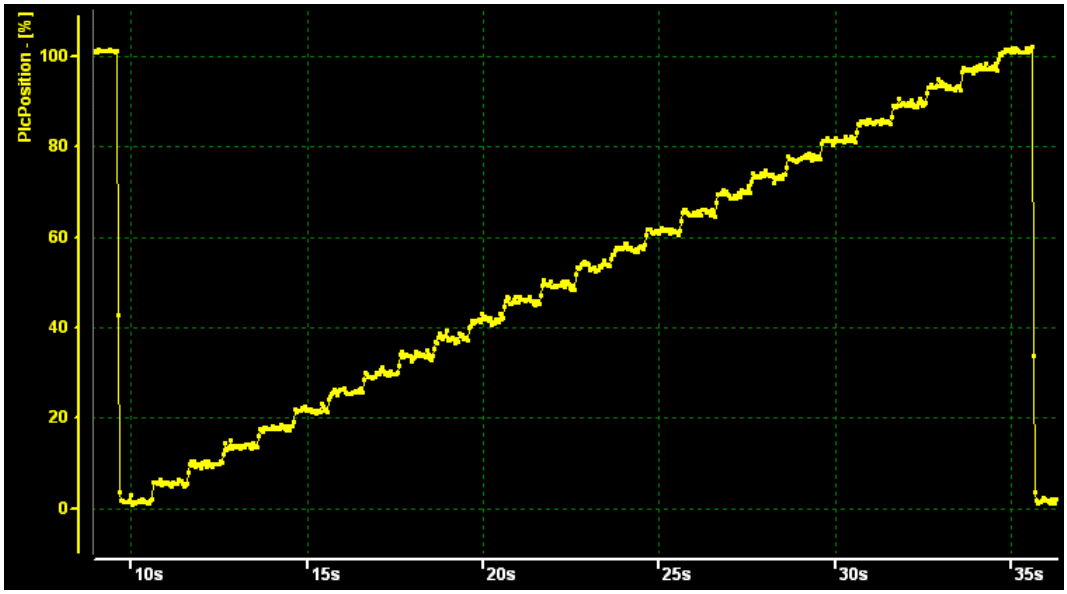


Fig. 5.32: lettura dei gradini di PWM con sistema "N"

Quindi, per entrambe le letture si è costruita su *Excel* la tabella di correlazione PWM reale – PWM letto medio ed è stato tracciato il grafico di confronto tra i due set di misure sui 25 punti considerati. L'andamento di figura 5.31 riporta dei valori già filtrati e mediati quadraticamente, quindi ogni gradino è caratterizzato da un valore di *dc* costante che è stato inserito direttamente nella relativa tabella; invece, il grafico di figura 5.32 riporta dei valori oscillanti e quindi si è proceduto diversamente: per ogni gradino i dati della porzione centrale sono stati importati da *CANape* ad *Excel*, ne è stata fatta la media aritmetica e il numero ottenuto è stato registrato nella tabella. Le figure che seguono presentano le tabelle e il grafici appena nominati.

Valori reali di *dc* [%]	Valori medi di *dc* letti con sistema "R" [%]
3,92	7,75
7,84	11,27
11,76	15,02
15,69	18,78
19,61	22,30
23,53	26,06
27,45	29,81
31,37	33,33
35,29	37,09
39,22	40,61
43,14	44,37

47,06	48,12
50,98	51,64
54,90	55,40
58,82	58,92
62,75	62,68
66,67	66,43
70,59	69,95
74,51	73,71
78,43	77,23
82,35	80,99
86,27	84,74
90,20	88,26
94,12	92,02
98,04	0,00

Tab. 5.2: tabella dc letti per sistema "R"

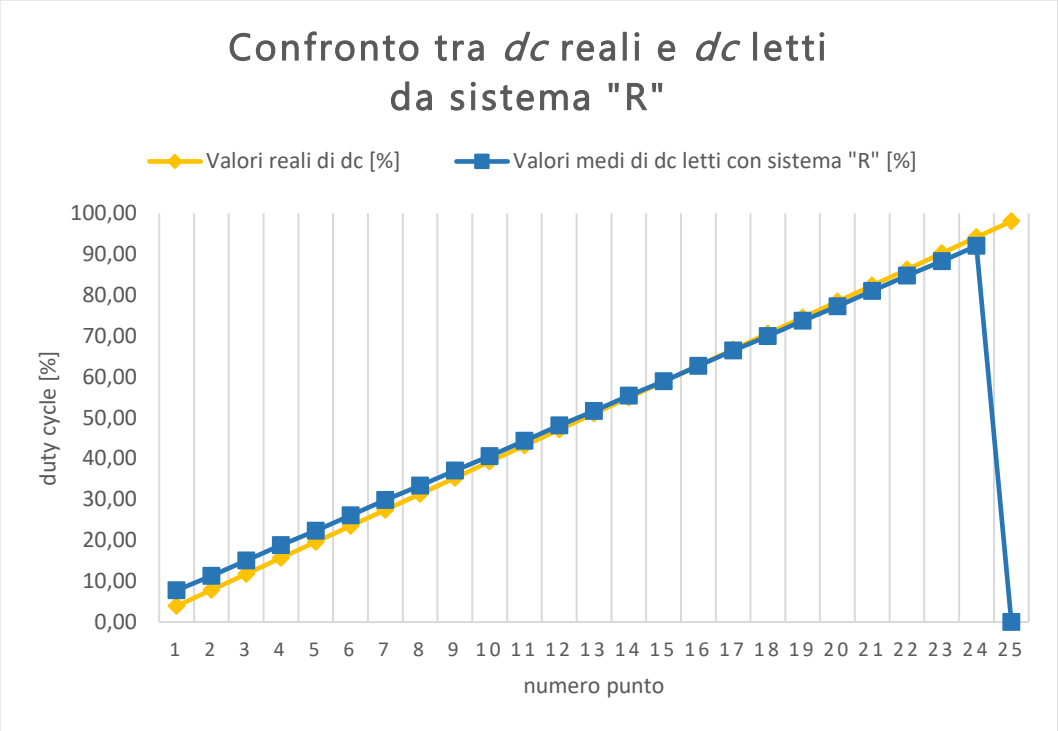


Fig. 5.33: grafico di confronto sulla lettura del dc di posizione per sistema "R"

Valori reali di *dc* [%]	Valori medi di *dc* letti da sistema "N" [%]
3,92	5,86
7,84	10,32
11,76	13,88
15,69	17,89

19,61	22,14
23,53	25,98
27,45	29,89
31,37	33,79
35,29	37,80
39,22	41,69
43,14	45,93
47,06	49,56
50,98	53,72
54,90	57,56
58,82	61,52
62,75	65,61
66,67	69,62
70,59	73,58
74,51	77,66
78,43	81,56
82,35	85,59
86,27	89,48
90,20	93,50
94,12	97,46
98,04	101,20

Tab. 5.3: tabella dc letti per sistema "N"

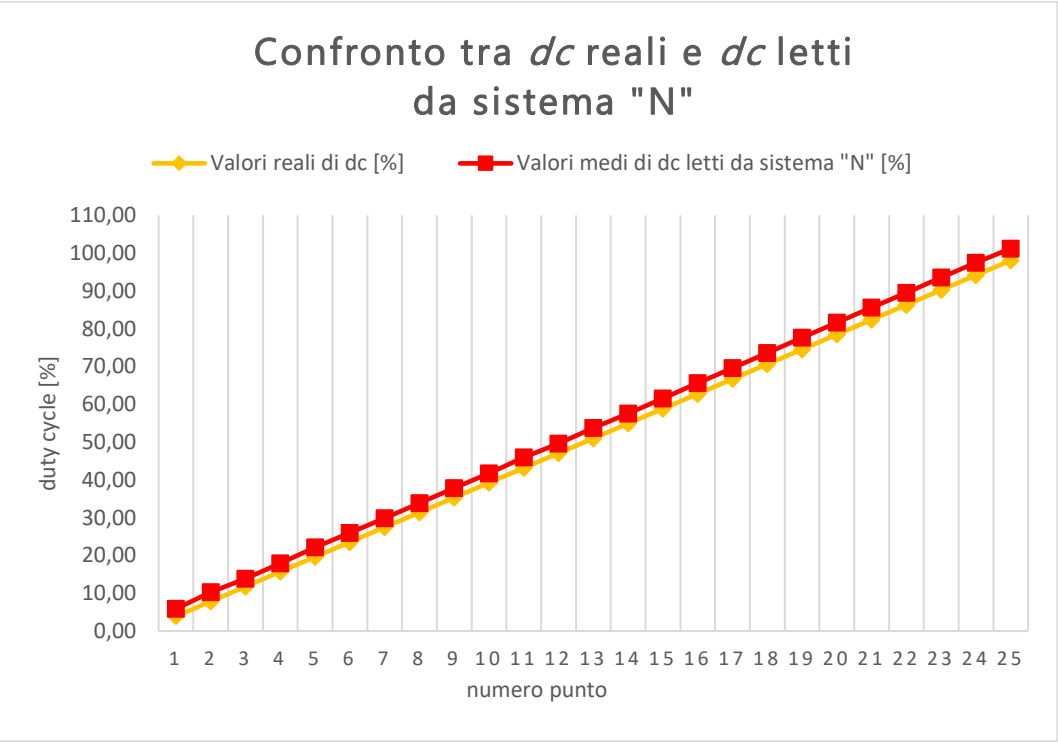


Fig. 5.34: grafico di confronto sulla lettura del dc di posizione per sistema "N"

Relativamente alla figura 5.32, si osserva che la ECU di riferimento produce un *dc* nullo in corrispondenza del 25-esimo punto di confronto; questo perché è programmata per restituire un valore nullo quando il *duty cycle* del PWM di ingresso supera il 95%. Siccome in condizioni di normale funzionamento si prevede che il PWM prodotto dal sensore Hall non superi il 90%, si è deciso di escludere dalle successive operazioni di taratura l'ultimo *dc* di riferimento. La figura seguente rappresenta sullo stesso piano gli andamenti dell'errore assoluto di *duty cycle* (letto vs vero) per le due centraline.

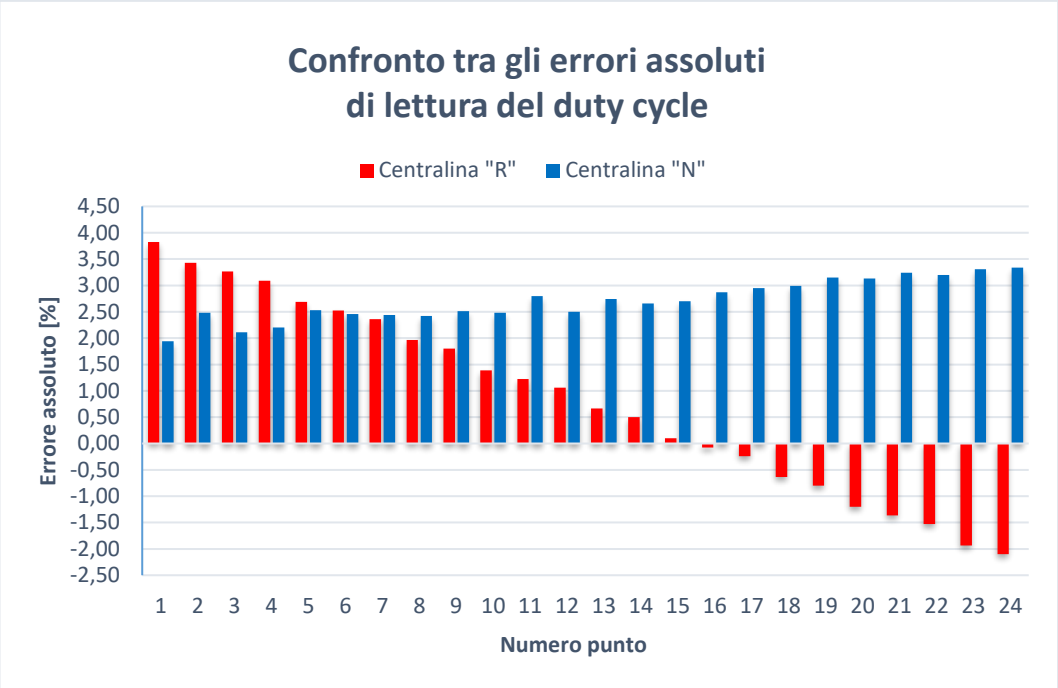


Fig. 5.35: istogramma degli errori assoluti di lettura del dc di posizione

Si osserva che l'errore assoluto si mantiene pressoché costante per la ECU nuova mentre per la ECU vecchia varia in modo lineare da un valore positivo a uno negativo.

L'immagine successiva rappresenta il confronto tra i valori restituiti dai due sistemi sui 24 punti di taratura.

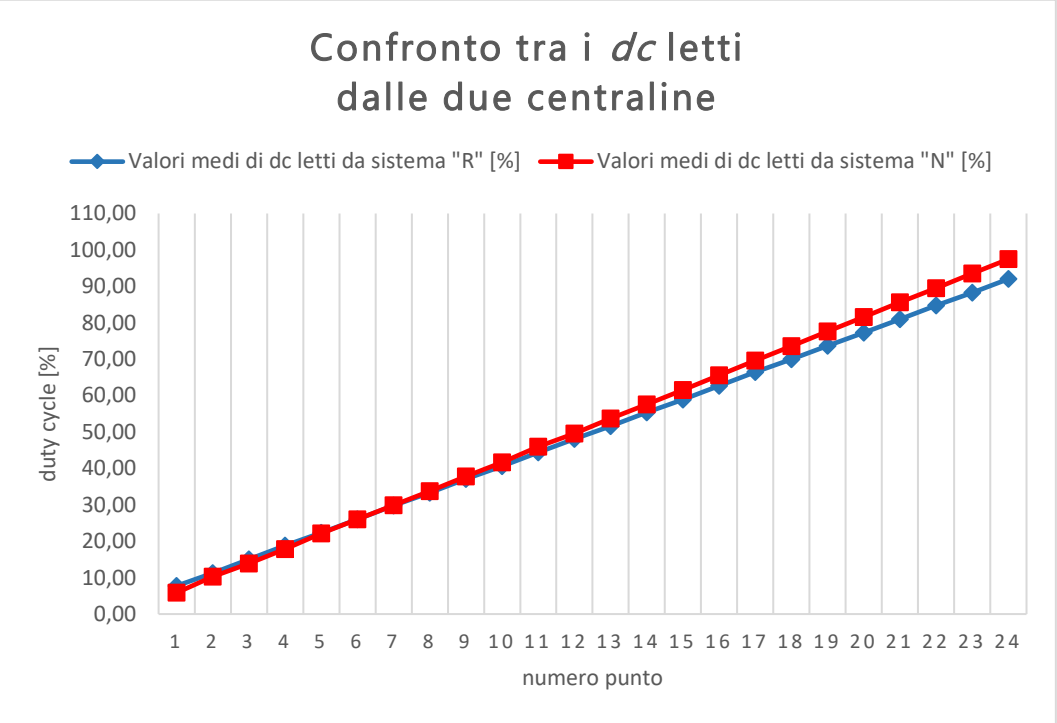


Fig. 5.36: grafico di confronto delle letture di dc di posizione tra i due sistemi

Il grafico di confronto mostra che, a parità di PWM in ingresso, i due sistemi fanno leggere due differenti *duty cycle*: ciò indica che la taratura era necessaria e richiede delle modifiche alla catena di lettura della centralina nuova. Per compensare la discrepanza tra le due letture sono state considerate due strade:

(1) Modificare il progetto *CANape*, ossia l'elaborazione del contenuto del messaggio CAN in ingresso.
(2) Modificare il valore del segnale prima della costruzione del messaggio CAN e della sua trasmissione.

Si è deciso di attuare la seconda possibilità, ritenendo più pratico realizzare la taratura mediante due sorgenti diversi per i *core* delle due centraline nuove, piuttosto che mediante due database *CANape* differenti. Il valore della posizione da incorporare nel *Data Frame* di uscita della ECU "N" è definito nel *task* di controllo del *park-lock*; il calcolo di tale valore è realizzato nel sorgente *parklock.c* ed è riportato in figura 5.37.

```
int16_t siTmp;

siTmp = (int)((tmpsensor_position * 0.1286) / 5.00 * 100.0);

*payload = (uint8_t)(siTmp >> 8);
payload++;
*payload = (uint8_t)(siTmp);
```

Fig. 5.37: calcolo originale del duty cycle di posizione da trasmettere via CAN

La prima riga (che è stata avvicinata al resto per comodità) definisce *siTmp*, variabile intera con segno a 16 bit in cui viene memorizzata la posizione intera da inserire nel messaggio. Nella seconda riga *siTmp* viene valutato mediante un *cast* ad intero del risultato di un calcolo a numeri reali che coinvolge la variabile di tipo intero *tmpsensor_position* (quella in cui è stato salvato il valore del *duty cycle* di posizione calcolato come media dei valori del relativo *buffer*); si ricorda che la moltiplicazione per 100 serve a migliorare la risoluzione di lettura su *CANape*. Nelle ultime righe, *siTmp* (16 bit) viene memorizzato nel *payload* del messaggio suddividendolo in due byte (8 bit ciascuno). Si osserva che il *cast* a *unsigned int* che avviene per le due metà della variabile è necessario per evitare che nella seconda metà il MSB sia associato ad un informazione sul segno.

In riferimento alla figura 5.37, dunque, il calcolo del *duty cycle* di posizione, al netto di altre operazioni secondarie è il seguente:

$$duty\ cycle = (tmpsensor\_position * 0.1286)/\ 5.00 \qquad (5.5)$$

Allo scopo di tarare la lettura CAN del sistema nuovo, il numero restituito dalla formula appena presentata deve essere corretto prima di venire inglobato nel *Data Frame* di uscita ed è stato necessario identificare una relazione matematica per il calcolo della correzione in funzione del *dc* di partenza. La tabella 5.4 riporta nelle tre colonne i *dc* di riferimento, i *dc* da correggere e la differenza tra i secondi e i primi; quindi, il grafico di fig. 5.38 presenta l'andamento della differenza trovata rispetto al *dc* da correggere.

