UNIVERSITA' DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria Industriale DII

Corso di Laurea Magistrale in Ingegneria dell'Energia Elettrica

Rete di comunicazione per un sistema di monitoraggio distribuito della carica e scarica di una batteria

Relatore: Prof. Bertoluzzo Manuele

Laureando: Beghetto Filippo

Anno Accademico 2019/2020

Indice

Sommario

Al giorno d’oggi viene posta una particolare attenzione all’incremento dell’efficienza energetica e all’utilizzo di fonti energetiche non inquinanti; nel settore automotive ciò si traduce nell’impiego sempre più comune di veicoli elettrici alimentati da batterie agli ioni di Litio. Affinché questa tipologia di accumulatori garantisca una lunga durata di vita e alte prestazioni si necessita di un sistema di monitoraggio che rilevi le condizioni operative di ogni singola batteria e ne gestisca il funzionamento in fase di carica e scarica. A tal proposito in questa trattazione si progettano e sviluppano due possibili reti di comunicazione di un sistema di monitoraggio distribuito, in un primo momento sfruttando la comunicazione secondo il protocollo CAN, e successivamente sviluppando un sistema di comunicazione radio. Si illustrano non solo le fasi di progettazione dei circuiti hardware e della programmazione software, ma anche i test sperimentali eseguiti per validare i due sistemi di monitoraggio distribuito implementati, confrontandoli infine tra loro per capire quale possa essere il più efficiente e il più adatto per un’applicazione aziendale.

Abstract

Nowadays, particular attention is paid to increasing energy efficiency and the use of non-polluting energy sources; in the automotive sector this translates to the increasingly common use of electric vehicles powered by lithium-ion batteries. In order for this type of accumulator to guarantee a long life and high performance, a monitoring system is required that detects the operating conditions of each individual battery and manages its operation during charging and discharging. For this scope, in this discussion two possible communication networks of a distributed monitoring system are designed and developed, at first using the communication according to the CAN protocol, and subsequently developing a radio communication system. This discussion illustrates the design phases of hardware circuits and software programming, as well as the experimental tests performed to validate the two distributed monitoring systems implemented, comparing them to understand which could be the most efficient and the most suitable for a business application.

1. Introduzione

Lo scopo ultimo di questa trattazione è quello di ideare e realizzare un sistema di gestione della batteria (BMS, Battery Management System) operante nella sua completezza, considerandone una sua futura applicazione nel settore automotive e quindi in stretta relazione con i veicoli elettrici alimentati da accumulatori (BEVs, Battery Electric Vehicles). Il punto di partenza di tale progetto coincide con la tesi di Laurea Magistrale in Ingegneria dell'Energia Elettrica presso l'Università degli Studi di Padova di Samuel Matrella, intitolata "Studio di un sistema per il monitoraggio della carica degli accumulatori di un veicolo elettrico" [1]. Il collega nella sua trattazione ha progettato e realizzato un circuito in grado di monitorare le principali grandezze fisiche che caratterizzano la ricarica di un sistema di accumulatori composto da quattro batterie al piombo acido poste in serie tra loro; questa tesi ne rappresenta quindi in qualche modo la continuazione naturale nell'ottica di voler realizzare un sistema di gestione della batteria completo. L'elaborato di Samuel Matrella ha portato ad ottenere un BMS allo stato "primordiale": funziona correttamente per quanto concerne l'acquisizione in real time dei valori di tensione e corrente di ogni accumulatore ma non realizza l'implementazione della successiva comunicazione tra i diversi microprocessori impiegati (uno per ogni pacco batteria da monitorare) e l'interfacciamento finale tra un nodo Master, che gestisce tutti i dati raccolti, e un PC che fornisce una visualizzazione dei risultati ottenuti. Per maggior chiarezza, pensando il BMS (Battery Management System) qui sviluppato destinato all'utilizzo in un veicolo elettrico (settore automotive utilizza esclusivamente accumulatori Li-ion come sorgenti di energia elettrica, la considerazione in questa tesi di batterie al piombo acido è determinata dai dispositivi disponibili nel laboratorio di Sistemi Elettrici per l'Automazione e la Veicolistica del Dipartimento di Ingegneria Industriale dell'Università di Padova), esso dovrebbe monitorare e gestire verosimilmente decine di moduli, a loro volta costituiti da centinaia se non migliaia di celle (in base alla capacità nominale complessiva ma anche alla casa costruttrice e dalla tipologia). Ecco che allora dal punto di vista progettuale diventa fondamentale non solo il circuito di acquisizione e misura dei parametri caratteristici di ogni batteria, realizzato dal collega Samuel Matrella, ma anche la realizzazione di un canale di comunicazione integrato al veicolo stesso che permetta di sfruttare i dati raccolti; in primo luogo pensando al navigatore accessoriato in un veicolo, tensione e corrente di ogni singolo pacco batteria possono essere usate per determinare lo SOC (State Of Charge) dell'intera sorgente di energia elettrica e quindi fornire un'indicazione visiva al guidatore sull'autonomia a disposizione, in secondo luogo un confronto degli andamenti di tali grandezze tra i vari pacchi durante un ciclo di ricarica potrebbe fornire indicazioni preziose per correggere o adattare le modalità di ricarica stessa, questo presupponendo la possibilità di uno scambio di informazioni tra BMS e caricabatteria. Prima di procedere con l'analisi del progetto realizzato per questa tesi, è necessario introdurre ed approfondire diverse nozioni teoriche riguardanti BMS, kit CY8CKIT-059, CAN bus e comunicazione UART, illustrate qui nel seguito e fondamentali per una comprensione accurata delle diverse fasi progettuali affrontate.

2. BMS-Battery Management System

I BMS sono dei sistemi elettrici di gestione e monitoraggio di cui sono equipaggiati gli accumulatori agli ioni di Litio che svolgerono diverse funzioni [2]; in particolare si occupano di monitorare e controllare le singole celle che compongono un pacco batteria Li-ion per prevenire:

- la carica a polarità inversa
- la sovraccarica e la sotto scarica, che intaccherebbero altrimenti la capacità della cella e di conseguenza anche la durata di vita del pacco batteria
- il surriscaldamento e lo scoppio, garantendo una maggiore longevità e sicurezza
- lo sbilanciamento dello stato di carica delle singole celle
- l'alta temperatura, così da far lavorare l'accumulatore in un'area operativa sicura

Nel corso degli anni, in particolare a partire dal primo decennio del ventunesimo secolo, si è assistito ad un sempre crescente sviluppo ed utilizzo di tale tipologia di accumulatori, dotati di elevate densità di potenza ed efficienza energetica oltre che di estesa vita operativa e peso ridotto, nel settore della telefonia mobile, automotive e dei trasporti in generale [3]. Questo contesto ha parallelamente richiesto progressi tecnologici nella progettazione e realizzazione di BMS sempre più complessi. Il progetto sviluppato in questa tesi rientra proprio nel quadro d'interesse del settore automobilistico, in particolare nel mondo dei veicoli elettrici a batterie che, essendo mossi da uno o più motori elettrici, basano completamente il loro funzionamento sull'elettricità fornita dagli accumulatori Li-ion.

Dal punto di vista operativo non tutti questi sistemi di gestione tuttavia mettono a disposizione le stesse funzionalità operative e in tal senso esistono nel mercato tipologie diverse a seconda del loro principio di funzionamento e complessità; è comunque basilare e fondamentale che tali sistemi forniscano una protezione dell'accumulatore cui sono associati, oltre che la determinazione e conseguente visualizzazione del suo State-of-Charge (stato di carica SOC, inteso come livello di carica rimanente, in relazione alla capacità stessa, che in un determinato istante la batteria è in grado di erogare), essendo questi dei requisiti minimi da soddisfare. Il monitoraggio dunque delle prestazioni elettriche di un accumulatore Li-ion si traduce nel successo stesso delle applicazioni che ne fanno uso ed occupa quindi un ruolo fondamentale sia nello sviluppo di tali sistemi che nella ricerca di soluzioni sempre più complesse ed efficaci; tuttavia l'innovazione nel campo dei sistemi di gestione della batteria è frenata ed ostacolata dalla mancanza di un metodo affidabile per calcolare lo SOC, la misura più basica che si può effettuare per una batteria e allo stesso tempo la più complicata da determinare con certezza.