Valori medi di *dc* letti da sistema "R" [%]	Valori medi di *dc* letti da sistema "N" [%]	Differenza da compensare [%]
7,75	5,86	-1,89
11,27	10,32	-0,94

15,02	13,88	-1,14
18,78	17,89	-0,89
22,30	22,14	-0,16
26,06	25,98	-0,07
29,81	29,89	0,08
33,33	33,79	0,46
37,09	37,80	0,71
40,61	41,69	1,08
44,37	45,93	1,57
48,12	49,56	1,44
51,64	53,72	2,08
55,40	57,56	2,16
58,92	61,52	2,60
62,68	65,61	2,93
66,43	69,62	3,18
69,95	73,58	3,62
73,71	77,66	3,95
77,23	81,56	4,33
80,99	85,59	4,60
84,74	89,48	4,73
88,26	93,50	5,24
92,02	97,46	5,44

Tab. 5.4: tabella per la compensazione degli errori di lettura del sistema N

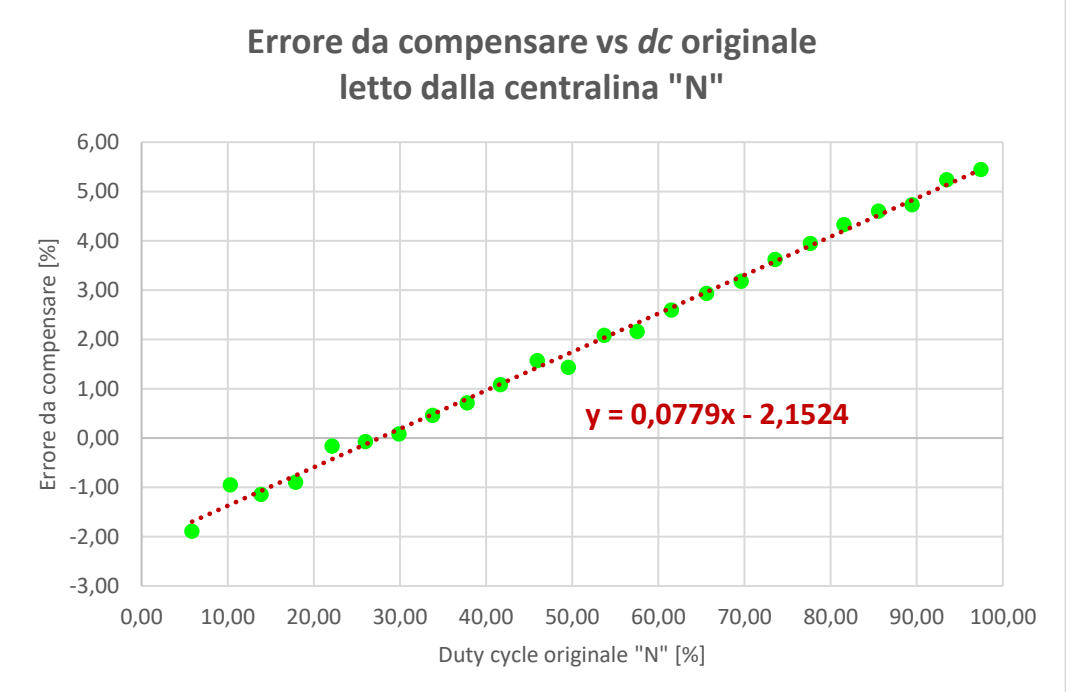


Fig. 5.38: andamento degli errori da compensare e retta interpolante

Dal grafico della figura precedente si nota che l'errore da eliminare varia in modo pressoché lineare con il *dc* originale della ECU da me gestita, i punti si dispongono approssimativamente su una retta; per questo motivo si è deciso di utilizzare *Excel* per tracciare la retta interpolante ai minimi quadrati e, soprattutto, visualizzarne l'equazione:

$$errore = 0.0779 * dc_{originale} - 2.1524 \qquad (5.6)$$

L'espressione lineare è stata utilizzata in *parklock.c* per ricavare il termine che poi è stato sottratto al *duty cycle* originale per ottenere il valore corretto da inserire nel messaggio CAN. La figura 4.3.15 riporta la parte di codice corretta, i calcoli sono inoltre spiegati dai commenti laterali.

```
int16_t siTmp;
float dc;
float correzione_dc;

dc = (tmpsensor_position * 0.1286) / 5.00;     // calcolo dc originale

correzione_dc = 0.0779 * dc - 2.1524;          // calcolo della correzione al dc

dc = dc - correzione_dc;                       // calcolo dc corretto

siTmp = (int)(dc * 100.0);                     // moltiplicazione x100 per ris e cast a intero
```

Fig. 5.39: calcolo corretto del duty cycle di posizione da trasmettere via CAN

Per implementare la correzione, si sarebbe potuto creare nel sorgente due vettori con i *dc* originali e quelli di riferimento, per poi servirsene per interpolare il *dc* di uscita noti i due *dc* originali affiancati nel vettore che includono quello ottenuto ricavato dal calcolo originale; siccome, però, la disposizione dei punti in fig. 5.38 è apparsa subito molto lineare, si è optato per la strada di approssimare la correzione esatta mediante una equazione del primo ordine.

Per osservare il risultato di questo primo step di taratura, i connettori dei due sistemi per il sensore di posizione sono stati collegati in parallelo alimentando il sensore con i 5V forniti dalla ECU nuova e mettendo in comune gli altri due fili di massa e PWM; inoltre, la MGU è stata collegata con il cablaggio di potenza al *driver* MAIN della ECU "N". A questo punto, è stato comandato l'*engage* dell'attuatore e ciascuno dei due progetti *CANape* ne ha memorizzato il *datalog*; gli andamenti registrati sono stati riportati in un unico piano per poterli confrontare e

verificare la buona riuscita della taratura. La figura 5.40 mostra il confronto, in verde si ha il *trend* registrato dalla ECU "R" e in azzurro quello memorizzato dalla ECU "N".

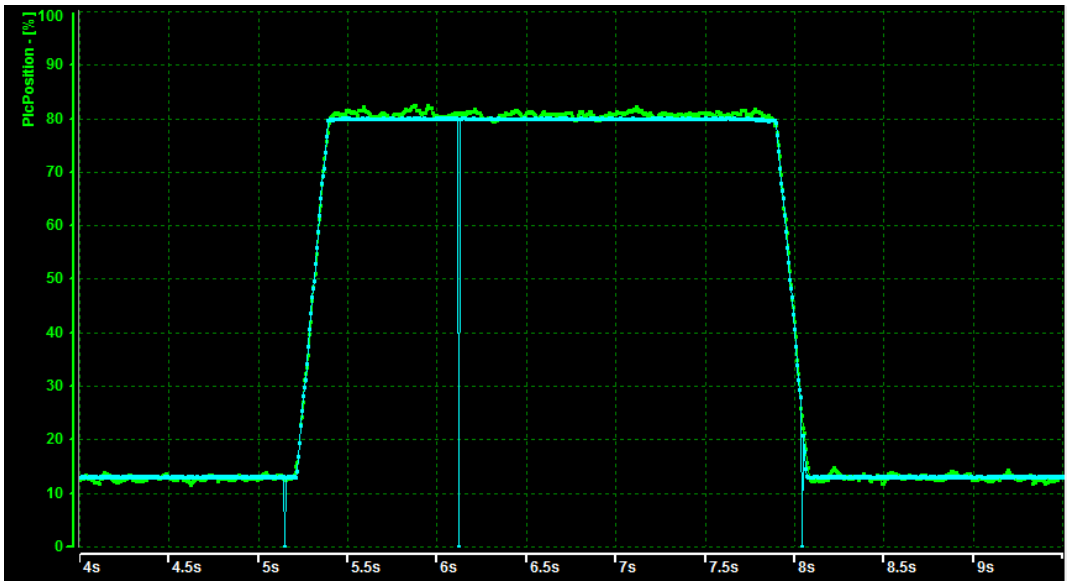


Fig. 5.40: confronto tra le letture di posizione a valle delle correzioni di taratura

L'immagine precedente fa presagire una buona corrispondenza tra le letture: le due figure successive ingrandiscono la sovrapposizione delle curve nelle fasi di *engage* e *disengage*.

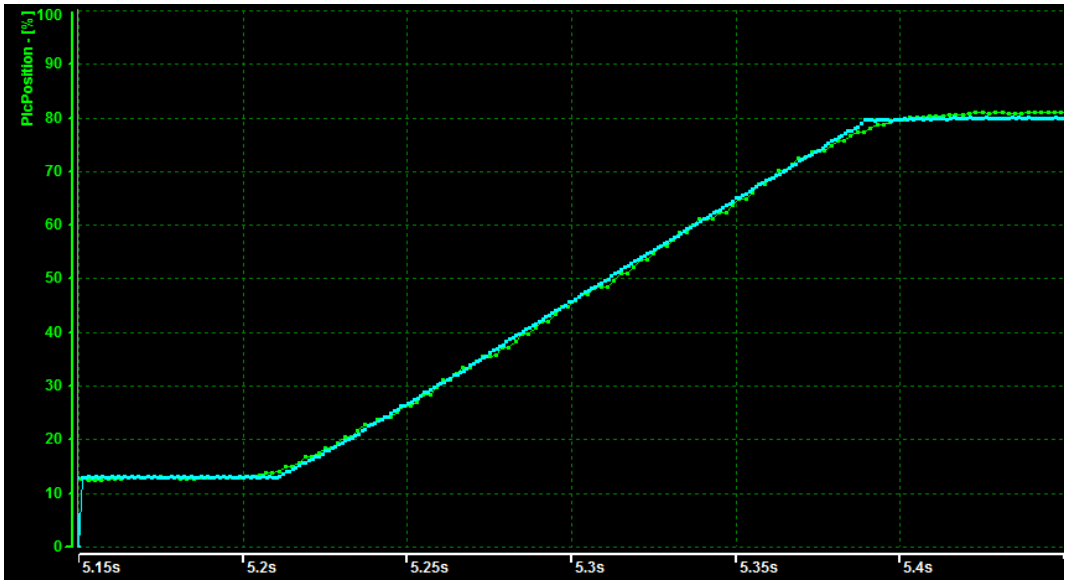


Fig. 5.41: confronto post-taratura tra le letture di posizione per fase di engage

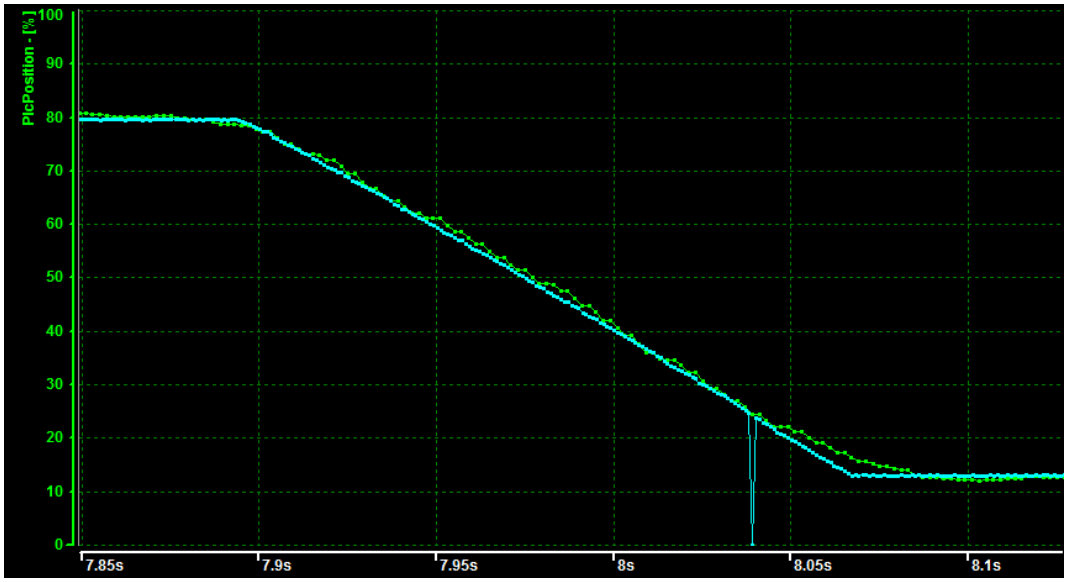


Fig. 5.42: confronto post-taratura tra le letture di posizione per fase di disengage

Entrambi gli zoom permettono di osservare un'ottima corrispondenza tra le letture di posizione. Nei tratti di movimento del dispositivo, la differenza è di uno o due punti percentuali al massimo ed è legata anche all'oscillazione intrinseca del segnale che arriva alla centralina nuova, per questo difficilmente correggibile solo via software. Nelle figure 5.41 e 5.42 appare un piccolo offset di *dc* in corrispondenza della posizione massima di innesto (leva in battuta contro il finecorsa) ma è semplicemente dovuto all'utilizzo dell'approssimazione lineare per la correzione e difatti non inficia la bontà della calibrazione.

Taratura della scala di lettura della posizione nel controllo

La riproduzione dell'andamento di posizione descritti dalla ECU di riferimento ha richiesto la comprensione e la modifica della strategia di controllo originale del *park-lock*; ciò ha determinato la necessità di saper correlare il valore della posizione che viene memorizzato dal *driver* al valore che si legge dal progetto *CANape* della nuova centralina. Nel progetto sorgente del *driver*, la posizione media viene assegnata alla variabile *POS_AVG_VAL*, la quale deriva da una media mobile fatta sul contenuto di un *buffer* che raccoglie i numeri calcolati da un convertitore A/D. Si nota che ogni scheda della centralina riceve su un apposito pin il valore analogico 0-3.3V generato dalla scheda di condizionamento e procede in modo autonomo alla conversione A/D (a 10 bit per i *driver* e a 12 bit per il *core*);

quindi, mentre le correnti sono misurate dai singoli *driver* e il loro valore è trasmesso al *core* via I2C, la posizione è letta da tutte le schede in modo indipendente.

Per questa seconda taratura relativa alla posizione è stato di nuovo utilizzato il clone *Arduino*, per trasmettere al connettore della ECU nuova un segnale PWM a 1000Hz con *duty cycle* crescente a gradini (range 10-240 con passo 10) come quello visto in precedenza; l'unica differenza è che l'ampiezza temporale di ogni gradino è stata modificata da 1s a 15s, per facilitare le rilevazioni nominate in seguito. Quindi, il PC è stato collegato al *driver* MAIN sulla porta JTAG/SMD mediante la sonda di programmazione MSP-FET della *Texas Instruments* (fornita insieme alla centralina); in questo modo, è stato possibile avviare il *driver* in modalità debug, caricando l'eseguibile del progetto sorgente originale sulla memoria *flash* della sonda invece che su quella del *driver*. Il funzionamento in debug è interessante perché permette, ad esempio, di seguire l'esecuzione di un sorgente un'istruzione alla volta e di controllare i valori assunti da una o più variabili di interesse (*watch variables*). In modalità debug l'esecuzione del programma è stata interrotta due volte per ogni gradino e i valori di *POS_AVG_VAL* corrispondenti sono stati inseriti in una tabella *Excel* (tab. 5.6); successivamente, è stata calcolata la media di ogni coppia di valori letti ed è stato tracciato il grafico (fig. 5.43) che lega il *POS_AVG_VAL* (asse delle ordinate) al *dc* letto su *CANape* (asse delle ascisse).

Valori medi di *dc* letti da *CANape* [%]	Misure *POS_AVG_VAL* [0-1023]		Media *POS_AVG_VAL* [0-1023]
7,52	51	53	52
11,64	93	94	93,5
14,92	132	138	135
18,61	173	165	169
22,54	211	210	210,5
26,09	252	247	249,5
29,69	282	284	283
33,29	326	322	324
36,99	358	365	361,5
40,58	400	401	400,5
44,49	436	436	436
47,84	487	477	482

51,68	513	518	515,5
55,22	548	562	555
58,87	591	602	596,5
62,65	630	639	634,5
66,34	671	680	675,5
69,99	714	720	717
73,76	756	753	754,5
77,36	794	795	794,5
81,08	830	826	828
84,66	869	867	868
88,38	909	902	905,5
92,03	950	945	947,5

Tab. 5.5: tabella per taratura POS_AVG_VAL

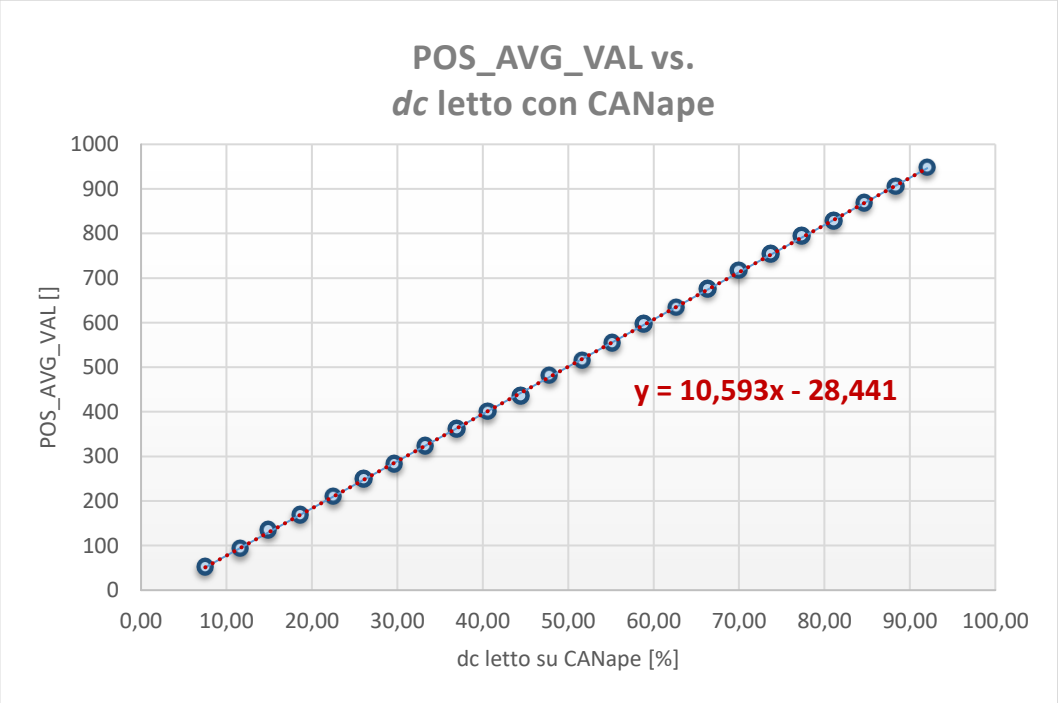


Fig. 5.43: grafico per taratura POS_AVG_VAL

L'andamento dei punti utilizzati per la caratterizzazione è perfettamente lineare e questo vuol dire che la catena di misura che associa un valore di *POS_AVG_VAL* al *dc* del segnale PWM proveniente dal sensore è praticamente priva di fonti di non linearità. Nel grafico di fig. 5.43 è stata tracciata anche la linea di tendenza lineare (in rosso), la cui equazione è la seguente:

$$POS\_AVG\_VAL = 10.593 * dc_{letto} - 28{,}441 \qquad (5.7)$$

L'espressione (1) permette, dato un valore di *duty cycle* letto su *CANape*, di conoscere il corrispondente valore memorizzato dal *driver* di *POS_AVG_VAL*; questa relazione è fondamentale perché permette di calibrare i parametri di un eventuale *loop* di controllo posizione in funzione degli andamenti di *duty cycle* che si desidera riprodurre. Inoltre, risulta molto importante saper effettuare anche la correlazione opposta, ossia essere in grado di passare da valore di *POS_AVG_VAL* a valore di *dc* di posizione; la formula da utilizzare si può ricavare direttamente da quella appena proposta ed è la seguente:

$$dc_{letto} = 0.0944 * POS\_AVG\_VAL + 2.6895 \qquad (5.8)$$

La relazione matematica (1) è stata implementata a livello software come 'macro', ossia una sorta di funzione molto semplice che esegue delle istruzioni di base in modo automatizzato quando viene richiamata, come ad esempio dei calcoli. La macro in questione (fig. 5.44) è stata definita in *config.h*, *header* del progetto dei *driver*: riceve come argomento il valore intero del *duty cycle* di posizione (che serve, ad esempio, per definire una soglia di innesto o disinnesto) e lo converte nel valore corrispondente nella scala utilizzata dal *driver* per memorizzare e fare i calcoli inerenti la posizione.

```
#define DC_TO_ADCVAL(x)                (int)(10.593 * x - 28.441)
```

Fig. 5.44: macro definita per la conversione del dc di posizione

Taratura della lettura di corrente

La taratura della corrente si è resa necessaria perché, come già anticipato, ho dovuto realizzare delle modifiche sui sorgenti dei *driver* volti ad ottenere delle particolari caratteristiche nell'andamento della corrente assorbita dal motore elettrico. In particolare, ho dovuto implementare la realizzazione di un gradino di corrente di 10A al raggiungimento dei due finecorsa e la limitazione della corrente nelle fasi di attuazione a 10A. Anche in questo caso, sono state messe in atto due procedure di caratterizzazione della lettura di corrente: una per il progetto *CANape* e una per l'anello di controllo; in questo caso, però, le due tarature sono state

eseguite in contemporanea e quindi si è deciso di non staccare in modo netto le due trattazioni.

Nel caso precedente, il riferimento per la calibrazione è stato il *dc* prodotto dal sistema di lettura "R" e non il valore reale caratteristico del segnale PWM trasmesso dal sensore o dalla scheda *Arduino*-clone; si è deciso di procedere in questo modo perché i vincoli imposti sulla legge di posizione hanno riguardato il raggiungimento della corrispondenza tra gli andamenti di *dc* ottenuti dalle due centraline e non tra quello dato dal sistema da tarare e uno ottenuto da uno strumento ideale di misura del *dc*. Per la corrente, invece, la situazione è differente: i gradini e le limitazioni da implementare mediante modifiche ai sorgenti riguardano l'effettiva corrente assorbita dagli avvolgimenti e non si tratta, quindi, di rendere la lettura della corrente effettuata dal sistema "N" uguale a quella data da "R".

Di conseguenza, entrambe le tarature della lettura di corrente hanno visto l'allineamento o la messa in relazione dei valori dati dalla GUI (*Graphic User Interface*) o memorizzati dal *driver* e quelli restituiti da uno strumento di misura di riferimento, idoneo a rilevare una corrente con precisione sufficiente per l'applicazione in esame; in particolare, in questo caso è stato utilizzato il *tester* (o multimetro) della figura successiva.

Fig. 5.45: tester impiegato per la taratura della corrente

Il tester permette di misurare, grazie a due puntali, o una differenza di tensione o una corrente: per questa applicazione è stato utilizzato, ovviamente, per il secondo tipo di misura. Questo dispositivo presenta quattro ingressi: uno è il 'comune', che

definisce un riferimento di zero (per la tensione e il segno per la corrente) per la misura che si sta per effettuare e a cui viene solitamente collegato il puntale nero, un altro ingresso è per la misura di tensione tra i due punti toccati dai puntali e gli ultimi due servono per la misura di una corrente. Ciascuna delle due linee di passaggio della corrente per la misura è caratterizzata da un fusibile, che apre il circuito quando la corrente supera un determinato valore massimo: 200mA per un fusibili e 10A per l'altro; in questo caso, i puntali sono stati collegati in corrispondenza del comune e dell'ingresso da massimo 10A di corrente. Inoltre, il *tester* presenta un selettore che rende possibile scegliere il fondo scala della misura della corrente, da cui deriva la risoluzione con cui il valore misurato è mostrato sul display: maggiore è il fondo scala e peggiore è la risoluzione (ossia più alta); per la taratura, il selettore è stato posto su 10A DC, ossia il valore più alto possibile, perché si è voluto caratterizzare il sistema sul più alto numero possibile di valori. Lo strumento di misura è stato collegato lungo la linea che va da un morsetto di potenza OUT1 del *driver* MAIN al corrispondente pin di alimentazione della MGU, così che il dispositivo fosse attraversato dalla corrente assorbita dal motore e il suo valore di corrente fosse mostrato sul display. Lo strumento è stato tarato pochi giorni prima del mio utilizzo e questo mi ha permesso di fare affidamento sui valori da questo restituiti.

Dal punto di vista pratico, la taratura è stata eseguita con l'ausilio del comando RELOAD MAIN, generabile mediante il relativo pulsante posto sulla scatola della centralina. I sorgenti originali implementati sui *driver* prevedono che, a seguito del ricevimento di tale comando, lo stato dell'applicazione (*app_status*) passi a *STATUS_RELOAD* e la macchina a stati di *app_fsm.c* dia il via al controllo del motore (funzione *motor_control*) secondo un algoritmo particolare: il *dc* della tensione di comando del motore viene calcolato mediante un unico anello chiuso di controllo della corrente, che viene impiegato per far seguire alla corrente assorbita dal motore un andamento di riferimento dalle caratteristiche precise, che ora verranno esaminate.

Le proprietà del *set point* di corrente seguito nella fase di *reload* sono identificate da quattro costanti (fig. 5.47), definite all'interno dell'*header* del *driver* di nome *config.h*.

```
#define RELOAD_SET_POINT                110
#define DEFAULT_STEP                    5
#define DEFAULT_STEP_PERIOD             10    // 1 = 2ms
#define DEFAULT_RELOAD_TIMEOUT          TIME_100MS(5)
```

Fig. 5.47: parametri originali del riferimento per il reload

La figura successiva mostra il grafico di un possibile riferimento di corrente per il *reload*, in cui vengono evidenziati i parametri visti nella figura precedente.

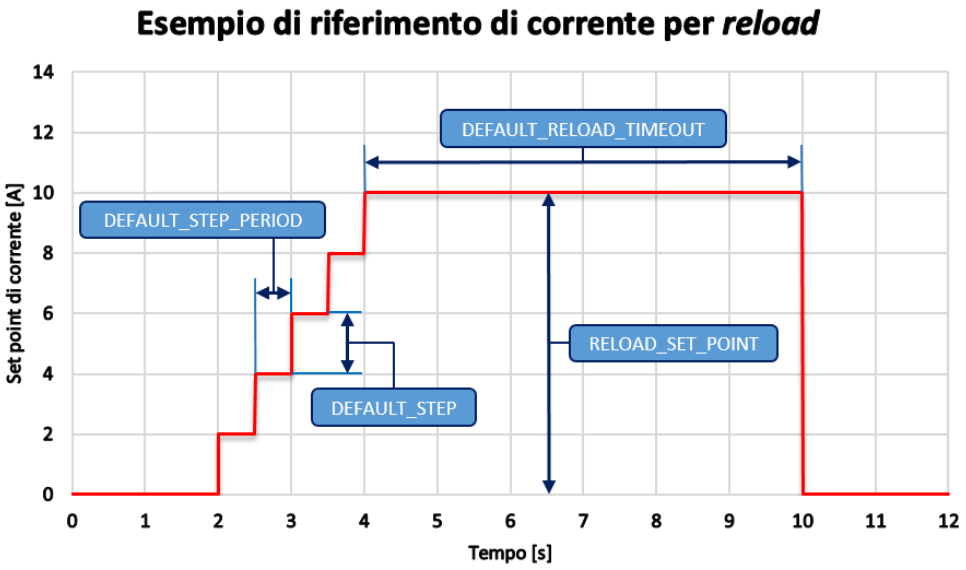


Fig. 5.48: possibile riferimento di corrente per il reload

L'andamento del riferimento per il *loop* di corrente inizia ad essere costruito quando viene trasmesso al *driver* dal *core* il comando di RELOAD MAIN; nel caso di figura 5.48 ciò avviene per Tempo = 2s. Si parte con una serie crescente di gradini di ampiezza (*DEFAULT_STEP*) e periodo (*DEFAULT_STEP_PERIOD*) costanti; nel grafico soprastante, i gradini sono da 2° e il periodo vale 0.5s. Quando il riferimento raggiunge il *RELOAD_SET_POINT* (nell'esempio 10A), la sequenza di gradini viene interrotta e il *set point* è mantenuto costante fino a quando non passa un certo intervallo di tempo (*DEFAULT_RELOAD_TIMEOUT*) dal

raggiungimento del limite: scattato il *timeout* (che qua vale 6s), il *driver* smette di controllare il motore e riferimento viene azzerato.

Siccome non è stata ancora effettuata la calibrazione della lettura di corrente, né su *CANape* né sul *driver*, non è stato possibile realizzare un grafico dalle caratteristiche concordi ai valori dei parametri indicati in *config.h*. Per quanto riguarda il timeout, la sua definizione avviene grazie all'utilizzo di una *macro* (trattasi di una sequenza di istruzioni 'condensata') di nome *TIME_100ms*: il tempo in ms fissato è uguale al numero intero fornito come argomento della *macro* moltiplicato per 100. Infine, in un commento laterale è specificato che un'unità di *DEFAULT_STEP_PERIOD* equivale a 2ms reali. Sia il *timeout* che il periodo temporale dei gradini sono stati scelti nell'esempio molto più grandi di quelli previsti dalle configurazioni di 5.47: ciò è stato fatto per facilitare la comprensione dell'andamento dalla visualizzazione del grafico.

Terminata l'introduzione, è opportuno passare alla descrizione delle attività che hanno portato alla correzione dei valori di corrente leggibili su *CANape* e alla definizione della relazione tra valore di corrente memorizzato e utilizzato internamente dal *driver* (*SO_AVG_VAL* e riferimenti) e quello reale. I parametri temporali caratterizzanti il riferimento di *reload* sono stati variati come segue (fig. 5.49), allo scopo di velocizzare il raggiungimento del *set point* e di avere più tempo a disposizione per le misure.

```
#define DEFAULT_STEP_PERIOD             5
#define DEFAULT_RELOAD_TIMEOUT          TIME_100MS(20)
```

Fig. 5.49: parametri modificati del riferimento per il reload

Quindi, il periodo dei gradini (e quindi il tempo necessario per raggiungere il *RELOAD_SET_POINT*) è stato dimezzato da 20ms a 10ms, invece il *timeout* che definisce il termine del controllo è stato aumentato da 0.5s a 2s.

Il *tester* è stato inserito lungo la linea di attuazione come descritto in precedenza; poi, il PC è stato interfacciato con la scheda MAIN mediante l'apposita sonda di programmazione, collegata al *driver* sulla porta J3. Vengono ora presentate le fasi eseguite per ciascun punto di calibrazione; in particolare, sono stati

considerati valori di *RELOAD_SET_POINT* da 0 in poi, a step di 5, fino a quando il tester non ha rilevato una corrente vicina ai 10A (sopra non si può andare per la presenza del fusibile).

(1) All'interno dell'IDE del *driver* (ossia CCS), il valore della costante *RELOAD_SET_POINT* è modificato e posto pari al punto di taratura di interesse; tale numero viene anche memorizzato su *Excel*.

(2) Il progetto viene compilato e si avvia la modalità debug, che consente di far eseguire il *firmware* dalla scheda in modo controllato e di mantenere sotto osservazione una o più variabili di interesse.

(3) Si avvia la registrazione su *CANape*.

(4) Viene dato il comando di RELOAD MAIN, che porta alla generazione di un grafico su *CANape* come il seguente.

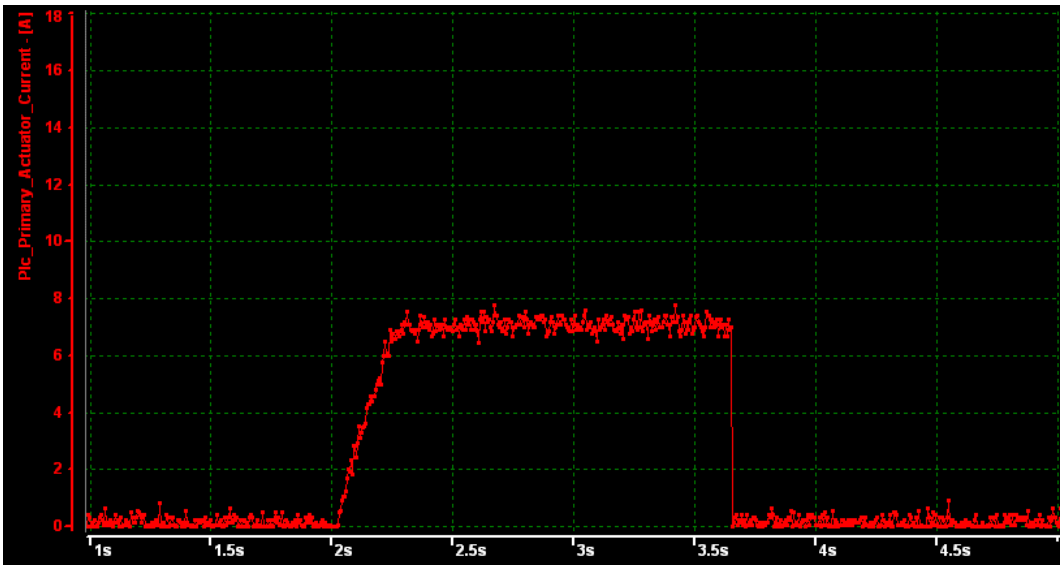


Fig. 5.50: andamento di corrente determinato da comando RELOAD MAIN

(5) Mentre l'andamento *CANape* rivela che il culmine della rampa è stato raggiunto e che la corrente assorbita dal carico dovrebbe essere approssimativamente costante, si effettuano tre letture di corrente dal display del tester; queste vengono memorizzate nella tabella *Excel* di prima e se ne calcola il valore medio.

(6) Si ferma la registrazione, si ingrandisce la porzione centrale del tratto quasi costante di corrente.

(7) Premendo il tasto destro del *mouse* e selezionando *Statistics*, è possibile leggere la media aritmetica del segnale in questione calcolata direttamente da *CANape* (fig. 5.51): questo valore viene salvato a fianco dei precedenti.

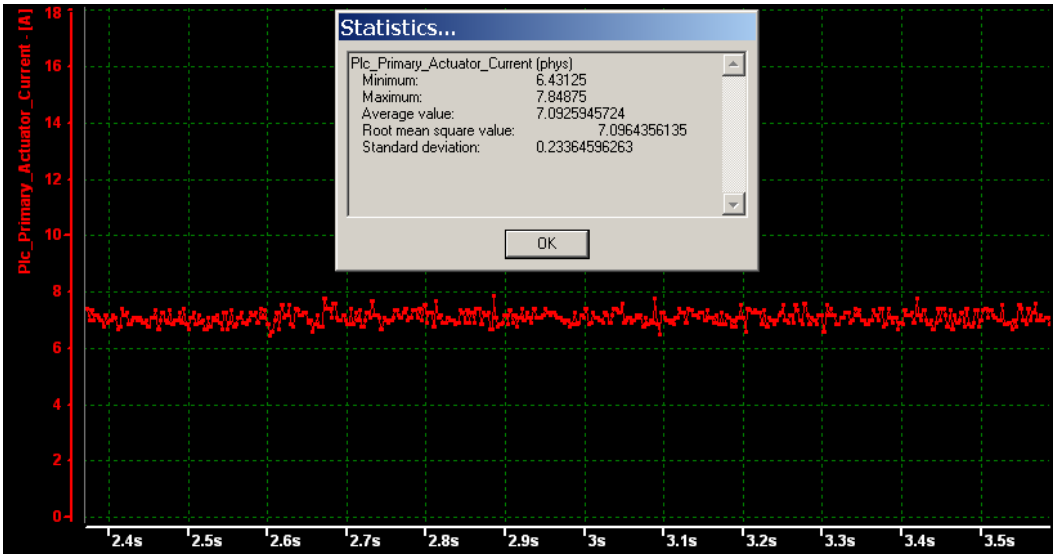


Fig. 5.51: visualizzazione del valore medio della corrente

La tabella successiva riporta i valori di interesse misurati relativi alla corrente.

RELOAD_SET_POINT	**Corrente media tester [A]**	**Corrente media *CANape* [A]**	**Errore da compensare [A]**
0	0,000	0,000	0,000
5	0,910	0,295	-0,615
10	1,980	0,742	-1,239
15	2,437	1,148	-1,289
20	3,210	1,572	-1,638
25	3,563	2,006	-1,557
30	4,310	2,403	-1,907
35	4,643	2,841	-1,802
40	5,030	3,293	-1,737
45	5,557	3,670	-1,887
50	5,910	4,092	-1,819
55	6,193	4,571	-1,622
60	6,593	4,950	-1,643
65	7,003	5,379	-1,624
70	7,227	5,821	-1,405

75	7,410	6,230	-1,180
80	7,743	6,649	-1,094
85	8,117	7,091	-1,026
90	8,350	7,517	-0,833
95	8,543	7,937	-0,607
100	8,663	8,356	-0,308
105	9,010	8,785	-0,226
110	9,313	9,204	-0,109
115	9,568	9,568	0,000

Tab. 5.6: tabella per la della lettura di corrente

I due grafici che seguono mostrano prima il confronto tra i valori reali di corrente e quelli che si potevano leggere da *CANape* prima della taratura e poi l'andamento dell'errore da compensare rispetto alla corrente originale da *CANape*.

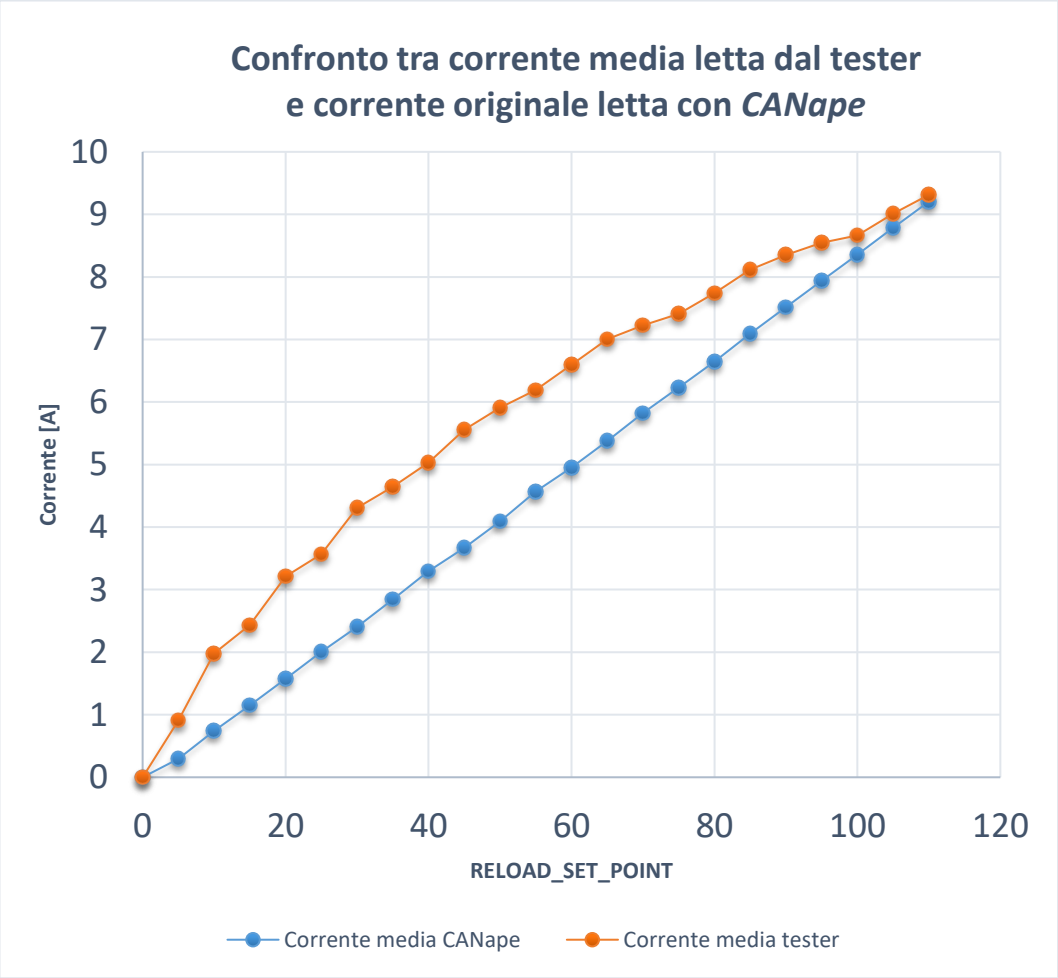


Fig. 5.52 : grafico di confronto tra corrente reale e corrente CANape prima della taratura

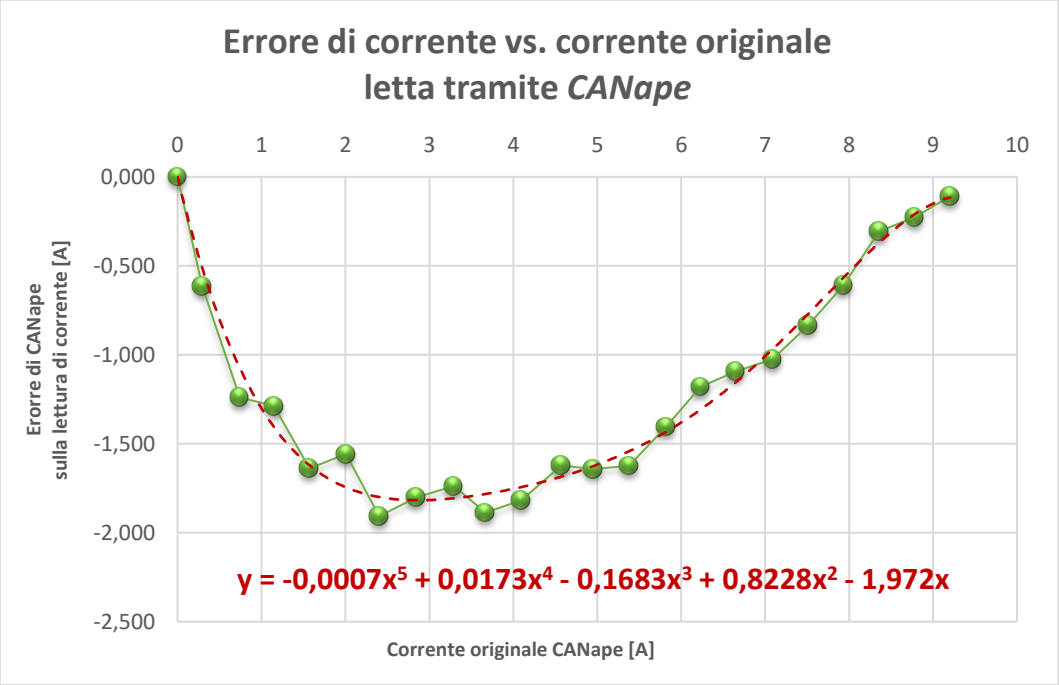


Fig. 5.53 : andamento dell'errore di corrente da correggere vs. corrente originale CANape

Analizzando i due grafici precedenti emerge che, mediamente, la differenza tra le due misure cresce (in modulo) nel tratto iniziale, quindi raggiunge una porzione di picco per poi diminuire e raggiungere approssimativamente lo zero per valori di corrente originale *CANape* vicini a 10A. Non è stato possibile effettuare misurazioni per correnti maggiori di 10A a causa della presenza del fusibile nel tester; si è dunque ipotizzato che, sopra i 10A di corrente assorbita, i valori presentati da *CANape* corrispondessero esattamente a quelli reali (e viceversa) e quindi la differenza da compensare si mantenesse uguale a zero da quel valore limite in poi. Si tratta di un'ipotesi molto forte e, a prima vista, avventata dato che non c'è nulla che impedisca all'errore di crescere in modulo per correnti elevate. Però, una delle richieste che mi sono state poste riguardo l'andamento della corrente nelle prove a fatica è che questa sia limitata a 10A (a meno di transitori repentini); pertanto, il range di misura e visualizzazione oltre i 10A di corrente non è di interesse e questo rende accettabile un'approssimazione del genere.