2.1.Schema di principio di un BMS

La prima funzionalità di un BMS deve essere il monitoraggio della tensione delle singole celle, fondamentale per determinare poi lo stato generale del pacco batteria; questo poiché tutte le celle sono caratterizzate da una finestra di tensione operativa che deve essere rispettata per garantirne un corretto funzionamento e parallelamente una durata di vita prolungata. Solitamente per un monitoraggio ed una protezione più efficacie si implementano, oltre alla rilevazione del voltaggio, dei sistemi per la misura della corrente o temperatura così da poter ottenere anche una stima più accurata dello stato di funzionamento del pacco batteria. In generale un sistema di gestione della batteria può essere composto da numerosi blocchi funzionali che possono essere più o meno raggruppati tra loro a seconda delle caratteristiche e complessità del prodotto finale che si vuole ottenere.

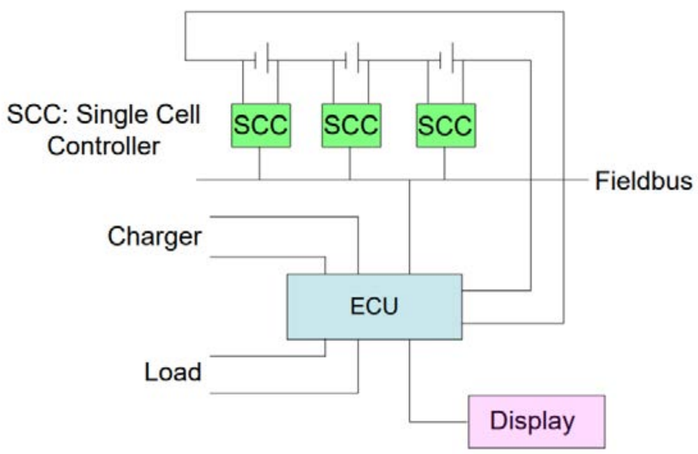


Figura 2.1.1: schema di base di un BMS

Per poter comprendere meglio il funzionamento di un sistema di gestione dell'accumulatore generico si può prendere come riferimento la figura 2.1.1: ogni singola cella, facente parte dell'array che costituisce il complessivo pacco batteria, viene monitorata da un microprocessore (cuore del Single-Cell Controller, SCC) che ne acquisisce la tensione e la protegge sia da una carica eccessiva che da una scarica troppo profonda. Inoltre, come già detto in precedenza, ne viene misurata anche la corrente e talvolta la temperatura. Da tenere in considerazione che quando si parla di SCC non ci si riferisce ad un mero microprocessore ma ad un sistema più ampio e complesso, che comprende anche elementi elettronici quali resistori, condensatori e convertitori ADC (analogico-digitali), oltre a sonde di corrente e amplificatori operazionali, affinché i segnali analogici di tensione e corrente acquisiti dalle singole celle possano essere tradotti ed elaborati in logica digitale dai microprocessori stessi. I vari controllori di cella devono essere in grado di comunicare le informazioni ottenute scambiandole tra loro e poi con una ECU (Electronic Control Unit) che si occupa di raggruppare i dati raccolti e comandare di conseguenza i vari sistemi elettrici ed elettronici alimentati dal pacco batteria monitorato; ecco allora che si rende necessaria l'integrazione di un fieldbus (bus di campo), tipicamente un CAN bus (Controller Area Network bus) nel settore automotive, che funge da mezzo di comunicazione. Più nello specifico il sistema di gestione della batteria, o quantomeno il microprocessore destinato alla comunicazione "verso l'esterno" (trasmette i dati raccolti dal bus in direzione opposta rispetto alla batteria), dovrà trasmettere e comunicare non solo con un'interfaccia verso l'utente per mettere a disposizione lo stato di funzionamento della batteria a livello grafico ma anche con il caricabatteria, fornendo in real time eventuali indicazioni per correggere la fase di carica, e con il carico alimentato durante una qualsiasi fase di scarica. In questo modo è possibile evitare sbilanciamenti tra il livello di tensione delle celle ma anche condizioni operative al di fuori dei limiti previsti per questa tipologia di accumulatori.

In quest'ottica generale si può ben capire come le soluzioni di battery management system possibili siano molto numerose, sia per quanto concerne la scelta dei vari componenti elettronici da impiegare nella parte hardware del sistema, che nella definizione della struttura e delle funzionalità da implementare poi nella parte software; la scelta tra le diverse opzioni deve essere dunque presa in base al contesto e settore di funzionamento in cui il BMS verrà collocato. Nelle prossime pagine si andrà ad affrontare dapprima una trattazione teorica più approfondita sui componenti principali utilizzati, necessaria per acquisire conoscenza e consapevolezza riguardo la seconda parte della tesi, nella quale si fornirà una spiegazione dettagliata delle scelte intraprese durante la definizione e lo sviluppo sia della parte software che hardware dei due sistemi di gestione della batteria realizzati basati su reti di comunicazione diverse.

2.2.Antefatto

Per procedere con la trattazione è fondamentale quantomeno fornire un breve riassunto dei concetti principali presenti in questa tesi; in particolare assume particolare rilevanza il funzionamento del circuito di acquisizione di tensione e corrente di ogni pacco batteria. Se ne deve cogliere e comprendere appieno la struttura e il principio operativo se si vuole poi capire appieno le scelte e considerazioni adottate nello sviluppo della tesi in esame. Si faccia riferimento allora allo schema circuitale in figura 2.2.1.

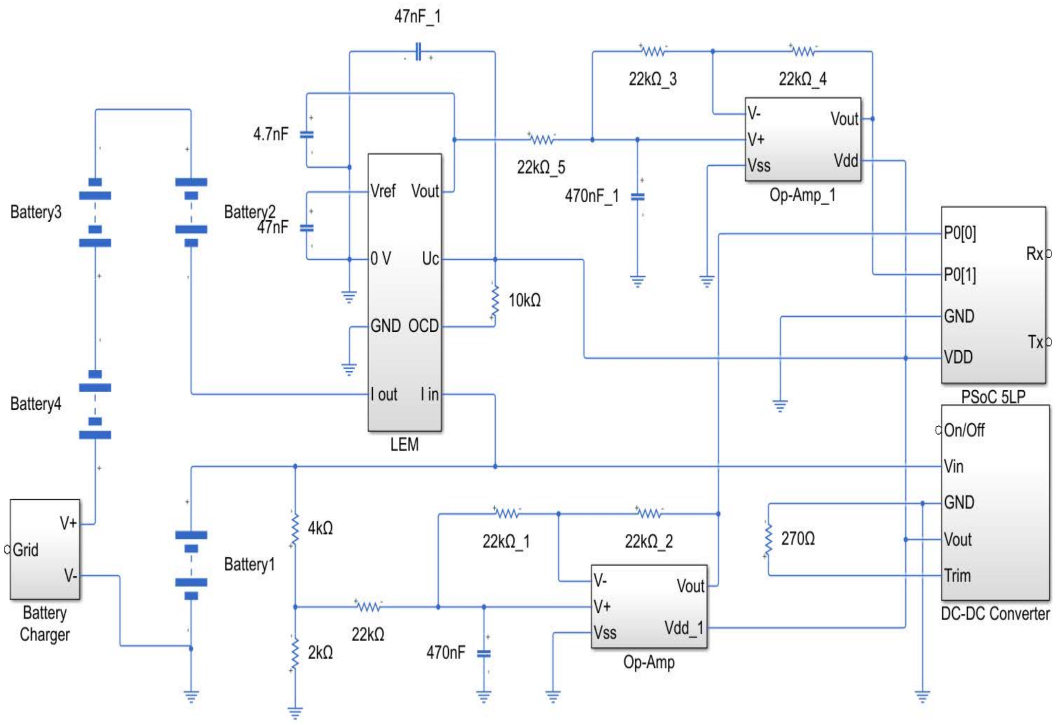


Figura 2.2.1: schema completo del circuito progettato da Samuel Matrella nella sua tesi magistrale

Il circuito è formato sulla sinistra da un caricabatterie collegato alla serie di quattro batterie al piombo acido; i componenti principali, oltre ai resistori e capacitori necessari per un corretto funzionamento, sono in prima battuta un amplificatore operazionale (funzionante da filtro passa basso) per l'acquisizione della tensione e una sonda LEM per la misura della corrente. Vi è poi un convertitore statico DC-DC (necessario per alimentare con il giusto voltaggio i vari componenti elettronici mediante il pacco batteria stesso) ed infine il componente chiave dell'intero sistema, che rivestirà un ruolo di fondamentale importanza nella presente tesi e che sarà approfondito maggiormente nelle prossime pagine: la scheda a microprocessore CY8CKIT-059 della Cypress Semiconductor Corporation. I due segnali analogici ottenuti quindi dai due circuiti di misura di tensione e corrente costituiscono altrettanti input analogici per il kit, che li convertirà in valori digitali e li elaborerà secondo gli scopi per cui è programmato.