In figura 5.53 è stata inserita la linea di tendenza di quinto ordine che approssima l'andamento dell'errore mostrato e la relativa equazione; questa espressione è stata inserita in *parklock.c* per correggere i valori di corrente al

primario e al secondario che vengono poi inseriti all'interno del messaggio CAN di uscita. Le due immagini successive mostrano le linee di codice di *parklock.c* relative al calcolo della corrente al primario (per quella al secondario valgono le stesse espressioni) da inserire nel CAN, prima nel sorgente originale e poi in quello opportunamente corretto.

```
tmpprimary_current = tmpprimary_current * 0.032258 * 100.0 * 2.2625;
```

Fig. 5.54: calcolo originale del valore di corrente primaria da trasmettere via CAN

```
float corr_primario;
float correzione_corr;

//calcolo la corrente senza correzione
corr_primario = tmpprimary_current * 0.032258 * 2.2625;

if (corr_primario > 10.0) { //se la corr. originale è sopra i 10A...

  tmpprimary_current = (int) corr_primario; //...mantengo il calcolo originale

} else { //se la corr. originale è sotto i 10A...

  //calcolo la correzione
  a5 = pow(corr_primario,5);
  a4 = pow(corr_primario,4);
  a3 = pow(corr_primario,3);
  a2 = pow(corr_primario,2);
  a1 = corr_primario;
  correzione_corr = -0.0007*a5 + 0.0173*a4 - 0.1683*a3 + 0.8228*a2 - 1.972*a1;

  corr_primario = corr_primario - correzione_corr; //correggo la corrente
  tmpprimary_current = (int)(corr_primario * 100.0); //faccio cast a intero e x100

}
```

Fig. 5.55: calcolo originale del valore di corrente primaria da trasmettere via CAN

La parte di codice presentata in figura 5.55 è accompagnata da commenti che spiegano, passo per passo, l'implementazione della correzione della corrente. Si aggiunge soltanto che, siccome il calcolo della correzione si basa sul valore di corrente che si leggerebbe sulla GUI configurata, il fattore di scala originale della corrente pari a 2.2625 è stato rimosso dal database *CANape* ed integrato nel codice sorgente; quindi, il fattore di scala passa da 0.02625 a 0.01, uguale a quello della posizione, e non sono unitari in quanto entrambi compensano la moltiplicazione per 100 nel codice che serve ad aumentare la risoluzione.

Per quanto riguarda la caratterizzazione della scala utilizzata dal *driver* per gestire i valori di corrente, anche in questo caso è stato necessario utilizzare due

approcci differenti a seconda che la corrente reale sia sopra o sotto i 10A. I grafici seguenti mostrano l'andamento della corrente reale e di quella originale *CANape* rispetto al *RELOAD_SET_POINT*, ossia il riferimento finale della rampa di corrente avviata con il pulsante RELOAD MAIN.

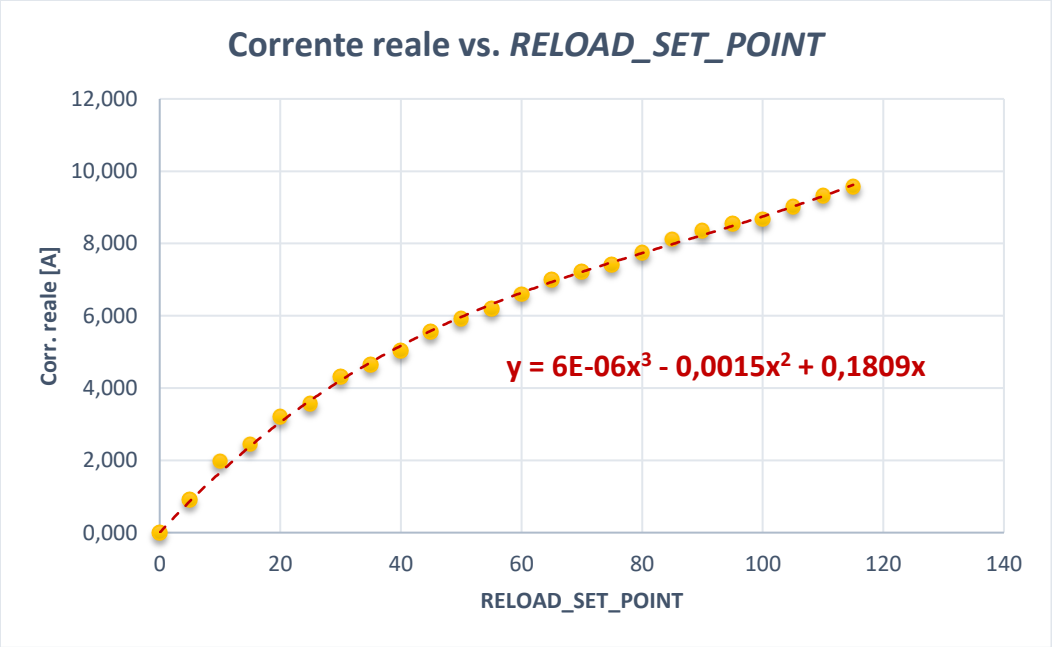


Fig. 5.56: andamento corrente reale vs. RELOAD_SET_POINT

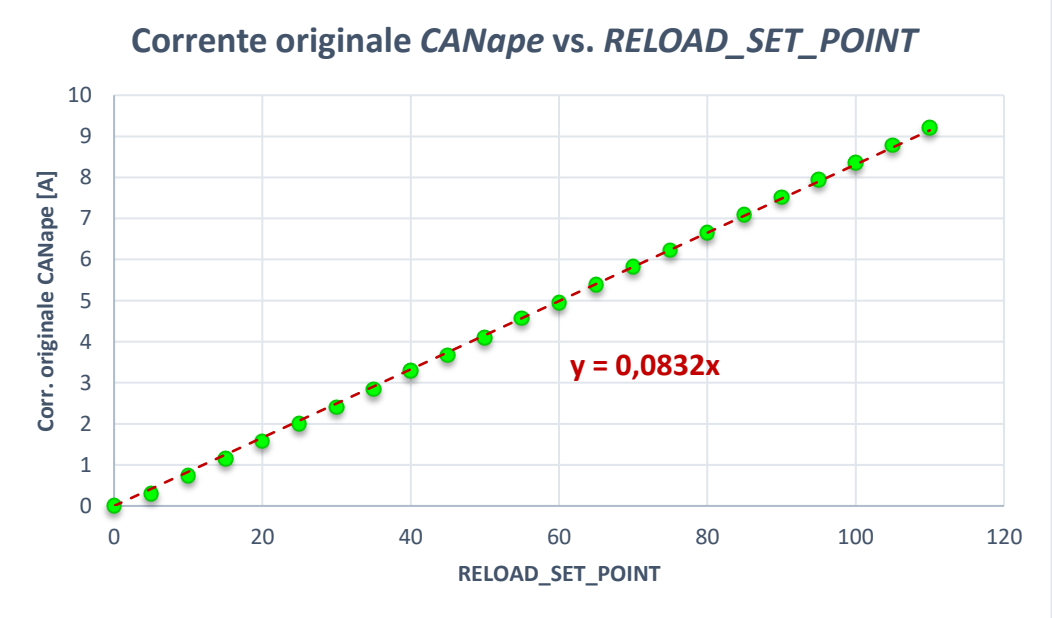


Fig. 5.57: andamento corrente originale CANape vs. RELOAD_SET_POINT

Pertanto, se si desidera individuare il valore effettivo di corrente che corrisponde ad un valore espresso nella scala del *driver*, bisogna utilizzare la legge cubica di figura 5.56 se il *RELOAD_SET_POINT* (o il *SO_AVG_VAL* e così via) è inferiore a 115 (10A reali), altrimenti si usa l'espressione lineare di fig. 5.57.

Allo stesso modo è molto utile saper passare dal valore reale a quello della corrente nella scala del *driver*; per fare ciò, sopra i 10A basta invertire la formula di primo grado appena vista, mentre sotto i 10A si rende necessaria una nuova interpolazione cubica, presentata nella figura successiva.

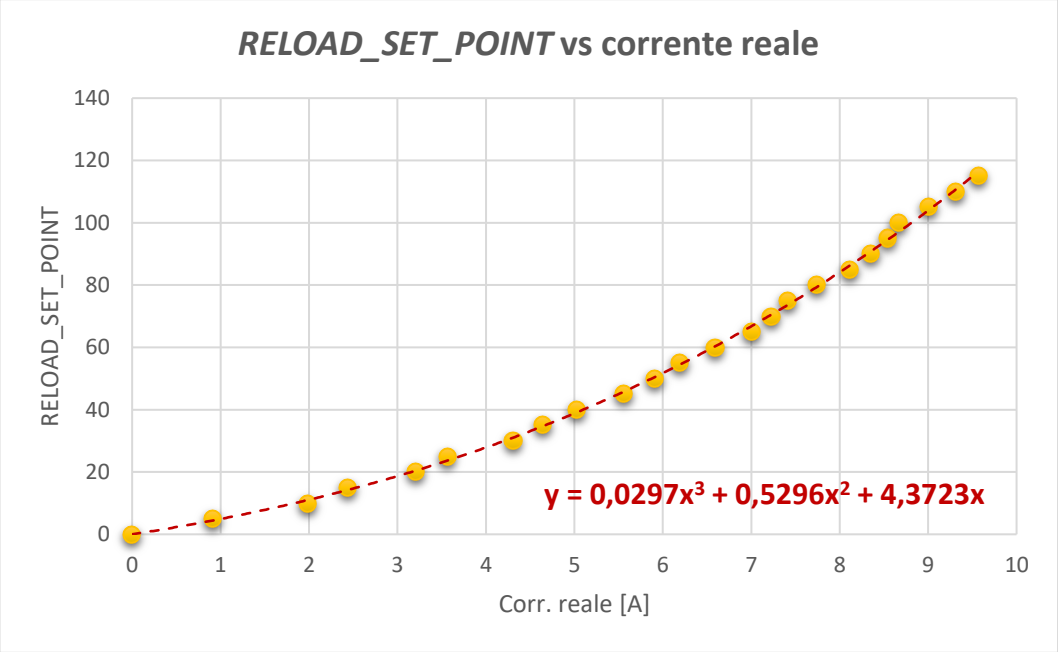


Fig. 5.58: andamento RELOAD_SET_POINT vs. corrente reale

La tabella successiva riassume le formule da utilizzare per le conversioni di scala legate al funzionamento del *driver*.

Conversione da corrente reale (x) a corrente *driver* (y)	
corrente reale ≤ 10A	**corrente reale > 10A**
$y = 0.0297x^3 + 0.5295x^2 + 4.3723x$	$y = x/0.0832$
Conversione da corrente *driver* (x) a corrente reale (y)	
corrente *driver* ≤ 115	**corrente *driver* > 115**
$y = 0.000006\,x^3 + 0.0015x^2 + 0.1809x$	$y = 0.0832x$

Tab. 5.7 : tabella per la conversione delle misure di corrente del driver

Le relazioni di conversione da valore reale a scala *driver* sono state implementate a livello software nella forma di macro, le quali effettuano la conversione in modo automatico quando vengono richiamate nel programma. Sono state introdotte due macro (una per la conversione sotto 10A e l'altra per quella sopra), le cui definizioni sono state inserite in *config.h* (fig. 5.59) del progetto *driver* e che sono state utilizzate, ad esempio, per definire i valori di riferimento per la generazione della rampa di *reload*.

```
#define AMP_TO_ADCVAL_inf10A(x)      (int)(0.000006 * x^3 - 0.0015 * x^2 + 0.1809 * x)
#define AMP_TO_ADCVAL_sup10A(x)      (int)(x/0.0832)
```

Fig. 5.59: macro definite per la conversione della corrente

5.5 Limitazione della corrente a 10A in attuazione

Una volta calibrato il nuovo algoritmo di controllo, si è potuto procedere verso il soddisfacimento di un altro requisito inerente il sistema da me configurato, questa volta relativo alla corrente assorbita dal motorino *brushed*. Mi è stato domandato di implementare nel controllo la limitazione a 10A della corrente erogabile in modo continuativo nelle fasi di *engage* e *disengage*. Questo genere di requisito è importante per evitare il danneggiamento dei componenti della catena di attuazione e di quella cinematica. Ad esempio, uno dei maggiori pericoli che concerne i motori elettrici è di natura termica, legato al raggiungimento di una temperatura interna troppo elevata. La potenza dissipata dagli avvolgimenti per effetto Joule è proporzionale al quadrato della corrente che li percorre; se la corrente assorbita è molto alta (rispetto a quella nominale) e ciò non avviene in modo transitorio, il calore prodotto non riesce a sfogare verso l'esterno: il motore, dunque, si surriscalda ed è probabile che uno o più elementi isolanti presenti intorno ai fili elettrici perda le proprie proprietà e determini un irreparabile cortocircuito che 'brucia' gli avvolgimenti in rame (fig. 5.60). Un altro possibile danneggiamento di origine analoga è il consumo completo delle spazzole di grafite, le quali finiscono per saldarsi sulle lamelle del collettore (fig. 5.61). In secondo luogo, molta corrente significa molta coppia e questa condizione, soprattutto se dura a lungo, può determinare lesioni alla meccanica, inceppamenti e addirittura *creep*.

Fig. 5.60: rotore con alcuni avvolgimenti bruciati

Fig. 5.61: motore con una spazzola saldata sul collettore

La limitazione a 10A è stata realizzata in modo molto semplice, sfruttando le strutture già implementate nell'algoritmo originale del *driver*. Quando si entra in una fase di attuazione controllata di innesto o disinnesto, la funzione *motor_control* (già nominata più volte in precedenza) avvia un *loop* di corrente a fianco dell'esecuzione delle istruzioni che portano al calcolo del *dc* di comando per interpolazione. Il *loop* di corrente riceve in ingresso un *set point* a gradino, che parte con *motor_control*, e come *feedback* la misura della corrente attualmente assorbita (entrambi i valori sono nella scala utilizzata dal *driver*); l'errore calcolato diventa l'input di un PID identico a quello presente nell'algoritmo originale. L'anello chiuso di corrente restituisce un valore di *dc*, quello che dovrebbe essere indirizzato al ponte per portare la corrente a 10A; il *dc* restituito dal *loop* e quello interpolato

sono confrontati e viene conservato il valore minore, usato quindi per definire la tensione di comando del motore. Pertanto, quando l'algoritmo implementato tende a portare la corrente oltre al limite prestabilito mediante un *duty cycle* eccessivo, il *loop* di corrente consente di livellare adeguatamente l'output di *motor_control*.

La figura seguente schematizza il sistema utilizzato per limitare la corrente erogata in modo continuativo a 10A.

Fig. 5.62: schema a blocchi dell'algoritmo di limitazione della corrente

Per verificare il funzionamento della limitazione introdotta, si è proceduto a modificare l'algoritmo di comando, in modo che nella fase di disinnesto fosse imposto un *dc* costante pari a 70 fino allo scadere del relativo *timeout* di 2s. Siccome il motore è caratterizzato (secondo la scheda tecnica) da 42A@12V e rotore fermo, i $12 * 0.7 = 8.4V$ di comando associati ad un *dc* pari a 70 dovrebbero portare all'assorbimento di circa $42 * 0.7 = 29.4A$. La figura successiva mostra quanto accade premendo DISENG MAIN dopo questa modifica.

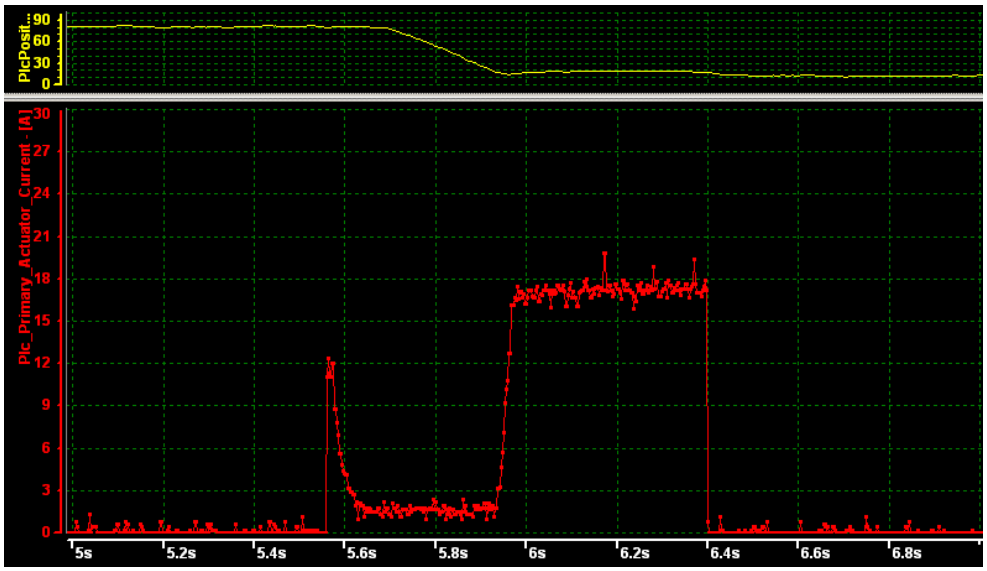


Fig. 5.63: effetto dell'assenza della limitazione a 10A

Come previsto, il *dc* di comando costante applicato determina un assorbimento corrente superiore a 10A in due punti:

1. Allo spunto, in quanto il motore deve produrre una coppia superiore alla media per vincere le resistenze associate all'attrito statico e alle inerzie per avviare il moto.
2. Quando la leva si trova schiacciata contro il finecorsa di *engage*, condizione in cui la forza controelettromotrice è nulla per la mancanza di moto e quindi la tensione netta percepita ai capi degli avvolgimenti è praticamente uguale a quella di comando.

Si osserva ancora che l'assorbimento di corrente registrato al termine della corsa di innesto è notevolmente inferiore a quello previsto (circa 17A reali contro i 29.4A teorici); questo è causato, con ogni probabilità, dal fatto che la resistenza degli avvolgimenti del motorino è aumentata con il passare del tempo a causa, ad esempio, del deterioramento del contatto spazzole-collettore.

La figura seguente mostra cosa succede se, nel codice sorgente, viene introdotto l'anello di controllo e la comparazione di cui si è trattato sopra, a parità di composizione dei vettori di riferimento per l'interpolazione.

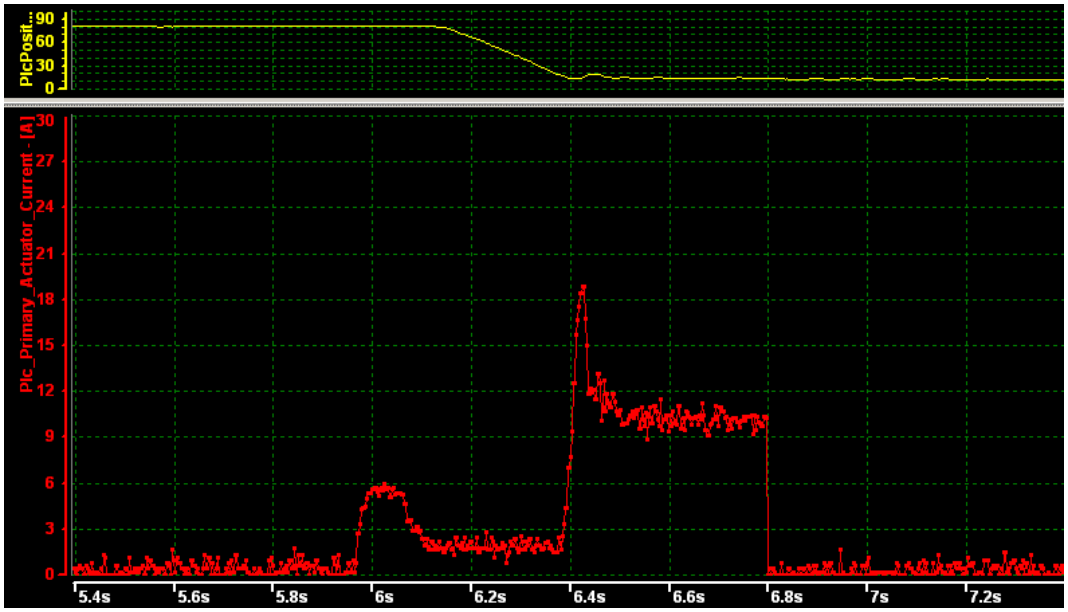


Fig. 5.64: corrente limitata a 10A con algoritmo di controllo CL

L'immagine precedente conferma che l'algoritmo implementato funziona: l'assorbimento continuo di corrente è bloccato ad un massimo di 10A. Si nota, però, che all'inizio (quando inizia a passare corrente) e al termine (quando la leva sbatte contro il finecorsa) della fase di *engage* vengono registrati dei picchi di corrente che superano i 10A. Infatti, il sistema di limitazione non va bene per i transitori: l'esistenza di intervallo temporale, seppur breve, che intercorre tra l'istante in cui la corrente supera i 10A e quello di correzione del *dc* di comando fa sì che una variazione repentina della corrente possa avvenire in modo indisturbato. In particolare, è impossibile ostacolare i picchi di corrente legati ad urti, come quello che si osserva a fondo innesto; in tale condizione il rotore si ferma almeno nell'istante dell'urto, in cui dunque viene meno la forza controelettromotrice proporzionale alla velocità di rotazione: gli avvolgimenti risentono di questo come se fosse un aumento istantaneo della tensione di comando e, di conseguenza, richiamano più corrente. In ogni caso mi è stato richiesto di impedire unicamente un assorbimento continuativo di corrente oltre i 10A, pertanto i risultati ottenuti si possono ritenere accettabili.

5.6 Gradino di corrente a 10A a fine corsa

Mi è stato domandato di soddisfare un altro requisito relativo all'andamento della corrente; questa volta però la specifica non riguardava il comportamento del *park-lock* nelle fasi di movimentazione ma quello che segue il raggiungimento delle posizioni di *engage* e *disengage*. Ho dovuto, infatti, modificare il progetto sorgente del *driver* in modo che venisse realizzato un gradino di corrente di ampiezza 10A e di durata 100ms sia a seguito del raggiungimento del finecorsa di innesto (il perno dell'arpione) che della posizione limite di disinnesto (definita dal contatto tra arpione e camma desmodromica).

Il fine pratico di questa nuova richiesta è quello di far simulare alla centralina per i test quanto viene realizzato dalla centralina on board per scopi diagnostici. Infatti, nell'introduzione del secondo capitolo si è evidenziato che i veicoli dotati di *park-lock* sono provvisti di un'unità di controllo che deve saper riconoscere gli errori nel funzionamento del dispositivo. In particolare, è noto che la ECU del

veicolo elettrico che verrà dotato del *park-lock Graziano* dispone del seguente algoritmo diagnostico: quando il sensore Hall trasmette alla centralina on board un segnale che identifica il raggiungimento di una delle due soglie, viene prodotto un gradino di corrente di attuazione con ampiezza pari a 10A, producendo coppia motrice sull'alberino di ingresso. Se si rileva una significativa variazione di *dc* di posizione allora la posizione di finecorsa non era stata raggiunta veramente e quindi sono prodotti dei messaggi che informano l'ECU centrale e l'utente della presenza di un malfunzionamento del *park-lock*.

Le modifiche effettuate sul codice hanno coinvolto principalmente la macchina a stati di *app_fsm.c* e, in particolare, sono state variate le istruzioni da eseguire quando la variabile *app_status* (che rappresenta lo stato dell'applicazione) assume i valori *STATUS_ENGAGED* e *STATUS_DISENGAGED*. Prima di spiegare i cambiamenti eseguiti su *app_fsm.c* è opportuno riportare le definizioni inerenti i tre *timeout* (o timer) che caratterizzano l'esecuzione del gradino. La figura 5.65 mostra la dichiarazione e l'inizializzazione dei *timeout* (in *app_fsm.c*), mentre la figura 5.66 presenta le definizioni pre-compilazione dei loro valori di default (in *config.h*).

```
/* Timeout per il gradino */
timeout_t prestep_timeout = 0; //timeout per avvio gradino 10 A
timeout_t step_timeout = 0; //timeout per fine gradino 10 A
timeout_t poststep_timeout = 0; //timeout per proseguimento dopo gradino
```

Fig. 5.65: inizializzazione dei timeout caratterizzanti il gradino a 10A

```
/* Valori di default per i timeout del gradino */
#define DEFAULT_PRESTEP_TIMEOUT             20  // x10ms = 200 ms
#define DEFAULT_STEP_TIMEOUT                10  // x10ms = 100 ms
#define DEFAULT_POSTSTEP_TIMEOUT            20  // x10ms = 200 ms
```

Fig. 5.66: definizione dei valori di default *dei timeout caratterizzanti il gradino a 10A*

Ci sono tre costanti di tempo che riguardano l'esecuzione di un gradino:

- *prestep_timeout*: è l'intervallo di tempo che intercorre tra quando viene accertato il raggiungimento di una soglia (a cui corrisponde l'aggiornamento di *app_status* a *STATUS_ENGAGED* o

STATUS_DISENGAGED) e l'istante in cui si avvia l'esecuzione del gradino. Il suo valore che è stato scelto come default è pari a 200ms.

- *step_timeout*: è la durata del gradino, imposta per specifica a 100ms.
- *poststep_timeout*: è l'intervallo di tempo che si lascia passare tra il momento in cui finisce il gradino di corrente e quello a partire da quale può arrivare un nuovo comando di innesto/disinnesto. Il valore scelto come default corrisponde a 200ms.

I valori dei tre timer sono definiti man mano che il codice relativo al gradino viene eseguito e sono azzerati tutti e tre insieme una volta terminato lo step a 10A, come verrà ben spiegato tra poche righe.

Passando all'implementazione pratica del gradino, inizialmente questa è stata realizzata imponendo per 100ms un *dc* di comando costante al ponte H tale da produrre la corrente desiderata di 10A. Il valore del *dc* è stato valutato facilmente, in quanto è noto dalla scheda tecnica del motore che il carico assorbe 42A@12V in condizione di stallo (rotore fermo); dunque, considerando costante la resistenza degli avvolgimenti, il *dc* ricercato si può calcolare con una semplice proporzione:

$$dc_{42A}: 42A = dc_{10A}: 10A \tag{5.9}$$

$$dc_{10A} = 100 * \frac{10}{42} \cong 24 \tag{5.10}$$

La figura 5.67 mostra le linee di codice modificate o introdotte dal nuovo in *app_fsm.c* e, in particolare, all'interno *switch* di *app_status*.

```
64 case STATUS_ENGAGED:
65 case STATUS_DISENGAGED:
66
67     if(!flag_step) {
68
69     if(prestep_timeout==0) {
70         set_timeout(prestep_timeout, TIME_10MS(DEFAULT_THR_PRESTEP_TIMEOUT));
71     }
72     if(!is_elapsed(prestep_timeout)) {
73         drv8701_pwm = 0;
74     } else {
75             if(step_timeout ==0) {
76                 set_timeout(step_timeout, TIME_10MS(DEFAULT_THR_STEP_TIMEOUT));
77             }
78             if(!is_elapsed(step_timeout)) {
79                 drv8701_pwm = 24;
80             } else {
81             if(poststep_timeout ==0) {
82                 set_timeout(poststep_timeout, TIME_10MS(DEFAULT_THR_POSTSTEP_TIMEOUT));
83             }
84             if(!is_elapsed(poststep_timeout)) {
85                 drv8701_pwm = 0;
86             } else {
87                 prestep_timeout = 0;
88                 step_timeout = 0;
89                 to_poststep = 0;
90                 flag_step = 1;
91                 }
92             }
93        }
94
95     } else {
96         reset_control();
97         status_from_position();
98         start_command();
99     }
100 break;
```

Fig. 5.67: codice di implementazione del gradino a 10A con dc costante

Si procede, dunque, alla descrizione riga per riga della porzione di codice appena riportata.

- 64,65. Le istruzioni successive vengono realizzate quando *app_status* è uguale a *STATUS_ENGAGED* o *STATUS_DISENGAGED*.
- 67. Innanzitutto, la variabile booleana *flag_step* viene dichiarata e inizializzata a 0 all'inizio di *app_fsm.c* ed è utilizzata per tenere traccia della realizzazione del gradino. La variabile è posta ad 1 quando il gradino è stato realizzato (ed è passato anche il tempo di riposo che lo segue) e rimessa a 0 quando *app_status* passa a *STATUS_ENGAGE*/*STATUS_DISENGAGE*, ossia a seguito del riconoscimento di un comando di attuazione. La condizione del primo *if* (la negazione di *flag_step*) è vera quando *flag_step* è 0, ossia quando il gradino deve ancora essere realizzato: in questo caso si prosegue

con l'esecuzione delle istruzioni successive, altrimenti si passa direttamente all'ultimo *else*.

- 69-71. Se *prestep_timeout* è nullo (condizione logica per entrare nel secondo *if*) allora vuol dire che *app_status* è appena stato aggiornato al seguito del raggiungimento di una delle due soglie e quindi bisogna avviare la procedura di generazione del gradino, scandita dai tre tempi nominati. Quindi, la macro *set_timeout* viene impiegata per impostare il valore di *prestep_timeout* al valore di default (200ms); in realtà, la macro definisce il numero di *tick* che il *clock* di sistema avrà battuto dopo 200ms dall'applicazione della macro.
- 72,73. La condizione logica del terzo *if* viene definita utilizzando la macro *is_elapsed*; questa prende in argomento un *timeout* in cui è stato memorizzato il relativo numero di *tick* di sistema e verifica se i *tick* attuali sono superiori a quelli del timer: se è vero (il *timeout* è trascorso) restituisce 1, se è falso restituisce 0. Dunque, la condizione *!is_elapsed(prestep_timeout)* è vera se la fase pre-gradino non è ancora terminata (in quanto il punto esclamativo indica una negazione) e, in questo caso, il *duty_cycle* di comando (attribuito a *DRV8701_pwm*) viene posto a zero.
- 74-79. Se invece *is_elapsed(prestep_timeout)* è vero, allora l'else conduce alla realizzazione del gradino vero e proprio. La sequenza di istruzioni è identica a quella precedente: prima si usa *set_timeout* per definire il valore di *step_timeout* corrispondente a 100ms e poi, se il timer non è trascorso, viene imposta l'applicazione di un *dc* di comando pari a 24.
- 80-85. Una volta superato il timeout di esecuzione del gradino e quindi terminati i 100ms a 10A, un nuovo else porta all'impostazione del timer *post_timeout* e all'applicazione per 200ms di *dc* di comando nullo all'*H-Bridge*.
- 87-94. Trascorse tutte e tre le fasi temporali che caratterizzano la realizzazione di un gradino, vengono eseguite per un ciclo solo le linee di codice racchiuse dal penultimo *else*, che servono principalmente a

predisporre le variabili di interesse all'esecuzione di un futuro gradino: i tre timer sono resettati a 0 e *flag_step* è posta uguale ad 1.

- 95-99. L'ultimo *else* comprende le istruzioni che descrivevano il comportamento del sistema per *app_status* uguale a *POS_ENGAGE*D e *POS_*DIS*ENGAGE*D prima che implementassi la realizzazione del gradino di corrente; vengono infatti eseguite quando *flag_step=1*, ossia il gradino è stato portato a termine. Quindi *reset_control* resetta i parametri di *motor_control* e delle funzioni ad esso associate, *status_from_position* determina il valore di *app_status* da quello di *pos_status* e, infine, *start_command* predispone il sistema per la lettura e l'interpretazione di un nuovo comando ricevuto via I2C.

La figura successiva mostra l'andamento della corrente in seguito ai comandi di innesto e disinnesto dopo le modifiche realizzate sul codice.

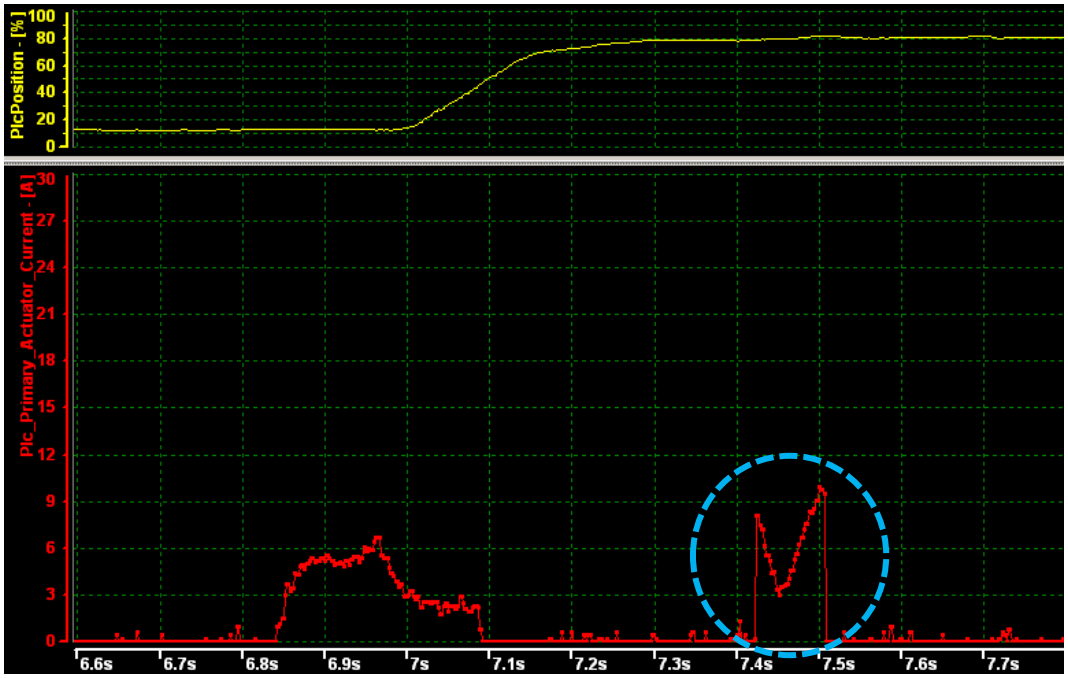


Fig. 5.68: gradino a 10A ottenuto con dc costante (cerchiato in azzurro)

L'immagine precedente rivela che il metodo di implementazione utilizzato ha delle problematiche evidenti; la corrente assorbita non vale 10A, nemmeno in modo approssimativo, e il suo andamento è tutt'altro che costante: all'inizio vale circa 8A, quindi decresce rapidamente a 3A per poi tornare a risalire. Inoltre, è errato alla

base pensare di ottenere dei gradini di ampiezza costante a 10A per tutta una prova a fatica utilizzando un *dc* (e quindi una tensione di comando) costante; infatti, il calcolo del *dc* da applicare per ottenere i 10A è stato realizzato sotto l'ipotesi di resistenza costante degli avvolgimenti. Si tratta, in realtà, di un'affermazione molto forte nel caso di prove che prevedono che il motore venga azionato per molte ore consecutive, come accade per le prove a fatica; il riscaldamento degli avvolgimenti porta all'aumento della loro resistenza e, a parità di tensione di comando applicata, ciò determina la riduzione della corrente assorbita.

Pertanto, si è deciso di abbandonare la realizzazione del gradino attraverso l'applicazione di un *dc* costante e di riutilizzare (come nel caso della limitazione di corrente) il codice che implementa il *loop* di corrente (PID + sommatore) già presente nei sorgenti originali. La figura successiva mostra solo la porzione di programma modificata, quella che descrive le operazioni che devono essere eseguite nella fase di *step_timeout.*

```
124 if(!is_elapsed(step_timeout)) {
125
126    pid_set_point = DEFAULT_STEP_SET_POINT;
127    dc_extended += update_current_pid();
128
129    if (dc_extended > (DRV_PWM_MAX << Nbit))
130    {
131        dc_extended = (DRV_PWM_MAX << Nbit);
132    }
133    else if (dc_extended < 0)
134    {
135        dc_extended = 0;
136    }
137
138    drv8701_pwm =(dc_extended >> Nbit);
139
140 }
```

Fig. 5.69: codice di implementazione del gradino a 10A con loop PID sulla corrente

In riferimento alla figura precedente, una volta iniziata la fase di esecuzione del gradino (riga 124), il *set point* del PID di corrente è impostato sul valore di default pari a 10A, definito in *config.h* (riga 126). Quindi, il *dc* esteso di 6 bit viene calcolato mediante sommatore a partire dall'uscita del PID (riga 127). Il valore così trovato viene limitato al valore massimo (corrispondente all'estensione di 100 di 6 bit) e posto a 0 se risulta negativo (righe 129-136). Infine, il *dc* di comando viene calcolato riducendo di 6 bit quello esteso.

L'andamento di corrente mostrato nella figura successiva indica che l'utilizzo dell'anello chiuso di corrente consente di realizzare lo step a 10A e le successive prove a fatica ne hanno validato il funzionamento sia nel breve che nel lungo periodo. Si nota che la corrente richiamata dal motore parte con un picco di circa 19A, per poi decrescere molto rapidamente (per effetto dell'algoritmo di controllo) e oscillare intorno ai 10A desiderati.

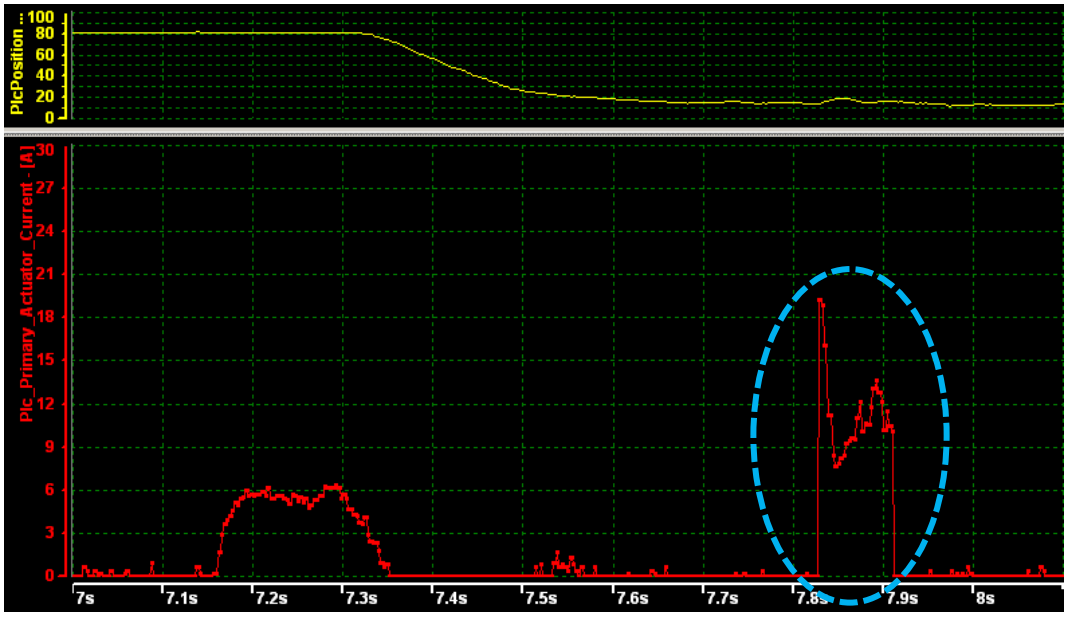


Fig. 5.70: gradino a 10A ottenuto con loop PID di corrente (cerchiato in azzurro)

L'implementazione del gradino di corrente a fine innesto e disinnesto è stata sfruttata per analizzare sperimentalmente gli effetti della variazione dei coefficienti del PID sull'andamento rilevato di corrente e verificare quanto affermato quando è stato descritto l'algoritmo originale di controllo. Si è scelto di effettuare questa analisi solo ora, variando i coefficienti del PID impiegato per l'esecuzione del gradino, perché l'assenza di movimento (e quindi di forza controelettromotrice) permette di concentrarsi sull'efficacia dell'algoritmo di controllo della corrente.