La scelta dei componenti elettronici ed elettrici per realizzare il circuito appena illustrato viene data per acquisita e non se ne dibatterà in questa sede; non è d'interesse per questa trattazione approfondire ulteriormente il progetto sviluppato in precedenza dal collega Samuel Matrella, che si può liberamente andare a consultare per ulteriori spiegazioni e chiarimenti. È necessario invece capire la rilevanza che tale progetto assume nella tesi in esame: il circuito appena analizzato può essere visto, assieme al pacco batteria cui viene collegato, come una struttura modulare. Nel complesso ad ogni accumulatore (pacco batteria) dovrà fare riferimento un sistema di questo tipo per poterne acquisire i valori di tensione e corrente in real time e concettualmente l'insieme dei due costituirà un nodo della rete che questo progetto è andato a sviluppare. Per ricostruire infatti lo stato di funzionamento dell'intero sistema di accumulo, come detto in precedenza formato da quattro batterie al piombo acido, è necessario far sì che un microprocessore CY8CKIT-059 (scelto per mantenere una continuità con la tesi del collega, dato che era un componente utilizzato in essa) riceva mediante un opportuno sistema di comunicazione i valori di tensione e corrente acquisiti da ognuno dei circuiti modulari appena menzionati; sarà poi tale schedina, operante in qualche modo da centro di comando dell'intero sistema, ad interfacciarsi con il caricabatterie e/o l'utente mediante PC a seconda degli scopi progettuali. Ecco allora che già si prefigura, parlando di trasmissione di dati tra microprocessori e con dispositivi esterni, la necessità di utilizzare componenti appositi, che verranno approfonditi nel seguito: in primo luogo un mezzo di comunicazione affidabile tra i vari kit CY8CKIT-059 costituenti i nodi della rete, cui farà capo uno in particolare per raccogliere l'insieme di dati, e in secondo luogo uno strumento di comunicazione tra quest'ultimo nodo cosiddetto Master e il PC, per ottenere un'interfaccia verso l'utente che consenta di visualizzare graficamente le informazioni acquisite ed elaborate. Rispettivamente la scelta è ricaduta sul componente CAN (realizzazione di un fieldbus particolare chiamato CAN bus per permettere lo scambio di dati tra diversi microprocessori) e su quello UART (protocollo di comunicazione asincrona conosciuto come RS232 o RS485 per la comunicazione tra CY8CKIYT-059 e PC). Prima di procedere oltre è necessario approfondire innanzitutto il kit CY8CKIT-059 di Cypress Semiconductor utilizzato, dato che rappresenta di fatto il cuore e cervello dell'intero battery management system realizzato.

3. CY8CKIT-059 PSoC 5LP prototyping kit

Il kit di prototipazione CY8CKIT-059 PSoC 5LP sviluppato dalla Cypress Semiconductor Corporate è progettato per essere una piattaforma di prototipazione semplice da usare e fornisce a basso costo una soluzione completa di sistema per un ampio range di applicazioni integrate. Appartenendo alla famiglia di dispositivi PSoC 5LP, ha integrato in sé la CPU ARM® Cortex™-M3 ma, essendo anche un System-on-Chip (SOC), mette a disposizione numerosi blocchi analogici e digitali ad alta precisione, programmabili e riconfigurabili [4]. La spiccata flessibilità della sua architettura ne garantisce un semplice, immediato e comprensivo utilizzo anche da parte di utenti con poca o nessuna esperienza nel campo dei kit di prototipazione. Il kit inoltre include altre caratteristiche vantaggiose ed utili, tra cui:

- connettore micro-USB per consentire lo sviluppo di un'applicazione USB
- capacitori di bypass per assicurare un'elevata qualità di conversione dei convertitori ADC (convertitori analogico-digitali)
- un LED per fornire un feedback visivo dei processi in atto
- un pulsante operante come un semplice input da parte dell'utente
- funzionamento mediante alimentazione nel range 3.3V-5.5V

Il kit inoltre integra in sé anche il KitProg di Cypress che consente funzioni di programmazione, debugging e collegamento, tra cui il componente USB-UART che verrà trattato in seguito [5]. Il kit contiene il board in figura 3.1.

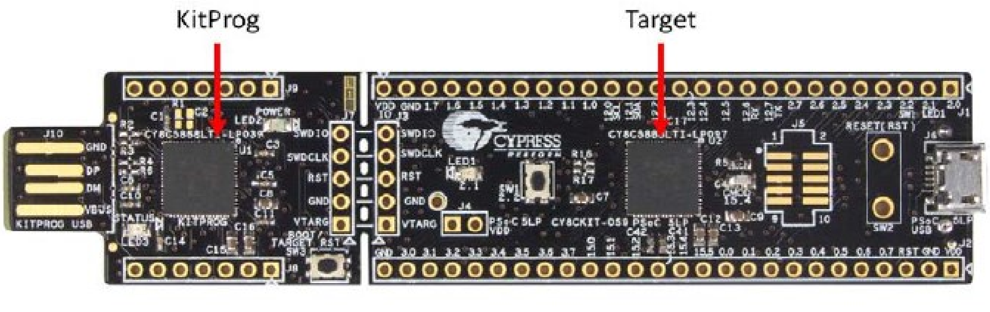


Figura 3.1: CY8CKIT-059 PSoC 5LP Prototyping Kit

Si nota come sia suddiviso in due parti: il KitProg sulla sinistra che permette la programmazione on-board e il debugging del target ad esso collegato, fornendo anche la possibilità di alimentazione della schedina a 5V mediante la connessione USB; la seconda parte è invece visibile sulla destra, dove è presente sia il microcontrollore Target formato dal microprocessore ARM® Cortex™-M3 e dalle periferiche sopra citate, sia due expansion headers che mettono a disposizione ventisei pin input/output l'uno. La caratteristica più vantaggiosa del kit deriva dall'essere un SoC (System on Chip): ogni qualvolta si voglia eseguire una modifica delle varie configurazioni analogiche possibili del board non è necessario cambiare schedina o ridisegnarla usando altri componenti, ma basta invece semplicemente implementare la struttura desiderata sul chip mediante l'ambiente di sviluppo chiamato PSoC Creator [6]. Questo permette un cambiamento della parte analogica diretto, semplice e sicuro, non richiedendo competenze tecniche e specifiche all'utente. Affianco al blocco analogico configurabile, se ne trova anche uno digitale programmabile mediante linguaggio Verilog (linguaggio di programmazione orientato alla descrizione di strutture hardware digitali, caratterizzato da una sintassi molto simile a quella di C e C++), ed entrambi rispondono poi al blocco centrale del microprocessore ARM® Cortex™-M3. Sfruttando le funzioni API (Application Program Interface) scritte in C e generate automaticamente da PSoC Creator ogni volta che si aggiunge un nuovo componente nello ambiente di lavoro, si possono creare collegamenti tra le varie parti digitali che il board mette a disposizione, oltre che inizializzarle e comandarle a seconda degli scopi progettuali che si stanno perseguendo. In figura 3.2 si può notare la struttura user-friendly di PSoC Creator, pensata e realizzata per essere semplice interagirci e di facile comprensione.