La figura seguente mostra, nello stesso piano, la sovrapposizione dei gradini di corrente a 10A che si ottengono aumentando la costante proporzionale (K_p) del PID. Per poter osservare gli andamenti oltre i transitori, la durata del gradino è stata aumentata a 1s (intervallo per cui l'assorbimento di 10A non rischia ancora di portare a problemi termici).

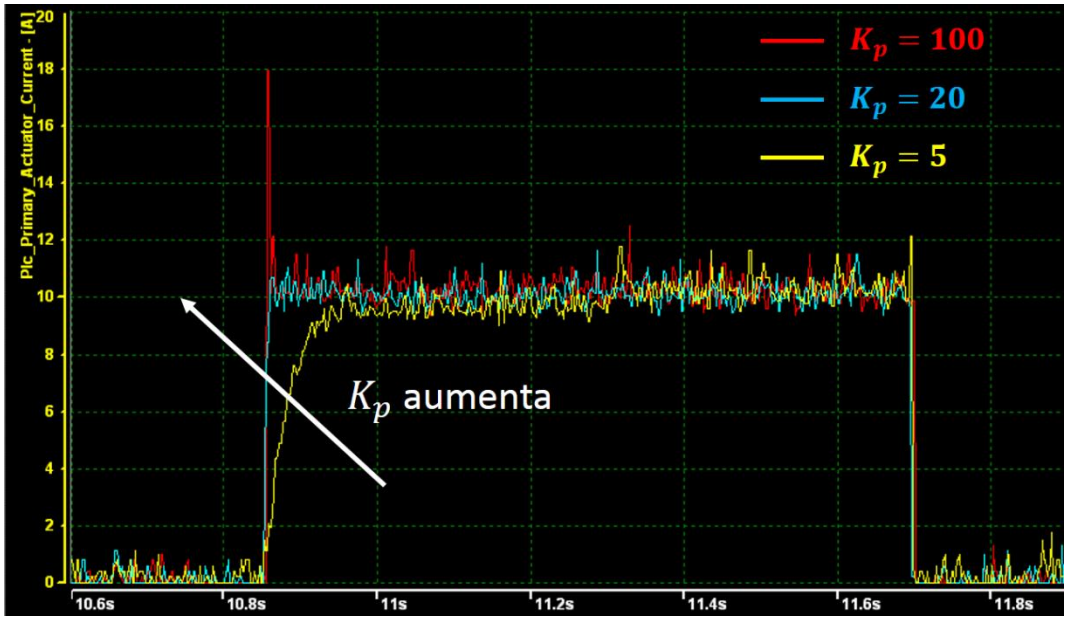


Fig. 5.71: effetto dell'aumento del K_p sul gradino di corrente

L'immagine successiva seguente mostra, invece, la sovrapposizione dei gradini che si ottengono aumentando la costante integrativa (K_i) del PID.

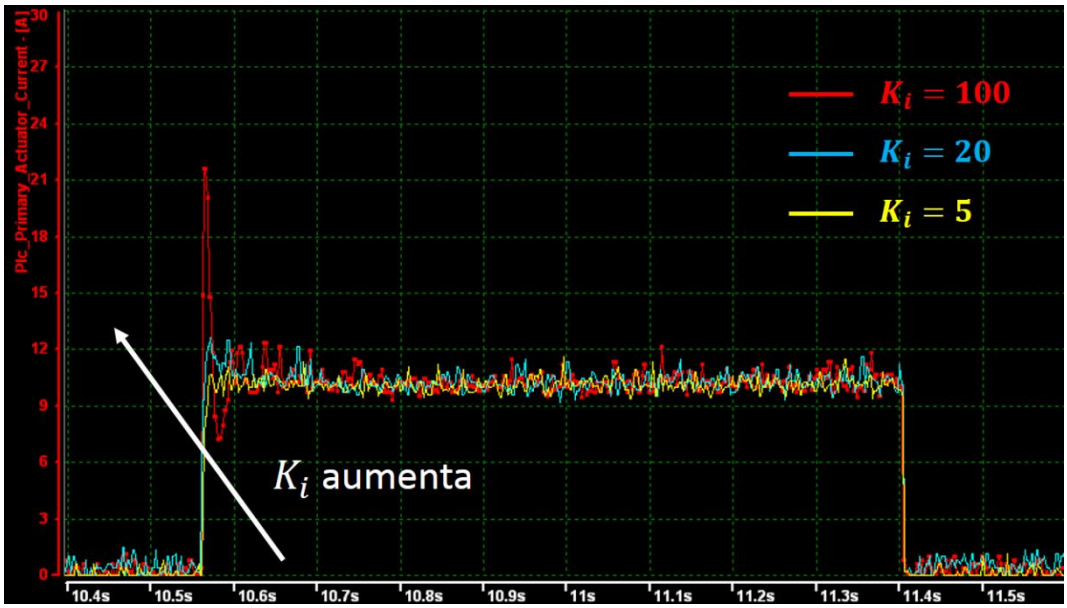


Fig. 5.72: effetto dell'aumento del K_i sul gradino di corrente

Le figure 5.71 e 5.72 confermano l'effetto dei contributi del controllore PID che è stato messo in evidenza nei paragrafi precedenti: aumentare il termine proporzionale e/o quello integrativo migliora la reattività del sistema ma lo avvicina

all'instabilità, come si evince dall'aumento della sovraelongazione rispetto al valore che si desidera raggiungere. Inoltre, le prove effettuate hanno confermato che i valori della costante proporzionale e di quella integrativa definiti di default nel progetto sorgente originali sono ottimi per il PID di controllo corrente utilizzato; mantenere $K_p = 20$ e $K_i = 5$ permette, infatti, di raggiungere il *set point* con grande rapidità e sovraelongazione praticamente assente.

Il *loop* di controllo presenta anche un sommatore all'uscita del PID, che serve a sostenere il *dc* di comando quando l'errore tra set e feedback è nullo o comunque molto piccolo. La presenza del sommatore, in generale, tende a rallentare un sistema di controllo (si tratta della discretizzazione di un integratore); però il guadagno intrinseco del sommatore dato dall'elevata velocità di esecuzione dell'algoritmo da parte del *driver* ne riduce l'effetto rallentante e, pertanto, gli effetti del PID rimangono ben visibili.

5.7 Automatizzazione del funzionamento ciclico

A questo punto, in cui le caratteristiche delle leggi di posizione e di corrente corrispondevano finalmente a quelle desiderate, è stato possibile procedere con l'implementazione sui sorgenti dell'esecuzione automatizzata delle prove a fatica. In particolare, l'utente della centralina doveva essere in grado di avviare la prova a fatica e di fermarla mediante l'impiego di due soli pulsanti: ENG MAIN per l'avvio e DISENG MAIN per lo stop.

Era possibile seguire due strade di implementazione differenti per raggiungere lo stesso obiettivo:

(1) Modificare i sorgenti del *core* e aggiungere un *loop* per comunicare i comandi di *engage* e *disengage* in modo alternato via I2C, dalla premuta del tasto ENG MAIN a quella del tasto DISENG_MAIN.
(2) Modificare i sorgenti dei *driver*, in particolare nelle parti che riguardano l'interpretazione dei comandi e la macchina a stati.

Tra le due modalità proposte è stata scelta la seconda, essenzialmente per la maggiore semplicità di strutturazione del progetto sorgente del *driver* rispetto a quello del *core*.

Passando alla descrizione delle modifiche e delle aggiunte praticate sul codice, la figura 4.7.1 mostra le prime righe di codice dello *switch* che, all'interno del sorgente di nome *i2c_protocol.c*, converte il numero scritto nel registro di memoria (*i2c_message[0]*), adibito alla memorizzazione del comando scritto dal *core* mediante I2C, in un comando (memorizzato nella variabile *requested_cmd*) significativo per la logica del *driver*.

```
switch(i2c_message[0])
{
case ENGAGE_BRAKE:
    i2c_cmd_engage();
    flag_ciclo = 1;
    tx_data_len = 0;
    break;
#if !AUX_BOARD_ENABLE
case DISENGAGE_BRAKE:
    i2c_cmd_disengage();
    flag_ciclo = 0;
    tx_data_len = 0;
    break;
#endif
```

Fig. 5.73: modifiche al codice di interpretazione del comando I2C

In entrambi i *case* presentati è presente la variabile booleana *flag_ciclo*, introdotta appositamente per implementare il funzionamento ciclico. Questa variabile è dichiarata nell'*header app.h* come *extern boolean* e inizializzata a 0 in *app_fsm.c*. La parola chiave *extern* indica che si tratta di una variabile globale, ossia che viene utilizzata e modificata da più sorgenti che includono *app.h*; l'informazione legata all'uso di *extern* è fondamentale poiché impedisce che il compilatore dia errore quando trova tale variabile in un sorgente in cui non è presente la sua definizione.

Se il contenuto del registro di memoria interrogato corrisponde ad *ENGAGE_BRAKE* allora vengono eseguite le seguenti istruzioni:

- La funzione *i2c_cmd_engage* viene chiamata per assegnare a *requested_cmd* (fig. 5.74) il valore corrispondente a quanto letto nel registro interrogato; questo valore, a sua volta, sarà trasformato in una serie di istruzioni da eseguire mediante la funzione *start_command* quando questa sarà chiamata dalla macchina a stati.

```
void i2c_cmd_engage(void)
{
    requested_cmd = CMD_ENGAGE;
}
```

Fig. 5.75: dichiarazione della funzione i2c_cmd_engage

- La variabile *flag_ciclo* viene posta ad 1; questo valore attribuito a *flag_ciclo* determinerà l'avvio dell'esecuzione dei cicli di attuazione, come si vedrà tra poco.
- La variabile intera *tx_data_len* è messa a 0, ad indicare che non devono essere trasmesse informazioni al *core* in risposta al comando appena trovato.

Se, invece, il registro di memoria *i2c_message*[0] contiene il messaggio binario che corrisponde a *DISENGAGE_BRAKE*, la sequenza delle operazione attuate è molto simile a quella prima:

- Si richiama la funzione *i2c_cmd_disengage* per attribuire un opportuno valore a *requested_cmd*.
- La variabile *flag_ciclo* è posto a 0 e ciò provocherà l'interruzione dell'esecuzione dell'attuazione ciclica.
- La variabile *tx_data_len* è messa a 0 per lo stesso motivo di prima.

Si sottolinea che le direttive di precompilazione presenti in figura 5.73 fanno sì che il codice relativo all'interpretazione del comando di *disengage* sia compilato ed integrato nell'eseguibile quando la compilazione è eseguita nella configurazione MAIN; infatti, nella configurazione AUX (almeno secondo il progetto originale) il *driver* deve comandare un solenoide con una rampa di corrente fino a 25A ed evidentemente per tale *driver* il comando di disinnesto non avrebbe significato.

Si può, dunque, passare alla spiegazione delle variazioni apportate al codice della macchina a stato realizzata dallo *switch* di *app_fsm.c*. In particolare, sono state apportate modifiche alle ultime linee di codice dei *case STATUS_ENGAGED* e *STATUS_DISENGAGED*. Innanzitutto, è stato necessario sdoppiare i due *case* appena nominati, in quanto le istruzioni eseguite dopo il gradino non devono essere più le stesse. Quindi, la figura successiva presenta il codice da eseguire quando *app_status* vale *STATUS_ENGAGED* ed è terminata la realizzazione dello step di corrente.

```
reset_control();
status_from_position();

if(flag_ciclo == 1)
{
    requested_cmd = CMD_DISENGAGE;
}

start_command();
```

Fig. 5.76: correzione del caso STATUS_ENGAGED della macchina a stati

Rispetto a quanto già presentato in precedenza, si rileva l'aggiunta di un *if* che consente di generare, internamente al *driver* (senza interazione con il *core*) e in automatico, il comando di *engage* del *park-lock*; infatti, se *flag_ciclo* vale 1 (ossia ho premuto il tasto ENG MAIN), a *requested_cmd* viene attribuito il valore *CMD_DISENGAGE* e ciò porterà, per mezzo di *start_command*, alla fase di controllo attivo del motore in disinnesto (*app_status* = *STATUS_DISENGAGE*).

Una volta terminata la fase di disinnesto, lo stato dell'applicazione passa a *STATUS_DISENGAGED* e, dopo l'esecuzione del gradino, il codice da eseguire (fig. 4.7.4) è del tutto simile a quello appena mostrato.

```
reset_control();
status_from_position();

if(flag_ciclo == 1)
{
    requested_cmd = CMD_ENGAGE;
}

start_command();
```

Fig. 5.77: correzione del caso STATUS_DISENGAGED della macchina a stati

Anche in questo caso, se *flag_ciclo* vale 1 viene prodotto un comando interno che determina l'avvio di una fase controllata di innesto. La generazione interna dei comandi a seguito dei gradini di corrente prosegue fino a quando la condizione dell'*if* rimane verificata, ossia fino al momento in cui, premendo il tasto DISENG MAIN, il *flag_ciclo* non viene azzerato.

La figura successiva mostra gli andamenti ciclici di posizione e corrente ottenuti caricando sul *driver* MAIN della centralina il *firmware* definitivo preparato per le prove di fatica in cella climatica coinvolgenti il sistema (*park-lock*, MGU, sensore, centralina, GUI) di cui sono state presentate le operazioni di taratura e di calibrazione del controllo.

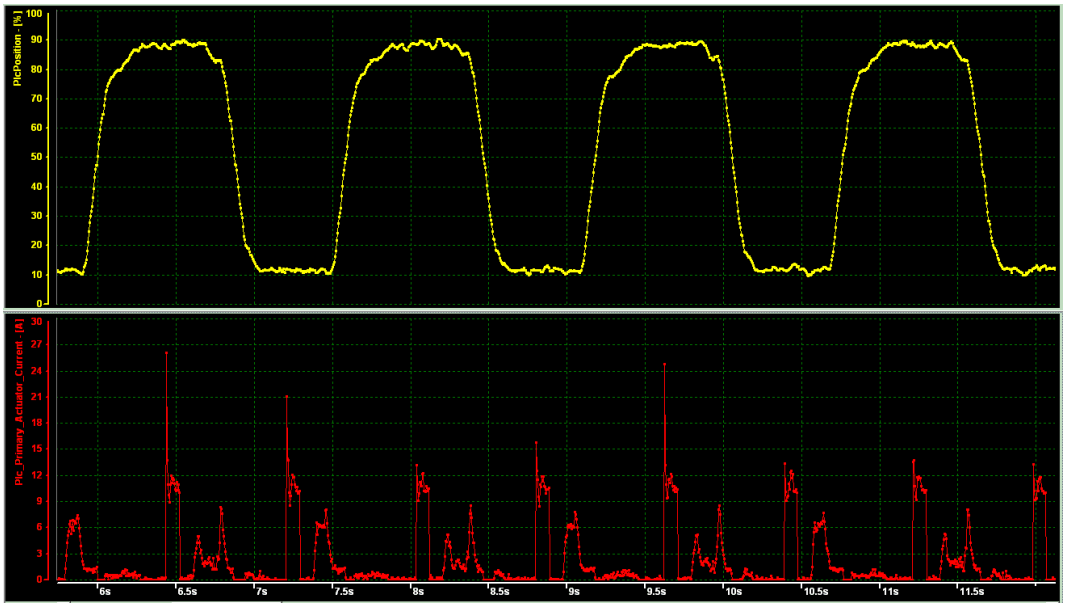


Fig. 5.78: andamento ciclico completo di tutte le implementazioni legate al controllo

CAPITOLO 6

Conclusioni e sviluppi futuri

6.1 Considerazioni conclusive sui risultati delle attività di tesi

Tutti i risultati, sia grafici che numerici, mostrati e commentati nei capitoli precedenti sono nati nel contesto delle fasi di configurazione del sistema di gestione del *park-lock*, le quali sono preliminari alla realizzazione di ciascuna prova di fatica prevista dal DVP. Quindi, siccome la ripetibilità è una delle caratteristiche fondamentali desiderate per il complesso di controllo, sarebbe errato sostenere che il sistema funzioni nel modo corretto basandosi solamente sui dati mostrati fino a questo punto. Si ritiene, perciò, che una possibile strada per valutare correttamente l'efficacia e la robustezza sul lungo periodo del sistema implementato possa essere la seguente:

1. Per prima cosa, bisogna rilevare e osservare attentamente il funzionamento del sistema dopo l'esecuzione di un grande numero di cicli, comparabile con quello totale richiesto da una prova di fatica, al fine di valutarne le performance in generale.
2. In secondo luogo, è opportuno confrontare gli andamenti di posizione e corrente a fine prova con quelli mostrati all'inizio della medesima, poco dopo la calibrazione dell'algoritmo di controllo e la definizione delle soglie all'interno del software del *driver*; tale comparazione è fondamentale per appurare la presenza o meno di derive relative al comportamento dell'attuatore indotte dalla centralina, le quali potrebbero compromettere il risultato e/o la validità della prova.

L'immagine successiva mostra, dunque, come si presentava all'utente l'interfaccia grafica di '*DoubleECU_Project*', predisposta per la gestione di due

centraline contemporaneamente, nel corso dell'esecuzione della prima prova di fatica molla (392766 su N cicli) e della seconda prova di fatica leva (45713 su 300000 cicli).

Fig. 6.1: prove di fatica controllate monitorate in parallelo

In riferimento alla fig. 6.1, la finestra grafica superiore mostra gli andamenti di posizione e corrente per l'attuatore comandato dalla ECU 1, mentre la finestra inferiore segue il dispositivo controllato dalla ECU 2. Come già evidenziato in precedenza, le informazioni trasmesse via CAN dalla ECU 1 non rappresentano l'evoluzione reale delle grandezze fisiche a causa del non funzionamento del *driver* AUX; pertanto, è sensato effettuare considerazioni sull'evoluzione temporale del sistema solo in riferimento agli andamenti mostrati dalla finestra inferiore. Tornando allo scopo dell'analisi, la figura 6.1 permette di osservare che la GUI funziona correttamente, sia dal punto di vista grafico che da quello di esecuzione delle funzioni implementate: le soglie di innesto e disinnesto mostrate a sinistra concordano con le indicazioni di inizializzazione e tutti i campi di conteggio dei cicli vengono aggiornati coerentemente alla prosecuzione della prova.

Osservando l'andamento del *dc* di posizione per la ECU 2 è possibile notare il ripetersi nel tempo in modo ciclico di un'anomalia rispetto alla legge di riferimento che si vuole imitare: tra la fine di una fase di disinnesto e l'inizio della

successiva di innesto è presente un picco locale di *duty cycle*, con ampiezza assoluta pari a circa 7%. La rilevazione di questo fenomeno porterebbe a pensare che l'alberino mosso dalla MGU 'rimbalzi' intorno alla posizione di finecorsa di disinnesto (camma contro cedente dell'arpione), per poi fermarsi sulla stessa; questa condizione operativa risulterebbe problematica per almeno due motivi:

- Per le sollecitazioni impulsive agenti sugli elementi del *park-lock*, gravose per la resistenza meccanica del dispositivo e sicuramente indesiderate anche nel normale funzionamento *on-board*.
- Per la mancanza della caratteristica di irreversibilità del moto che la MGU dovrebbe integrare grazie alla presenza del doppio stadio vite senza fine – ruota a denti elicoidali.

È stato facile verificare (ad esempio, comandando più volte il *disengage* mentre l'attuatore è fermo nella posizione di disinnesto) che i picchi di posizione non sono causati da un movimento reale dell'alberino su cui è calettata la leva ma nascono dall'interferenza elettromagnetica tra l'elevata corrente di attuazione del motorino elettrico e i processi di rilevazione/trasduzione/trasmissione che caratterizzano il funzionamento del sensore Hall a bassi PWM: infatti, il disturbo si sviluppa esattamente in corrispondenza del gradino di corrente erogato a fine disinnesto. L'assenza del disturbo nella posizione di *park-lock* innestato porta a pensare che l'entità dell'interferenza dipenda dalla posizione della leva porta-magnete rispetto agli avvolgimenti e, quindi, sia provocata da un'accentuata distorsione del campo magnetico rilevato dal sensore.

Per quanto riguarda la ripetibilità del controllo del motore, le figure 6.2 e 6.3 rappresentano, in ordine, gli andamenti di posizione e corrente per l'attuatore controllato dalla ECU 2 all'inizio (17 su 300000 cicli) e alla fine (298001 su 300000 cicli) della seconda prova di fatica leva. Per ciascuna condizione vengono riportati due cicli consecutivi completi, per i quali viene riconosciuto come istante iniziale quello in cui gli avvolgimenti cominciano ad assorbire corrente dall'*H-Bridge*. In ciascuna delle due figure l'asse dei tempi è riscalato sulla durata esatta dei soli due cicli considerati e siccome il periodo dei cicli, con ogni probabilità, sarà variato

leggermente tra inizio e fine prova, l'intervallo temporale rappresentato in figura 6.2 è diverso da quello mostrato in figura 6.3; di conseguenza, le due figure successive potranno costituire il riferimento per un confronto sulle ampiezze (quindi sui valori assunti dalle grandezze) ma non sulla durata delle diverse fasi.

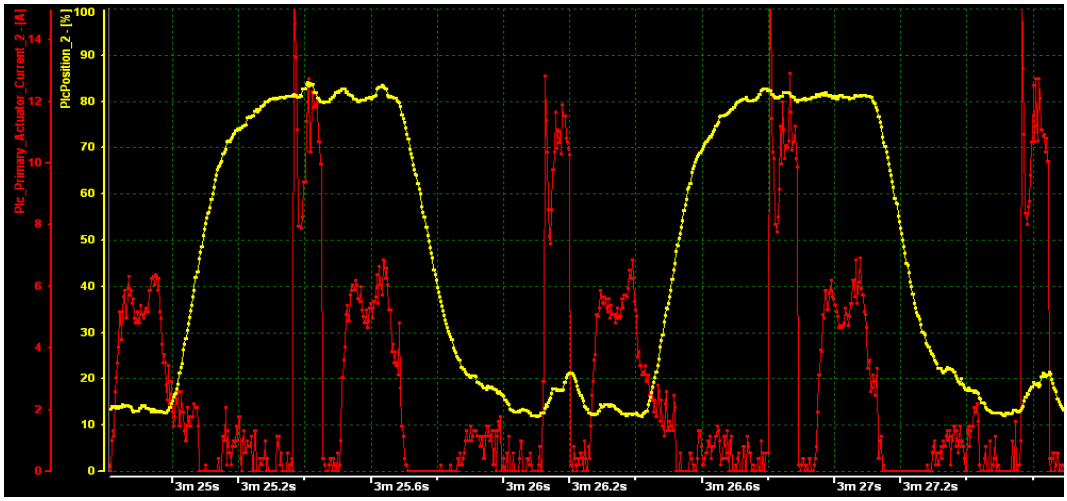


Fig. 6.2: andamenti di posizione e corrente all'inizio di una prova di fatica

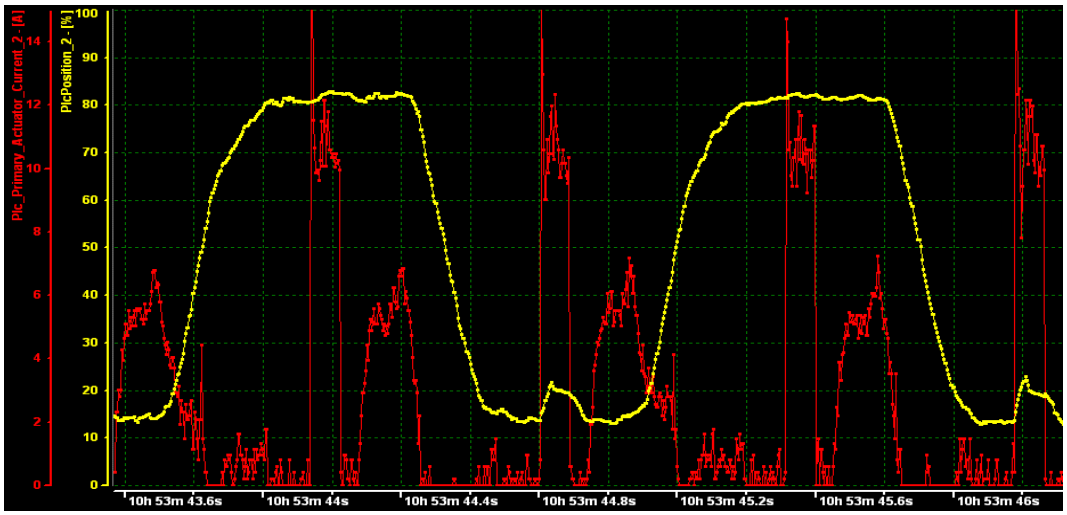


Fig. 6.3: andamenti di posizione e corrente alla fine di una prova di fatica

Guardando le due immagini precedenti, è possibile affermare che le leggi di posizione e corrente realizzate dalla MGU hanno conservato nel tempo le principali caratteristiche qualitative necessarie per assicurare la validità della prova di fatica della leva. In particolare, una corretta sollecitazione della leva porta-magnete richiede sia la presenza del tratto di rallentamento alla fine della corsa di innesto

che quella del gradino di corrente a 10A al termine della medesima fase; è possibile riscontrare entrambe le caratteristiche indicate negli andamenti delle due figure precedenti e questo fatto costituisce una prima testimonianza della ripetibilità del sistema implementato. Si nota ancora che i valori medi di *duty cycle* estremi, corrispondenti alle posizioni limite di innesto e disinnesto, si sono mantenuti pressoché costanti tra l'inizio e la fine della prova, con una variazione massima di circa 1% assoluto; si tratta di un fatto interessante poiché conferma la robustezza del trasduttore di posizione e che la leva porta-magnete non ha subito cedimenti strutturali macroscopici in fase di test.

Dal momento che le figure 6.2 e 6.3 non permettono di paragonare sul piano temporale le leggi considerate, nell'immagine successiva ciascuna coppia di andamenti analoghi (posizione e corrente) è stata trasposta in un singolo grafico e le curve sono state traslate nel tempo così da sovrapporle esattamente in corrispondenza dell'istante iniziale del primo dei due cicli ($t = 0s$).

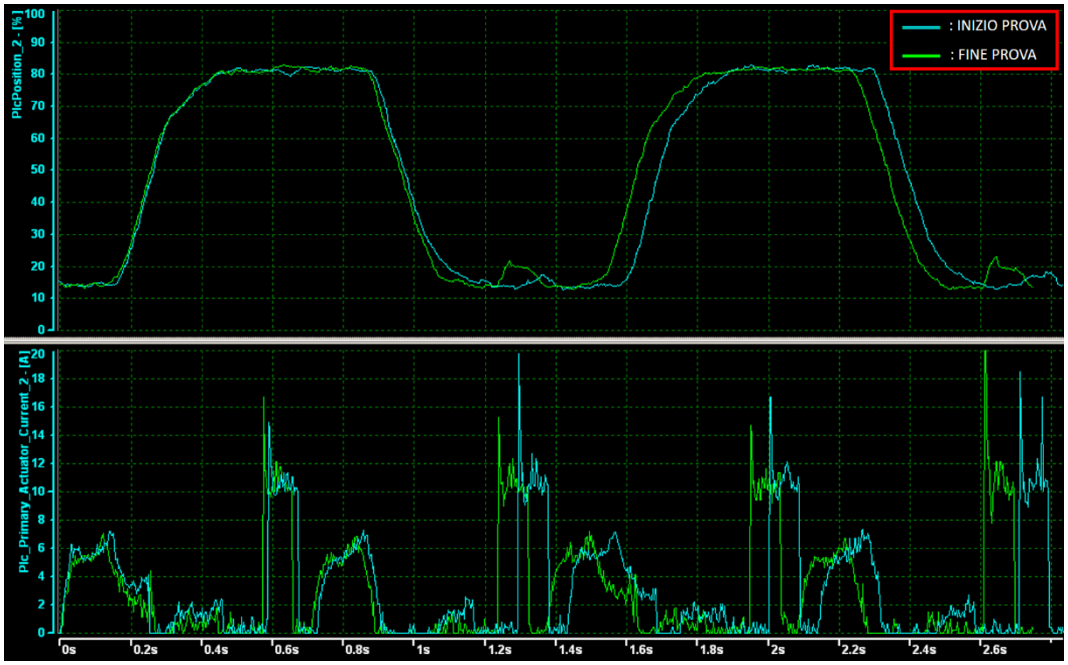


Fig. 6.4: confronto temporale tra i cicli di attuazione a inizio e fine prova

In riferimento alla figura 6.4, è evidente la presenza di uno sfasamento temporale tra le curve; infatti, tra la condizione iniziale della prova (linea azzurra) e quella finale (linea verde) si è verificato un ridotto ma apprezzabile incremento

della frequenza di esecuzione dei cicli. Esaminando attentamente il primo ciclo di posizione a sinistra si riscontra che, mentre in entrambi i casi l'alberino di ingresso segue approssimativamente la stessa legge di innesto (fig. 6.5), nella fase di disinnesto si muove più rapidamente nel secondo dei due, ossia a valle dell'esecuzione di oltre 295000 cicli (fig. 6.6, le due curve sono sincronizzate in corrispondenza dell'inizio dell'assorbimento di corrente per eliminare il piccolo ritardo sviluppato nella fase di innesto). Pertanto, il segnale di posizione impiega meno tempo per raggiungere la soglia di disinnesto impostata nei sorgenti del *driver*, il gradino di corrente viene generato prima e così cresce l'anticipo delle curve in verde rispetto a quelle in azzurro.

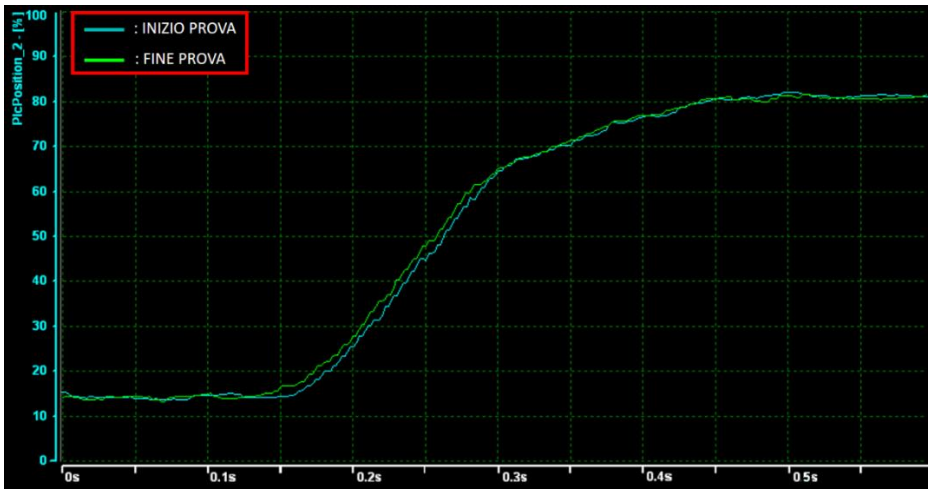


Fig. 6.5: sovrapposizione delle leggi di innesto di inizio e fine prova

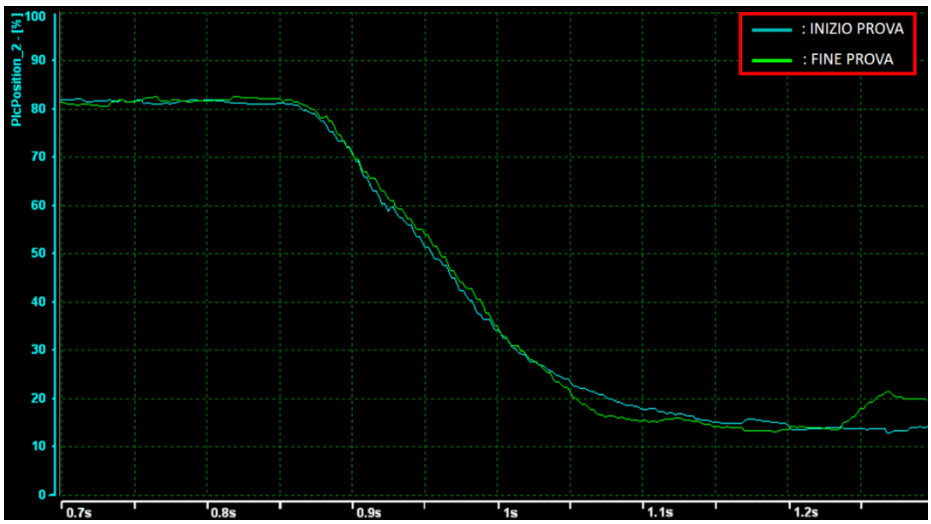


Fig. 6.6: sovrapposizione delle leggi di disinnesto di inizio e fine prova

L'analisi di fig. 6.6 evidenzia che il discostamento tra le leggi di *disengage* si impone nel tratto di avvicinamento alla posizione di finecorsa, percorrendo il quale il sistema si muove più rapidamente a fine prova. Si ricorda che la legge di riferimento presentata nel capitolo 3 e che si desidera riprodurre nelle fasi di calibrazione prevede che l'attuatore rallenti a fine disinnesto (come avviene nella curva di posizione azzurra di fig. 6.4) per evitare urti indesiderati e applicare il gradino di corrente a 10A ad un sistema che non può compiere un'ulteriore rotazione nel medesimo verso. La perdita di questa caratteristica nel moto è da imputare alla diminuzione nel tempo del carico che si oppone alla rotazione dell'alberino nelle posizioni finali di disinnesto; tale calo di coppia resistente è causata dalla variazione di fattori sensibili alla durata del moto continuativo del sistema: ad esempio, l'aumento della temperatura e il generale rodaggio del meccanismo determinano la riduzione degli attriti dinamici e viscosi interni alla MGU e di quelli che agiscono tra i vari componenti del *park-lock*.

La tabella 6.1 riporta i tempi caratteristici di innesto e disinnesto (calcolati con e senza il ritardo di spunto), oltre che il periodo di esecuzione dei cicli, per le condizioni di inizio e fine prova di fatica leva mostrate in figura 6.4; i valori indicati derivano dalla media delle quantità relative ai due cicli considerati per ciascun caso. Sono proposte, inoltre, le durate tipiche della legge del moto di riferimento, così da poter sviluppare delle considerazioni aggiuntive. Si ricorda che con la locuzione 'ritardo di spunto' si fa riferimento all'intervallo di tempo che intercorre tra l'inizio dell'assorbimento di corrente da parte del motorino e l'istante in cui si osserva l'avvio del moto dell'alberino; tale ritardo è indotto dagli attriti statici, dalle rigidezze e dalle inerzie (quindi dalla dinamica) propri del sistema.

CASO	TEMPI DI INNESTO [ms]			TEMPI DI DISINNESTO [ms]			PERIODO [s]
	Solo corsa	Spunto + corsa	Ritardo spunto	Solo corsa	Spunto + corsa	Ritardo spunto	
Riferimento	280	360	80	380	470	90	/
Inizio prova	301	468	167	361	511	150	**1.417**
Fine prova	313	446	133	313	456	143	**1.376**

Tab. 6.1: tempi caratteristici per la valutazione della ripetibilità

Allo scopo di facilitare per via grafica la comprensione dei dati inseriti nella tabella precedente, l'immagine 6.7 riporta un istogramma a barre per il confronto tra i tre casi sulla base dei tempi tipici già nominati.

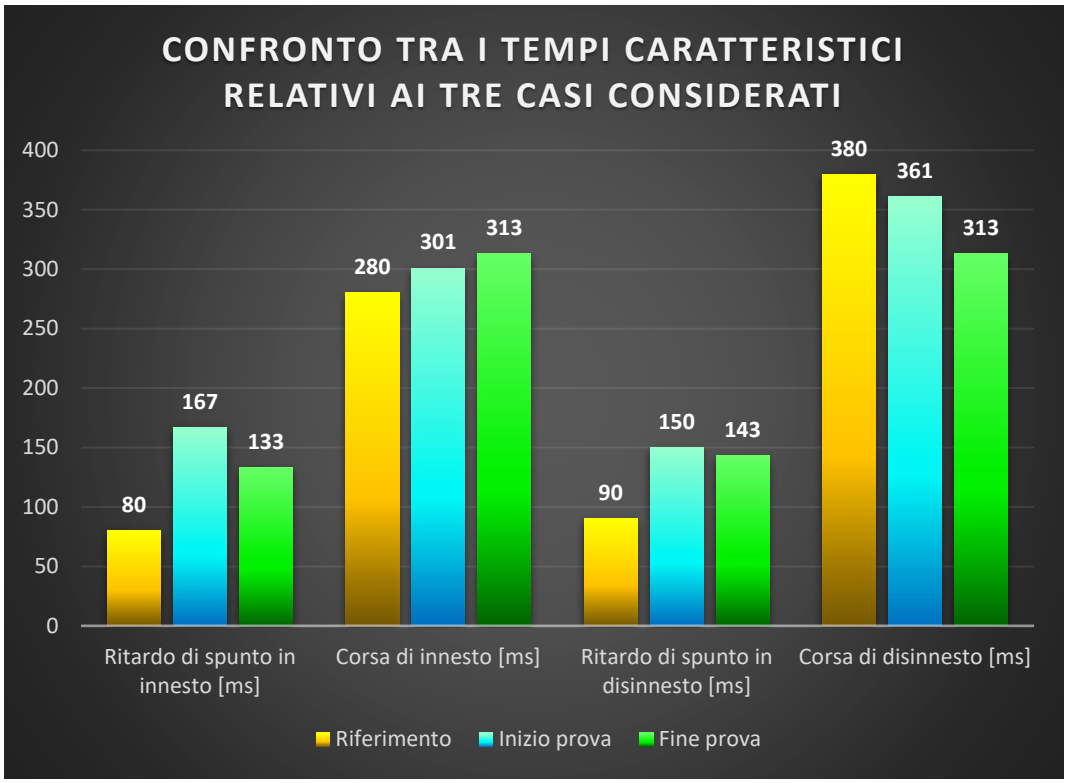


Fig. 6.7: istogramma dei tempi caratteristici per la valutazione della ripetibilità

Relativamente alla figura precedente, si nota subito la differenza tra la legge di riferimento e le altre per quanto riguarda l'entità del ritardo di spunto: le corse di innesto e disinnesto di inizio e fine prova presentano un tempo di indugio (legato ad attriti, inerzie, ecc.) più lungo di 50-60ms rispetto agli andamenti originali. Questo fenomeno non può essere attribuito ad un peggioramento funzionale del sistema di controllo *park-lock*, poiché è presente fin dalle fasi di calibrazione dell'algoritmo di controllo; esso, infatti, è dovuto alle diverse caratteristiche elettroniche (in particolari del ponte H) ed informatiche della centralina di riferimento rispetto a quelle delle due ECU identiche riprogrammate: il primo dispositivo, infatti, riesce a fornire una corrente elevata (10A) più velocemente di quanto non sappiano fare gli altri due, mostrando così una performance migliore in termini di prontezza. D'altronde non mi è stato richiesto di provare ad allineare i

due sistemi configurati a quello di riferimento anche in termini di ritardo, quindi il divario evidenziato non svaluta il risultato del lavoro svolto.

Per quanto riguarda le fasi di movimento della MGU (quindi le attuazioni al netto degli spunti), sia per la corsa di innesto che per quella di disinnesto si evidenzia uno scarto di circa 20ms (in valore assoluto) tra la durata del moto rilevata nella fase iniziale e quella di riferimento per la calibrazione. Queste differenze in parte sono aleatorie e in parte sono dovute alla calibrazione imperfetta del controllo; però, in quanto sono relativamente ridotte, non minano l'idoneità delle condizioni operative ai fini della validazione della leva. In secondo luogo, limitando il confronto ai tempi caratteristici di inizio e fine prova, l'istogramma conferma quanto già affermato in precedenza: l'attuazione di *engage* diviene leggermente più lenta (di circa 10ms) mentre quella di *disengage* notevolmente più rapida (di circa 50ms).

La tabella 6.2 riporta le variazioni percentuali delle quantità di interesse (tempi di ritardo e di corsa, periodo di esecuzione dei cicli) rispetto a quanto rilevato all'inizio della prova di fatica leva, così da ricapitolare i dati più significativi per la valutazione della ripetibilità del sistema di controllo *park-lock* e fornire uno spunto per le considerazioni conclusive.