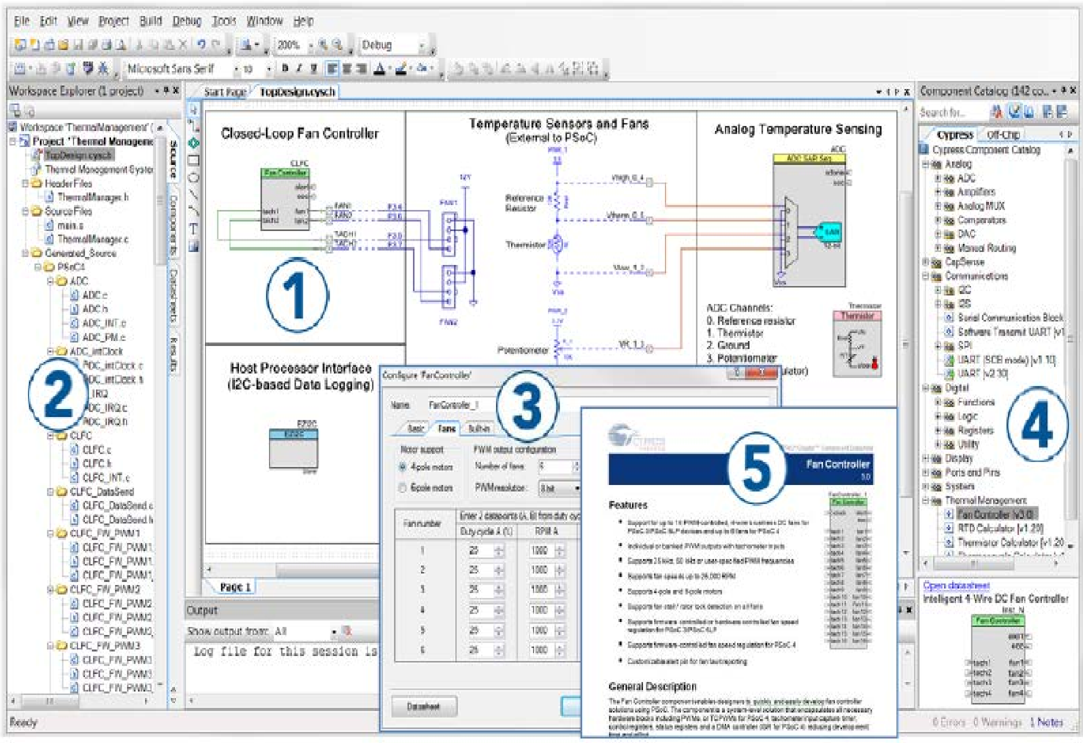


Figura 3.2: interfaccia grafica dell'ambiente di lavoro PSoC Creator

Nel complesso dunque le funzioni logiche implementabili dall'utente coprono un range molto ampio di complessità, dalle semplici operazioni booleane ai UDB (Universal Digital Block ovvero dei blocchi digitali programmabili costituiti per realizzare macchine di stato). Ancora una volta si deve sottolineare l'attenzione da parte di Cypress nel fornire un ambiente di sviluppo più semplice ed accessibile possibile anche per programmatori senza particolare esperienza: i componenti che la scheda mette a disposizione sono infatti rappresentati nel Creator come dei blocchi grafici facilmente gestibili e modificabili e si possono trovare in un'apposita libreria. Ogni modifica dei parametri di un componente viene automaticamente implementata a livello software mediante delle funzioni predefinite e questo evita all'utente il dover realizzare personalmente i driver in C necessari per lavorare con tali blocchi digitali. La maggior parte del codice che definisce le operazioni del Kit viene quindi scritto di default semplicemente interagendo con i componenti mediante l'interfaccia grafica del Creator.

Un'altra differenza sostanziale che distingue un microcontrollore ordinario dal CY8CKIT-059 sta nella diversa struttura dei due: mentre il primo è comandato dalla CPU e il suo intero funzionamento dipende da essa (si interfaccia tramite registri all'intero set di periferiche, senza central processing unit non potrebbe funzionare), il kit della Cypress è caratterizzato da una ben differente struttura gerarchica. Per quest'ultimo infatti CPU, pin di input/output, componenti digitali e analogici assumono una stessa importanza, come si evince dalla figura 3.3.

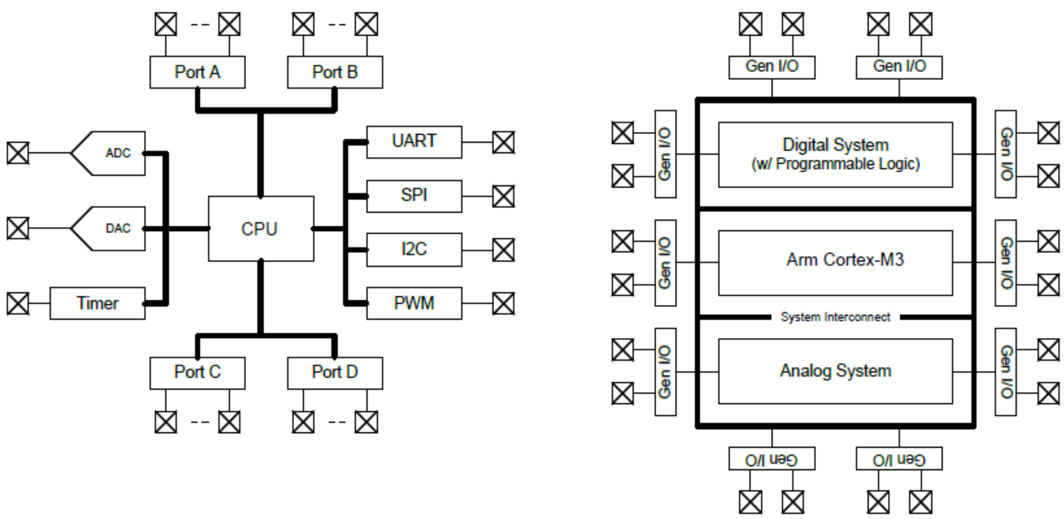


Figura 3.3: a sinistra la struttura di un microprocessore ordinario, a destra di un PSoC

Se nel microcontrollore tradizionale qualsiasi compito da assolvere interessa inevitabilmente l'intervento della CPU e l'esecuzione del codice del firmware, indipendentemente da quale sia lo specifico task, nel prototyping kit in esame è possibile svolgere determinati operazioni o routine senza dover mettere in gioco la central processing unit; si dice in quest'ultimo caso che è consentita la Asyncronous Parallel Processing. Ecco allora che per un dispositivo PSoC il cuore e il cervello del sistema non coincidono con la CPU ma piuttosto con il poter programmare e indirizzare le periferiche, tant'è che sono proprio queste le funzionalità che vanno a definire le caratteristiche del sistema finale; in questo modo la central processing unit stessa è meno coinvolta nel dover assolvere tutti i compiti richiesti al kit e può invece focalizzarsi sugli oneri computazionali.

In figura 3.2 era già stata messa in evidenza l'interfaccia grafica di PSoC Creator, che è definito da Cypress come un IDE (Integrated Development Environment) ovvero un ambiente di progettazione integrata: permette infatti lo sviluppo sia della parte hardware che software di tutti i sistemi basati sulle famiglie di componenti PsoC 3, PSoC 4 e PSoC 5LP. Se per il software si deve ricorrere al linguaggio C e alle API già introdotte in precedenza, è per l'hardware che si rende ancora più evidente la semplicità di utilizzo di questo ambiente di lavoro; per costruirne il sistema infatti l'utente deve solo trascinare nello schema del progetto i diversi componenti necessari, ricercabili all'interno della libreria standard Cypress presente nell'interfaccia grafica, e ognuno descritto da un proprio datasheet, consultabile sia per quanto concerne i principi di funzionamento che per le Application Program Interface richiamabili. Ogni componente si presenta come un blocco grafico che fornisce all'utente la possibilità di variarne facilmente parametri e caratteristiche ma anche creare collegamenti tra l'uno e l'altro; l'ultimo passo necessario per la loro semplice e veloce inizializzazione deve essere compiuto via software, tramite apposite API consultabili nel datasheet del componente stesso. Esiste un'altra libreria disponibile su PSoC Creator, composta da componenti esterni al kit ma sfruttabile per creare un'immagine grafica dello schema completo raffigurante l'intero circuito che si vuole realizzare.

Conclusa ora la panoramica sul kit di prototipazione CY8CKIT-059 usato nel battery management system implementato, si proseguirà ora nella trattazione soffermandosi più in dettaglio sui due componenti principali del circuito d'esame: il componente CAN, assieme a delle spiegazioni più dettagliate sul CAN bus, e il componente UART. Sostanzialmente rappresentano entrambi due metodi di comunicazione tramite cui il microprocessore può scambiare dati con l'ambiente esterno, inteso come dispositivi che possono essere un PC o altri microprocessori; tuttavia sono profondamente diversi, sia nel principio di funzionamento che poi nelle rispettive applicazioni pratiche, come si evincerà nel seguito.

4. CAN bus

Il CAN bus, acronimo di Controller Area Network bus, è nato nel 1989 dalla collaborazione tra Bosh ed Intel; inizialmente concepito e realizzato come bus di campo per auto, al giorno d'oggi è largamente utilizzato nell'ambito dell'automazione industriale per permettere la comunicazione fra dispositivi elettronici intelligenti. È un protocollo di comunicazione seriale che supporta efficientemente controlli distribuiti real time con un elevato livello di sicurezza [7]. In figura 4.1 è rappresentato un generico esempio di tre nodi connessi ad una rete CAN.

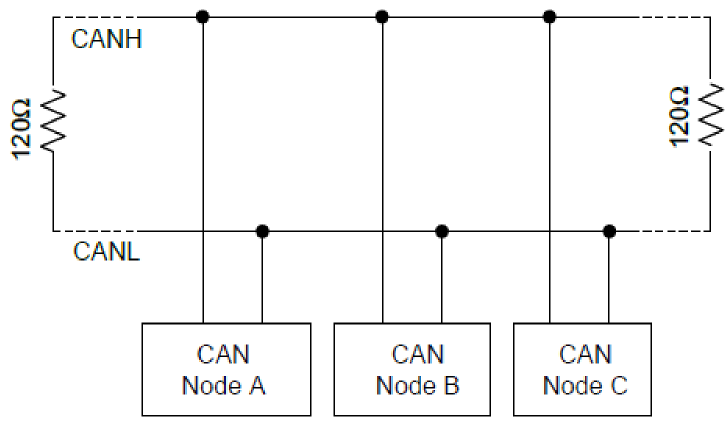


Figura 4.1: nodi connessi ad una rete CAN

Il fieldbus così definito viene impiegato quando si rendono necessarie delle stringenti esigenze di comunicazione, tra cui:

- bassi tempi di latenza
- alta immunità ai disturbi elettromagnetici
- determinismo
- elevato throughput
- flessibilità di configurazione e rilevazione degli errori di trasmissione

Inoltre, rispetto alle comunicazioni analogiche o punto-punto, essendo una tipologia di comunicazione digitale è più robusta per natura rispetto a quella analogica e assicura un risparmio su costi d'installazione e cavi, oltre che sulla manutenzione e sull'evoluzione del sistema. Il CAN bus in particolare implementa una trasmissione seriale asincrona, ovvero ogni frame viene trasmesso ad intervalli temporali casuali. I clock interni del ricevitore e trasmettitore sono indipendenti tra loro e dunque il nodo che riceve deve sincronizzarsi sul bit in arrivo: il segnale in ingresso viene campionato ad una frequenza multipla di quella di trasmissione (è fondamentale allora che il ricevitore sia a conoscenza di quest'ultima per poter applicare una corretta frequenza di campionamento), considerando i fronti del segnale come inizio di un bit. La decisione sul valore logico del bit ricevuto viene assunta considerando i campioni prelevati intorno alla metà di ogni bit, come si evince dalla figura 4.2.

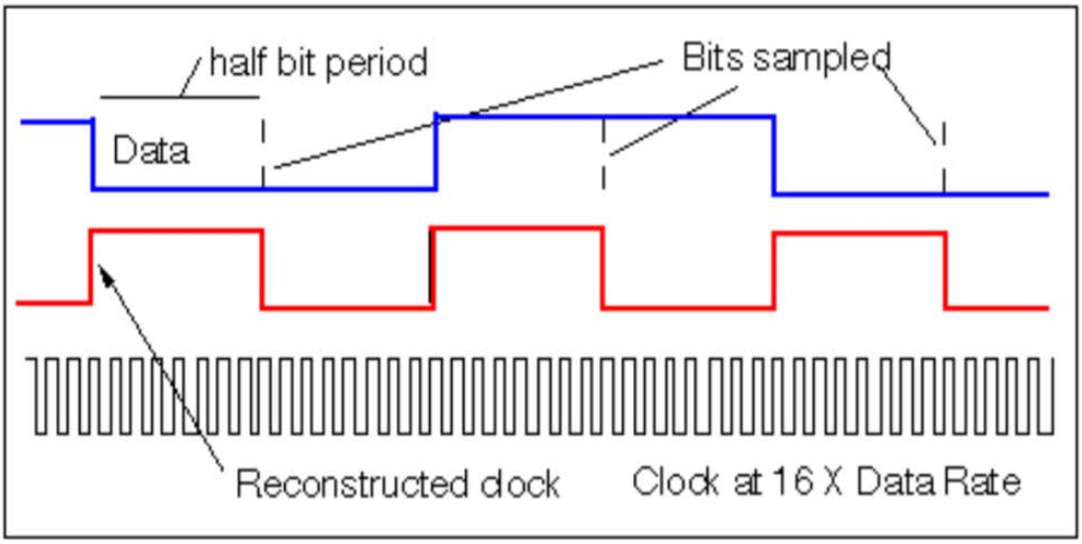


Figura 4.2: esempio di sincronizzazione di bit nel caso di frequenza di campionamento pari a sedici volte la frequenza di trasmissione

Il protocollo CAN è inoltre caratterizzato da un preciso bit timing: ogni bit è infatti formato da quattro segmenti temporali distinti, ognuno dei quali ha una durata multipla di un time quantum (tq, unità ti tempo ottenuta per divisione dell'oscillatore interno di ogni nodo). Di conseguenza un bit può avere una durata dagli otto ai venticinque tq, a seconda dei settaggi impostati dall'utente in fase di progettazione e caratterizzazione del CAN bus da realizzare.

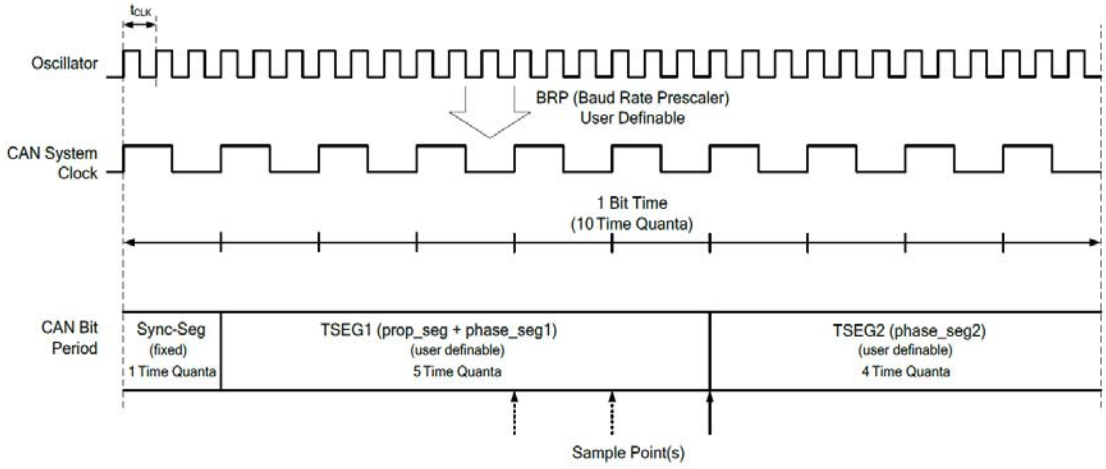


Figura 4.3: bit timing nel protocollo CAN

Come messo in evidenza in figura 4.3, questi quattro periodi temporali sono chiamati:

1. SYNC SEG (segmento di sincronizzazione, 1 tq), necessario per sincronizzare i vari nodi sul bus; in questo intervallo ci si aspetta di trovare almeno un fronte del segnale
2. PROP SEG (segmento di tempo di propagazione, da 1 ad 8 tq), serve per compensare i ritardi temporali fisici della rete; è due volte la somma del tempo di propagazione del segnale sulla linea del bus, il ritardo del comparatore in ingresso e il ritardo del driver di uscita
3. PHASE SEG1 e PHASE SEG2 (segmenti di fase di buffer, da 1 ad 8 tq l'uno a seconda della lunghezza massima del bit in termini di time quantum), sono utilizzati per compensare gli errori di fase dei fronti; possono essere allungati o accorciati mediante risincronizzazione

Il CAN bus è costituito da due linee fisiche (cavi), chiamate CAN_H e CAN_L, ed è terminato alle estremità da due resistenze da 120 ohm l'una; per trasmettere un bit 0 si deve impostare nel bus secondo le specifiche del protocollo CAN una tensione differenziale pari a +2V (nella realtà è sufficiente una tensione compresa tra +1.5V e +3V) mentre per un bit 1 non deve esserci differenza di potenziale tra le due linee. Il bit 0 è dominante mentre il bit 1 è recessivo (coincide con un bus allo stato di riposo): ciò significa che se si vuole ottenere un valore logico 0 sul bus è sufficiente che almeno un nodo che trasmetta 0, dato che tale valore logico domina sull'1 recessivo. Di conseguenza se al contrario si vuole ottenere un valore logico 1 sul bus, tutti i nodi devono trasmettere 1 ovvero un bit recessivo.

Altro aspetto fondamentale da tenere in considerazione è la modalità con cui i vari nodi della rete possono accedere al CAN bus, che di fatto è Multi Master: il protocollo CAN implementa infatti un CSMA/CA, ovvero Carrier Sense Multiple Access/Collision Avoidance. Ciò significa che ogni singolo nodo è permanentemente in ascolto sul bus (Carrier Sense) e quando viene rilevato libero allora tutti i nodi che hanno un messaggio da inviare tentano di trasmettere (Multiple Access); ai messaggi viene assegnata, nel momento in cui si configura il bus, una priorità, cosicché quando si verifica una collisione (più nodi tentano di trasmettere simultaneamente una volta rilevato il bus libero) solo il nodo che genera il frame con priorità maggiore continua a trasmettere (Collision Avoidance). Di fatto quindi non si verifica mai una collisione nella trasmissione di messaggi sul bus, nonostante l'accesso ad esso rimanga aperto a tutti i nodi collegati. Per comprendere come i nodi del bus riconoscano quale fra loro abbia il messaggio a priorità più alta, e quindi quale fra essi sia autorizzato a trasmettere, si deve riprendere il concetto di valore logico 0 dominante e 1 recessivo: la collisione di messaggi viene riconosciuta da un nodo quando esso tenta la trasmissione di un bit 1 ma sul bus rileva un bit 0 (dominante sull'1). Questo nodo dovrà allora terminare la propria trasmissione e potrà liberamente riprovare la comunicazione quando avrà rilevato nuovamente il bus libero, ovvero avrà letto su di esso sette bit recessivi consecutivi. Per maggiore chiarezza e comprensione di questo concetto chiamato "arbitraggio del bus" ed appena introdotto, si può fare riferimento alla figura 4.4.

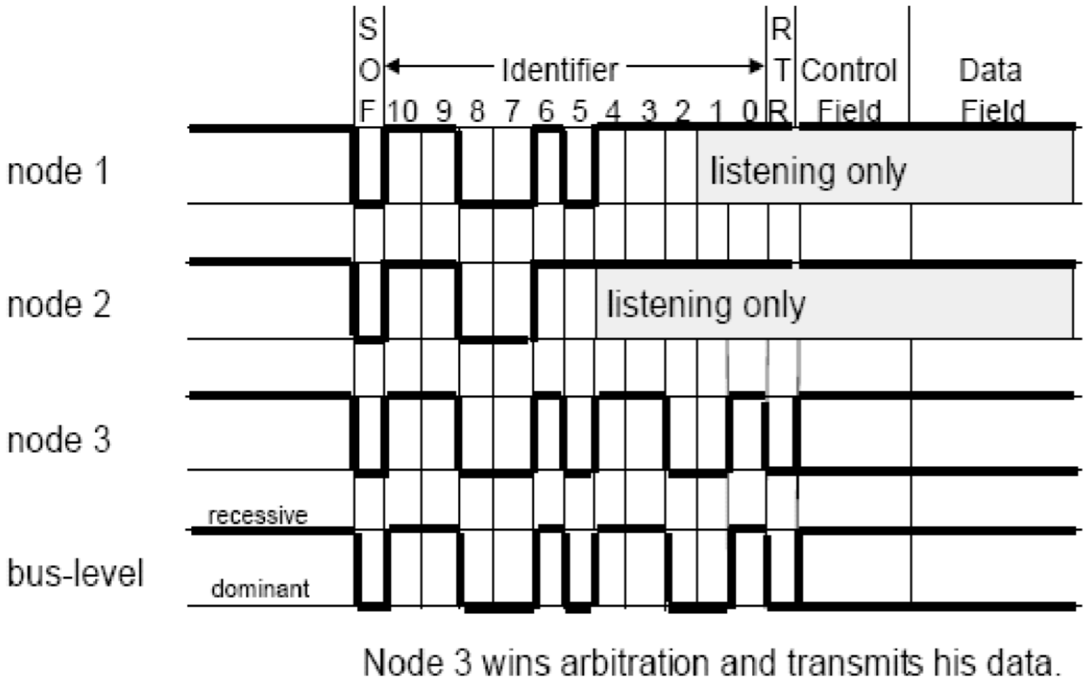


Figura 4.4: arbitraggio del CAN bus, processo di selezione del nodo che la priorità di trasmissione

È importante capire come è impostato il transfer layer (livello trasmissione) nelle specifiche CAN, ovvero la struttura del frame e la codifica delle informazioni, e a tal proposito è funzionale e significativa la figura 4.5.

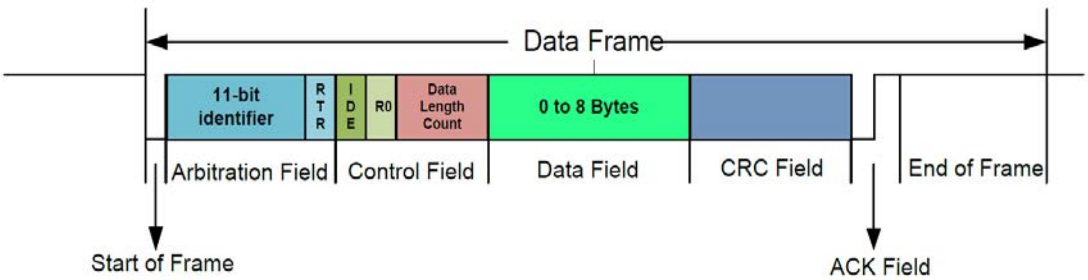


Figura 4.5: struttura del frame nel protocollo CAN

Il primo bit coincide con il SOF (Start Of Frame), ovvero l'inizio del messaggio, che deve essere secondo specifiche un bit 0 dominante; segue poi l'Arbitration Field (campo di arbitraggio, da 12 a 32 bit) dove ha luogo l'arbitraggio del bus spiegato in precedenza. Questo campo è diviso in due parti: la prima coincide con l'identifier (identificatore), usato per descrivere il significato dei dati inclusi nel data frame, mentre la seconda rappresenta il bit RTR (Remote Transmission Request) necessario per specificare se il messaggio sia un data frame o un remote frame. Viene poi il Control Field (campo di controllo, 6 bit) dove viene indicato il numero di byte contenuti nel Data field (di lunghezza compresa tra 0 a 8 bytes), che costituisce il messaggio vero e proprio che si vuole inviare. Esiste poi il campo CRC (Cyclic Redundancy Check), che implementa un metodo di controllo e rilevazione dell'errore, ovvero permette di verificare che il messaggio inviato sul bus coincida con quello effettivamente letto dal nodo ricevente. Un ulteriore step per constatare che almeno un nodo abbia ricevuto il messaggio inviato nel bus viene implementato nell' ACK field (Acknowledge Field, campo di riconoscimento) di lunghezza pari a 2 bit: il primo bit viene trasmesso recessivo 1 ed ogni nodo che riceve il frame correttamente lo riscrive dominante 0. Viene segnalata infine la terminazione del frame mediante l'EOF (End Of Frame) ovvero 7 bit recessivi 1.