QUANTITÀ	VARIAZIONE [%]
Ritardo spunto di innesto	-20%
Durata corsa di innesto	+4%
Ritardo spunto di disinnesto	-5%
Durata corsa di disinnesto	-13%
Periodo di esecuzione dei cicli	-3%

Tab. 6.2: variazioni percentuali delle quantità di interesse

In riferimento alla tabella 6.2, si ritiene importante riportare le seguenti osservazioni:

- Come già sottolineato più volte, tra l'inizio e la fine della prova di fatica leva la durata della corsa di aggancio aumenta leggermente (+4%) mentre quella della corsa di sgancio diminuisce notevolmente

- (-13%), a causa della diminuzione delle coppie agenti in opposizione alla rotazione dell'alberino al termine della corsa di disinnesto.
- Si registra, inoltre, la riduzione del ritardo legato allo spunto sia per la corsa di innesto (-20%) che per quella di disinnesto (-5%); questo fenomeno testimonia l'aumento della reattività del sistema quando deve essere messo in moto da fermo e si può associare alla riduzione degli attriti (soprattutto statici) ostacolanti il moto degli elementi del *park-lock*, a sua volta determinato dalle conseguenze meccaniche e termiche di un uso continuativo del dispositivo e della MGU.
- Infine, il periodo di esecuzione dei cicli diminuisce (quindi la frequenza aumenta) verso la fine della prova (-3%), coerentemente ai fenomeni e alle derive precedentemente evidenziate.

In conclusione, la ripetibilità associata all'azione del sistema configurato può essere valutata positivamente, ma solo in relazione alla specifica applicazione considerata. Le prove di fatica che sono state gestite mediante le due centraline non richiedevano un accurato controllo in posizione della rotazione dell'alberino affinché i risultati potessero essere considerati significativi ai fini della validazione. La prova di fatica leva, infatti, domandava la ripetibilità delle sollecitazioni agenti sulla leva, pertanto del rallentamento a fine innesto e del gradino di corrente; quella di fatica molla necessitava la massima torsione che la molla può subire nel normale utilizzo a bordo veicolo, garantita dal raggiungimento del finecorsa di disinnesto e dal contatto 'dente su dente' tra arpione e rocchetto. Qualora i requisiti di ripetibilità sulla legge di posizione fossero più stringenti e il loro soddisfacimento vincolasse l'accettabilità del risultato, allora sarebbe opportuno (se non necessario) modificare la strategia di controllo poiché quella implementata non è sufficientemente robusta.

6.2 Possibili miglioramenti del sistema e sviluppi futuri

Come è stato osservato nelle conclusioni, l'algoritmo di controllo in anello aperto con interpolazione della tensione di comando dal feedback di posizione ha permesso di realizzare le prove di fatica molla e leva del *park-lock* con ripetibilità sufficiente da ritenere i risultati dei test rappresentativi per le necessità di

validazione. Nonostante la valutazione positiva dell'efficacia del sistema di controllo, il confronto di figura 6.4 ha messo in evidenza una certa variabilità nel tempo sia delle leggi di spostamento seguite dall'alberino del *park-lock* che dei valori di *duty cycle* corrispondenti alle posizioni di finecorsa. Tali deviazioni di comportamento nel tempo rispetto a quanto rilevato in fase di set-up potrebbero, però, non essere accettabili per la validazione del *park-lock* in relazione ad una nuova applicazione; ad esempio, se si desiderasse montare il dispositivo di sicurezza su una nuova trasmissione sviluppata dalla *Oerlikon Graziano* per un cliente, quest'ultimo potrebbe esigere dei vincoli operativi più stringenti per le medesime prove di fatica precedentemente considerate.

Pertanto, per essere pressoché certi che il sistema di controllo e misura della posizione e, per certi versi, della corrente fornisca risultati di validità incontestabile, sarebbe appropriato implementare sui *driver* delle centraline un algoritmo di controllo ad anello chiuso, più affidabile e preciso di uno in *open loop*. Si ricorda che la strategia di controllo ad anello aperto è stata scelta per esigenze di controllo della tensione applicata agli avvolgimenti e, soprattutto, per questioni di premura nel definire una prima soluzione funzionante al fine di eseguire prove urgenti sul *park-lock*; terminate le prove per il DVP della trasmissione per veicolo elettrico bisognerà impostare un nuovo algoritmo di controllo che semplifichi e renda più robusta la validazione del dispositivo per applicazioni future.

Il progetto sorgente originale dei *driver* delle centraline prevedeva un algoritmo con due anelli chiusi: uno per la posizione e uno per la corrente. Si ritiene che un controllo in *closed loop* sulla posizione dell'alberino dell'attuatore non sarebbe ottimale per l'applicazione per due motivi:

1) Perché i vincoli imposti sulla legge del moto del *park-lock* per i test di fatica (soprattutto quello della leva) del DVP non riguardavano tanto la posizione quanto la velocità di rotazione dell'albero; in particolare, era richiesto che all'inizio della rotazione (sia in *engage* che in *disengage*) il moto fosse il più rapido possibile e che, raggiunta

un certo angolo, l'alberino rallentasse in modo da evitare un urto troppo forte in corrispondenza del finecorsa.

2) Perché la legge di controllo viene applicata tra due soglie di posizione identificate nella fase di configurazione: fuori dal campo compreso tra i due limiti di innesto e disinnesto il controllo viene azzerato. È stato osservato, però, che tra l'inizio e la fine delle lunghe prove di fatica i valori restituiti dal sensore Hall per la rotazione minima e massima della leva cambiano; pertanto, si può affermare che basare la durata della fase di controllo sul valore di feedback di posizione è sbagliato e potrebbe portare ad effetti indesiderati.

Sulla base delle considerazioni precedenti, l'implementazione di un controllo in anello chiuso sulla velocità, dotato o meno di un *loop* interno di controllo corrente, si pone come la scelta più sensata [35]. Il set di velocità potrebbe essere costituito da un primo gradino di valore elevato e, in seguito, uno inferiore, da applicare quando l'albero raggiunge una certa posizione o dopo un certo intervallo di tempo dall'inizio della fase di controllo. La condizione ottimale di ottenimento del feedback di velocità consisterebbe nella sua rilevazione con un trasduttore apposito, il cui segnale verrebbe inviato al *driver* (che deve essere configurato per convertirlo e memorizzarlo); per semplicità, però, si potrebbe ricavare la velocità derivando numericamente la posizione e mediandola dinamicamente per ridurre gli inevitabili disturbi. Il *loop* chiuso di velocità dovrebbe, infine, comprendere un contributo integrativo; in questo modo il riferimento per il comando (o per il *loop* di corrente) rimarrebbe correttamente diverso da zero anche quando l'errore sulla velocità si annulla. L'immagine seguente schematizza una possibile strategia di controllo che potenzialmente incrementerebbe l'efficienza e l'efficacia del sistema di gestione delle prove di fatica (fig. 6.8).

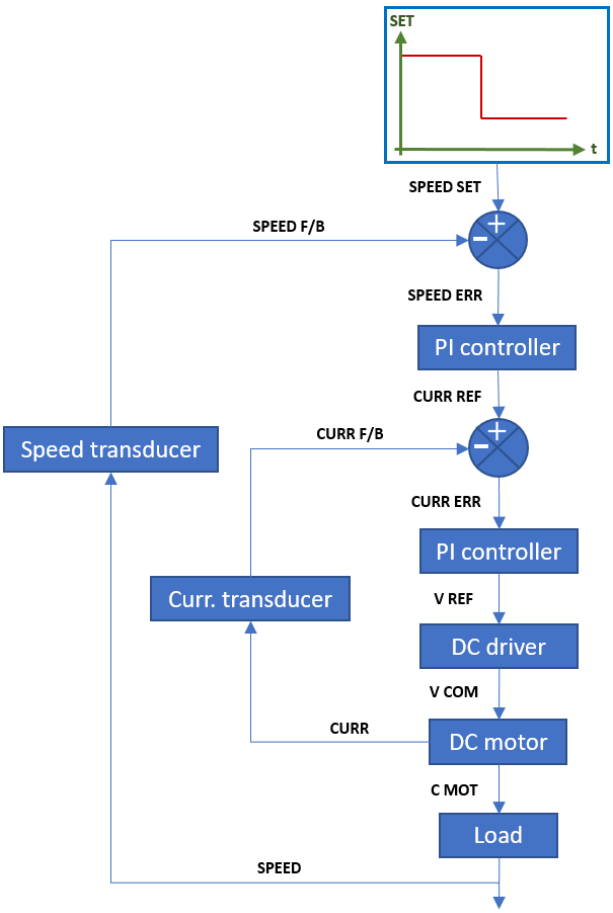


Fig. 6.8: algoritmo di controllo della velocità in anello chiuso

Inoltre, converrebbe modificare i sorgenti dei *driver* in modo da aggiungere delle funzioni per la definizione dinamica e non più statica delle soglie di innesto e disinnesto, le quali delimitano le fasi di esecuzione dell'algoritmo di controllo del moto dell'attuatore. Ad esempio, i *dc* di finecorsa potrebbero essere identificati come quelli restituiti dal sensore Hall dopo il gradino di corrente, oppure ci si potrebbe basare sulla rilevazione prolungata di velocità nulla. Si evidenzia che tutte le affermazioni precedenti che hanno portato ad individuare nell'algoritmo di controllo della velocità una soluzione ottimale valgono per la specifica applicazione di validazione considerata; qualora si desiderasse testare con la medesima centralina un *park-lock* differente (nella struttura e/o nel funzionamento) o quello di prima venga impiegato su una trasmissione differente, la strategia di attuazione andrebbe rivista in base alle condizioni operative richieste per test di fatica del DVP.

Al fine di determinare la tipologia di controllo migliore e di stimarne i parametri ottimali, è possibile utilizzare come punto di partenza il modello matematico a blocchi mostrato in fig. 6.9, realizzato dall'ing. Scalici con l'ausilio del programma *LMS Imagine.Lab Amesim* (detto anche semplicemente *Amesim*).

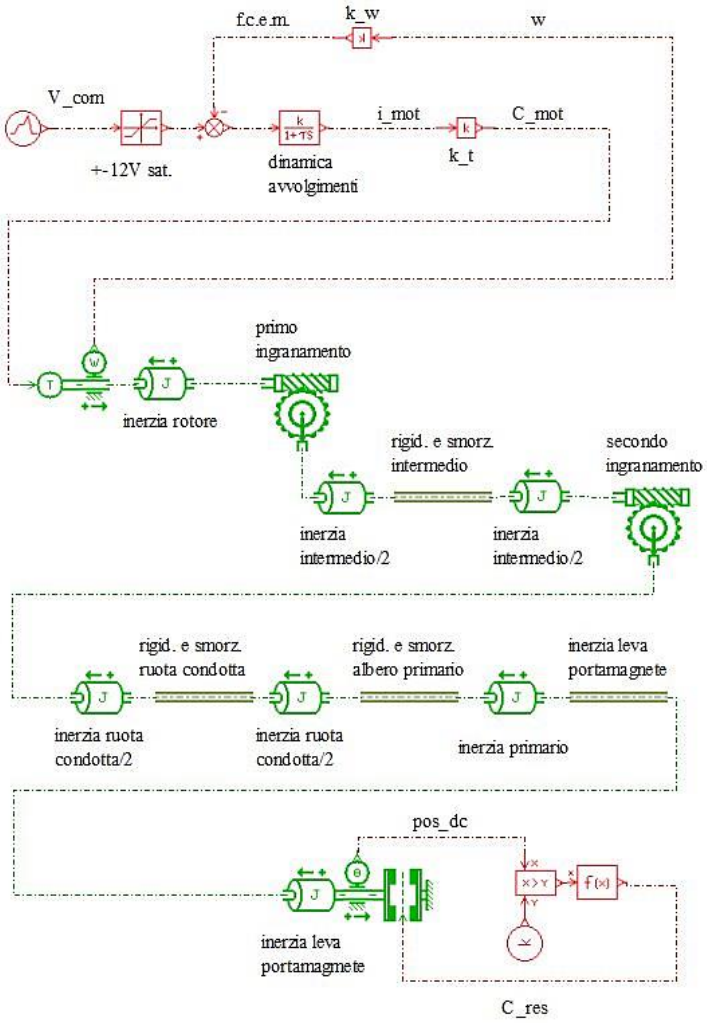


Fig. 6.9: modello Amesim del park-lock senza logica di controllo

Amesim è un ambiente di simulazione dinamica unidimensionale prodotto da *Siemens PLM Software*; esso permette di modellare sistemi multifisica articolati più rapidamente e intuitivamente di quando non consenta un programma come *MATLAB*/*Simulink*. Il sistema fisico viene rappresentato con un insieme di blocchi tratti da librerie tematiche: ciascuno di essi rappresenta un certo componente (più o meno complesso) e viene caratterizzato con un set di equazioni che ne rappresentano i fenomeni fisici di interesse; i blocchi sono collegati l'uno con l'altro

a rappresentare un'interazione input/output che coinvolge una o più variabili fisiche incognite. Definita la struttura del modello, la compilazione definisce il set di equazioni differenziali che descrive in modo completo il sistema e produce un programma per risolverlo; esso viene, infine, risolto discretamente nel tempo per definire così l'evoluzione dinamica di tutte le grandezze fisiche incognite.

Nel modello di figura 6.9 sono stati modellati tutti i componenti elettrici e meccanici del *park-lock*: il motorino elettrico, gli ingranaggi dell'attuatore, l'alberino di ingresso, etc. Ogni elemento è stato caratterizzato con i parametri fisici richiesti dalle equazioni che lo descrivono: la resistenza e l'induttanza per gli avvolgimenti elettrici, l'inerzia per le masse rotanti, rapporto di riduzione e rendimento meccanico per gli ingranamenti e così via. La coppia di reazione della molla è stata inclusa in quella resistente al moto dovuta all'attrito e al peso proprio dell'arpione, valutata sperimentalmente ed espressa come funzione dell'angolo di rotazione dell'alberino. Si osserva che modellare la coppia resistente come una funzione del solo angolo costituisce una significativa approssimazione: per migliorare l'affidabilità del modello sarebbe opportuno effettuare ulteriori rilevazioni per correlare la coppia almeno anche alla velocità di rotazione.

Il modello *Amesim* di figura 6.9 è stato predisposto per ricevere in ingresso una legge temporale della tensione di comando degli avvolgimenti elettrici, definita dall'utente in fase di parametrizzazione: esso rappresenta, infatti, solo i componenti elettrici e meccanici del *park-lock*. Qualora, però, si desiderasse studiare tutto il sistema costituito dal dispositivo di sicurezza e dalla sua centralina di controllo, il software *Amesim* offre delle librerie specifiche che permettono di simulare anche le caratteristiche elettroniche (ritardo di calcolo, discrezione di funzionamento, …) e logiche (algoritmo di controllo del moto, limitazione di corrente, …) del complesso in esame. Pertanto, la realizzazione di simulazioni basate su modelli completi e caratterizzati con cura, aderenti al comportamento reale del sistema, darebbe facoltà di confrontare tra loro varie architetture di controllo (anello chiuso di posizione, di velocità, …), di individuare tra di esse la migliore e, addirittura, di stimarne i relativi parametri ottimali (da confermare con prove sperimentali a seguito dell'implementazione sul *driver*).

Bibliografia

[1] Standard SAE J915, “Automatic Transmissions – Manual Control Sequence”, Society of Automotive Engineers, 2017.

[2] Standard SAE J2208, “Park Standard for Automatic Transmissions”, Society of Automotive Engineers, 2016.

[3] Orthwein W. C., “Clutches and Brakes: Design and Selection”, 2 ed., CRC Press, London, 2004.

[4] Genta G., Morello L., “The Automotive Chassis: Volume 1: Components Design”, ed. 2009, Springer, Torino, 2009.

[5] Genta G., Morello L., “The Automotive Chassis: Volume 2: Systems Design”, ed. 2009, Springer, Torino, 2009.

[6] Kudo K., Wakabayashi Y., “Simulation of a Pawl and Gear Parking System for an Automatic Transmission”, JSAE Annual Congress Proceedings, v. 17, n. 1, pp. 4-26, 1996.

[7] Yamaoka T., Kobayashi A., Nogami S., Omoto H., “Development of Parking Lock System Behaviour Analysis”, Honda R&D Technical Review, v. 18, n. 1, pp. 2-40, 2006.

[8] Jeyakumaran J., Zhang N., “Dynamic Analysis of an Automatic Transmission Parking Mechanism”, ICTAM conference at University of Adelaide, Australia, v. Conference Proceeding, pp. 1-2, 2008.

[9] Gierer G., Rühringer U., Mittleberg J., “Parking brake, notably for automatic transmission systems of motor vehicles”, U. S. Patent 6,471,027 B1, 29 Oct., 2002.

[10] Kanehisa T., Yamauchi Y., Koyama T., "Parking lock device and method for automatic transmission", U. S. Patent 6,401,899 B1, 25 Nov., 1999.

[11] Steinhauser K., Schmidt T., Wagner J., Kiebler H., "Parking lock system", U. S. Patent 2001/0198190, 18 Aug, 2011.

[12] Bauer M., Gieles W., "Device for operating parking lock", U. S. Patent 2012/0160631, 28 Jun., 2012.

[13] Michael J., Ehrmaler R., Neuner J., "Motor vehicle having an automatically shifted transmission", U. S. Patent 5,919,112, 6 Jul., 1999.

[14] Barton E. A., Fong. W. R., Pinkley G. A., Pearce E. M. Jr., Morisawa K., Kaneko K., Tanaka K., "Park lock for narrow transmission", U. S. Patent 8,881,883 B2, 11 Nov., 2014.

[15] Ferraresi C., Raparelli T., "Meccanica applicata", 3 ed., CLUT, Torino, 2007.

[16] Jacazio G., Piombo B., "Meccanica Applicata alle Macchine", v. 1, 2 ed., Levrotto & Bella, Torino, 1999.

[17] *Treccani.it*, s.v., "centralina", consultato il 2 settembre 2018, http://www.treccani.it/vocabolario/centralina/

[18] Guo H., "Automotive Informatics and Communicative Systems: Principles in Vehicular Networks and Data Excange", 1 ed., Information Science Reference, Hershey, 2009.

[19] Marshall C. E., Goates E. L., "Integrated electronic control of pawl-gear park function of an automatic transmission", U. S. Patent 5,696,679, 9 Dec., 1997.

[20] Standard ISO 26262, "Road vehicles - Functional safety", International Standard Organization, 2011.

[21] STMicroelectronics, *STM32F205xx STM32F207xx*, 15818 datasheet, October 2012 (rev. 9).

[22] Texas Instruments, *DRV8701EVM User's Guide*, SLVUAG3 user's guide, March 2015.

[23] Texas Instruments, *DRV8701 Brushed DC Motor Full-Bridge Gate Driver*, SLVSCX5B datasheet, March 2015 (rev. July 2015).

[24] Texas Instruments, *Mixed Signal Microcontroller*, SLAS635J datasheet, April 2011 (rev. May 2013).

[25] Texas Instruments, *Understanding the I2C Bus*, SLVA704 application report, June 2015.

[26] *Treccani.it*, s.v., "codice sorgente", consultato il 23 settembre 2018, http://www.treccani.it/enciclopedia/codice-sorgente_%28Enciclopedia-della-Scienza-e-della-Tecnica%29/

[27] IAR Systems, *IAR Embedded Workbench IDE User Guide*, UARM-16 user guide, July 2009.

[28] Texas Instruments, *Code Composer Studio™ v7.x for MSP430™ - User's guide*, SLAU157AP user's guide, May 2005 (rev. November 2017).

[29] *Wikipedia.org*, "Processo (informatica)", consultato il 24 settembre 2018, https://it.wikipedia.org/wiki/Processo_(informatica)

[30] Deitel P. J., Deitel H. M., *C.*, "Corso completo di programmazione", 4 ed., Apogeo, Torino, 2010.

[31] Viktorov V., Colombo F., "Automazione dei sistemi meccanici – Corso di base", IV ed., CLUT, Torino, 2016.

[32] VOCIS Driveline Controls Ltd., *CANape GT Manual*, 2010.

[33] Vector Informatik GmbH, *CANape CASL*, user's guide, 2015 (v. 1.2).

[34] M. H. Rashid, "Elettronica di Potenza", Hoepli, Milano, 2005.

[35] Profumo, "Azionamenti elettrici I", Pitagora Editrice, Bologna, 1998.

[36] Lehman J., (2013), *Closed Loop vs Open Loop Control*, estrapolato da https://www.dataforth.com.

[37] Sorli M., Quaglia G., "Meccatronica. Le basi della meccatronica. Struttura dei dispositivi di attuazione controllata. Caratteristiche statiche e dinamiche degli strumenti", vol.1, Epics, Torino, 2017.

[38] Brunelli A., *Strumentazione di misura e controllo delle applicazioni industriali – vol.1: caratteristiche generali sensori e trasduttori*, vol.1, GISI Gruppo Imprese Strumentazione Italia, Milano, 1990.