Nonostante la struttura appositamente studiata del protocollo CAN, tale metodo di comunicazione è comunque passibile di errori di vario tipo, a partire da errori di bit stuffing (si ricevono più di 5 bit uguali consecutivi in un frame), differenza di Cyclic Redundancy Check tra ricevitore e trasmittente, ACK field non dominante (nessun nodo ha ricevuto il frame inviato nel bus), errore di bit (il trasmettitore vede un bit sul bus diverso da quello che ha inviato) e l'errore di forma (divergenza nel formato fissato del messaggio, e.g. un Data Lenght Code maggiore di 8 bytes). Ad ogni nodo sono dunque associati due contatori diversi: un REC (Receive Error Counter) che viene incrementato di 8 ogni volta che il nodo stesso riceve un frame errato e diminuito di 1 in caso contrario, ed un TEC (Transmit Error Counter) che opera allo stesso modo ma in relazione ai frame inviati dal nodo in questione. Finché entrambi i due contatori sono minori di 127 il nodo si dice in stato Error Active, ovvero in caso di errore trasmette nel bus un Active Error Frame (messaggio di soli bit 0 dominanti così da resettare il bus e far ripartire la fase di arbitraggio per la trasmissione); quando invece uno dei due registri REC o TEC superano il valore127 il nodo entra nello stato Error Passive, i.e. in caso di errore invia un Passive Error Frame (messaggio di soli bit recessivi 1, ininfluenti per quanto concerne l'arbitraggio) e quindi non influenza più gli altri nodi, essendo di fatto neutro rispetto alla comunicazione del CAN bus stesso.

Esiste però una situazione più estrema che si verifica quando il contatore TEC di un nodo supera il valore 255: in tal caso il nodo in questione viene disconnesso dalla rete e prima di ricollegarsi al bus dovrà aspettare un certo periodo di tempo, così da potersi resettare e riconfigurare, evitando che costituisca un intralcio alla comunicazione [8]. L'intero processo appena descritto è ben rappresentato schematicamente in figura 4.6.

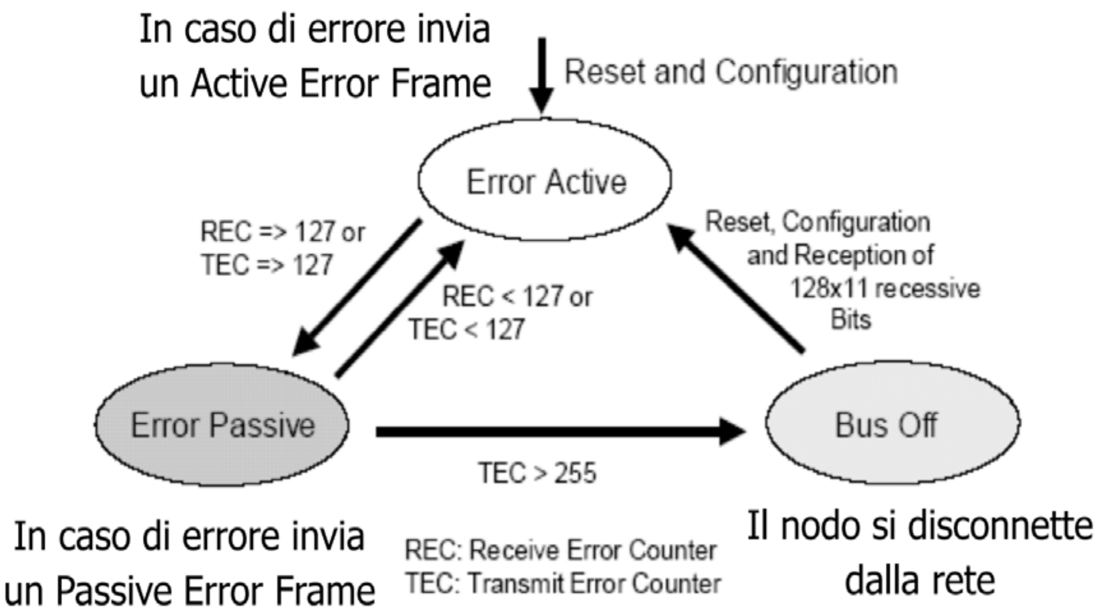


Figura 4.6: stati di errore di un nodo CAN

In definitiva quindi la probabilità che nel bus vi sia un frame corrotto/errato e nessun nodo lo riconosca in quanto tale è infinitesimale, visti i diversi livelli di accertamento della correttezza del messaggio e della capacità di un nodo di auto-monitorarsi ed auto-escludersi in caso di funzionamento errato.

Si è già ampiamente parlato della priorità che caratterizza ogni messaggio scambiato nel CAN bus; si deve precisare però per una maggiore comprensione come essa venga assegnata ad ogni frame che interessa la comunicazione fra nodi. A differenza del modus operandi convenzionale nel settore dei fieldbus, nel protocollo CAN non sono i nodi della rete ad essere caratterizzati ciascuno da un indirizzo specifico bensì lo sono i diversi messaggi che essi possono scambiare. Proprio i frame infatti sono contraddistinti da un ID (Identifier), ovvero un identificativo: questo può essere lungo

11 bits (base frame format) oppure 29 (extended frame format) in base al valore che in fase di parametrizzazione della rete si attribuisce all'IDE, come si vedrà nel corso della trattazione. In figura 4.7 è rappresentata graficamente la differenza di formato tra le due tipologie di formato che differenziano i data frame.

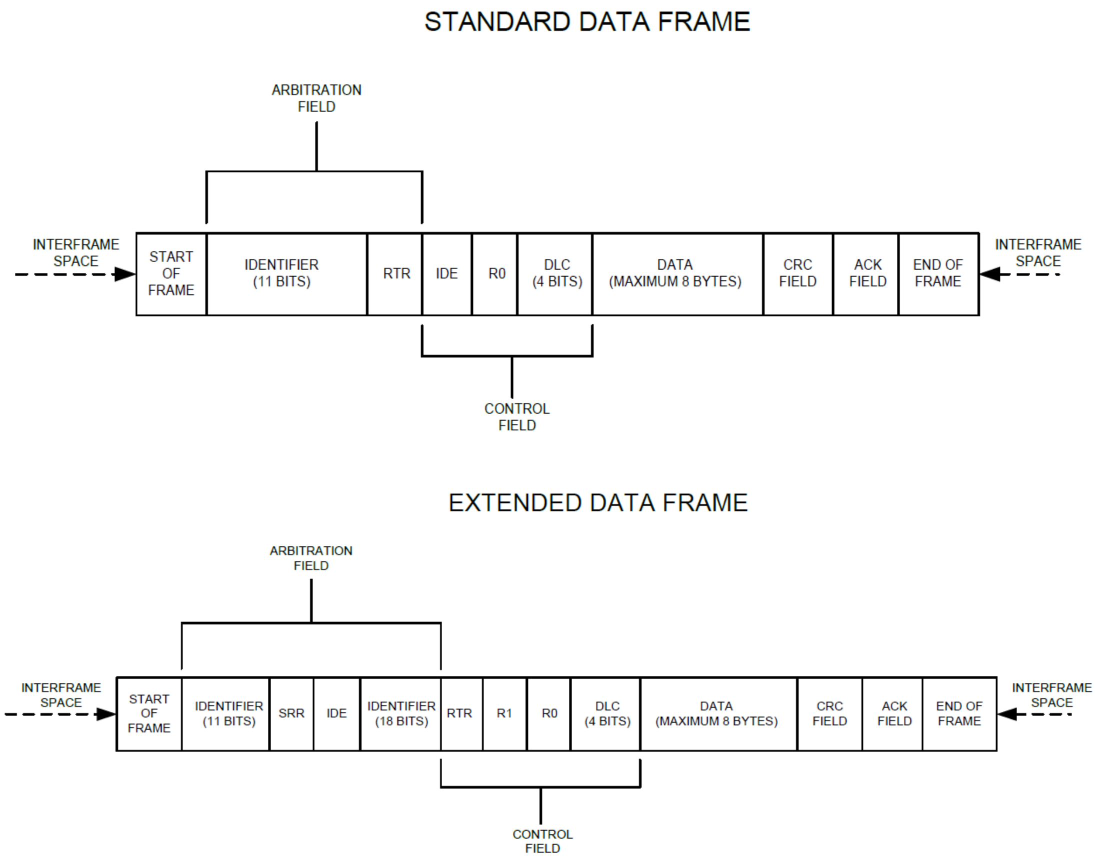


Figura 4.7: formato di uno standard data frame confrontato con quello di un extended data frame

Fino ad ora nella trattazione sono stati fatti coincidere i messaggi trasmessi e ricevuti in un CAN bus con dei data frame; si deve invece tenere in considerazione che possono essere scambiati diverse tipologie di frame, che si distinguono in cinque categorie:

- Data frame, che contiene i dati che un nodo trasmette e che è strutturato come spiegato in precedenza
- Remote frame, associato ad una RTR (Remote Transmission Request) ovvero ad una richiesta di trasmissione di un data frame con un preciso identificatore

- Error frame, trattato in precedenza, trasmesso in caso di rilevazione di errore da parte di un nodo della rete
- Overload frame, costituisce un ritardo introdotto solitamente tra data frame oppure remote frame per indicare che un nodo è già occupato con un pacchetto dati precedentemente ricevuto
- Interframe frame, che funge da separatore tra frame diversi

L'utilizzo di una tipologia di messaggio rispetto ad un'altra dipende ovviamente dallo scopo e dalla struttura che l'utente si è prefissato in fase di progettazione e dalle condizioni operative attuali del bus; inoltre la scelta non è una singola e specifica, intendendo con ciò che per scambiare un determinato messaggio sul bus si può arbitrariamente decidere se inviarlo direttamente all'interno della routine di uno dei nodi come data frame oppure richiederlo mediante remote frame. Questa considerazione è stata specificata per sottolineare ancora una volta la flessibilità del protocollo CAN e della progettazione di un fieldbus che lo implementi: le scelte che si devono intraprendere nel corso dello sviluppo di un progetto di tale tipo, come quello in esame in questa trattazione, dipendono allora non solo dai componenti e strumenti che sono a disposizione, ma anche dall'esperienza e dall'attitudine degli sviluppatori, oltre che alle loro competenze in ambito sia software che hardware.

Conclusa quest'introduzione generale sulle specifiche principali del protocollo CAN, si andrà ora ad esaminare più nel dettaglio il componente CAN che il kit CY8CKIT-059 della Cypress mette a disposizione; in particolare, oltre ad un'analisi prettamente teorica di come esso si interfacci con il microprocessore della schedina, si analizzerà come utilizzarlo e configurarlo sfruttando l'ambiente di sviluppo PSoC Creator.

5. Componente CAN del kit CY8CKIT-059 di Cypress

Il kit CY8CKIT-059 in dotazione permette l’utilizzo del componente CAN che graficamente nel PSoC Creator appare come in figura 5.1.

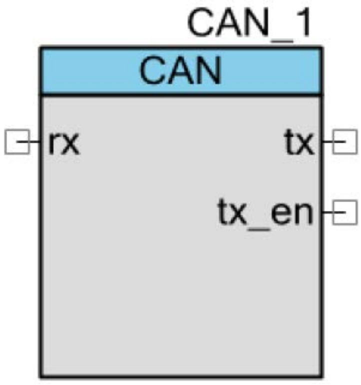


Figura 5.1: interfaccia grafica in PSoC Creator del componente CAN

Il Controller Area Network (CAN) implementa le specifiche CAN2.0A (standard frame) e CAN2.0B (extended frame) come definito nella specifica Bosch ed è conforme alla norma ISO-11898-1 [9]. Il componente può essere utilizzato per impostare le comunicazioni al fine di trasmettere e/o ricevere messaggi sulla rete CAN in cui viene posto ad operare. Come si nota in figura 5.1, visivamente il componente mette a disposizione connessioni di diverso tipo, in particolare:

- Rx, connesso al CAN Rx bus di un transceiver esterno, input corrispondente al segnale ricevuto dal CAN bus
- Tx, connesso al CAN Tx bus di un transceiver esterno, output corrispondente al segnale inviato al CAN bus
- Tx_en (Tx enable), rappresenta un segnale tramite cui il componente CAN va ad abilitare la trasmissione di messaggi da parte del transceiver esterno cui è collegato; si tratta di un segnale di output opzionale e quindi non indispensabile per il corretto funzionamento del componente
- Interrupt, altro segnale di output opzionale che costituisce un vero e proprio evento di interrupt generato dal componente in seguito ad una sua determinata azione (e.g. ricevere o inviare un messaggio)

5.1.Parametri del componente

Una volta trascinato un componente CAN nello spazio di lavoro di PSoC Creator, cliccandovi sopra due volte con il tasto sinistro del mouse si apre la finestra di Configure (configurazione) che si può vedere in figura 5.1.1.

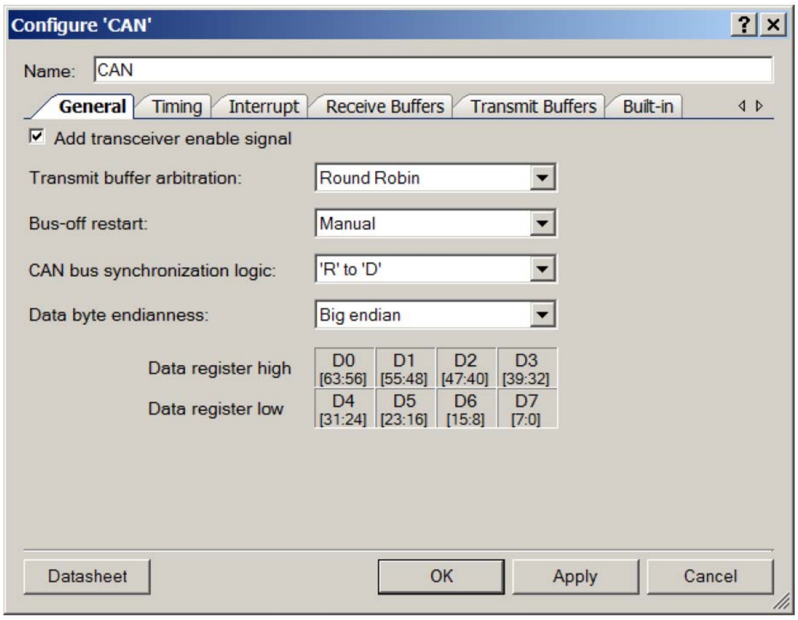


Figura 5.1.1: finestra di dialogo generale per settare il componente CAN

Nella prima finestra di dialogo sono possibili diverse azioni: si può rendere visibile il segnale tx_en citato in precedenza, scegliere la modalità di arbitraggio del buffer di trasmissione (Round Robin di default, le mailboxes vengono servite in un ordine definito: 0-1-2 ... 7-0-1 assicurando ad ognuna la stessa probabilità di essere trasmessa, oppure si può impostare Fixed Priority per stabilire manualmente quale sia quella a priorità di invio più alta), il tipo di reset del bus (manuale di default, oppure automatico ma è consigliata secondo manuale la prima opzione), la logica di sincronizzazione del CAN bus (‘R’ to ‘D’ di default, ovvero in corrispondenza di un fronte da recessivo a dominante del segnale, oppure Both Edges per un fronte di qualsiasi tipo) ed infine l’endianess del byte di dati (posizione del byte nel data field di un frame, Big Endianess di default).

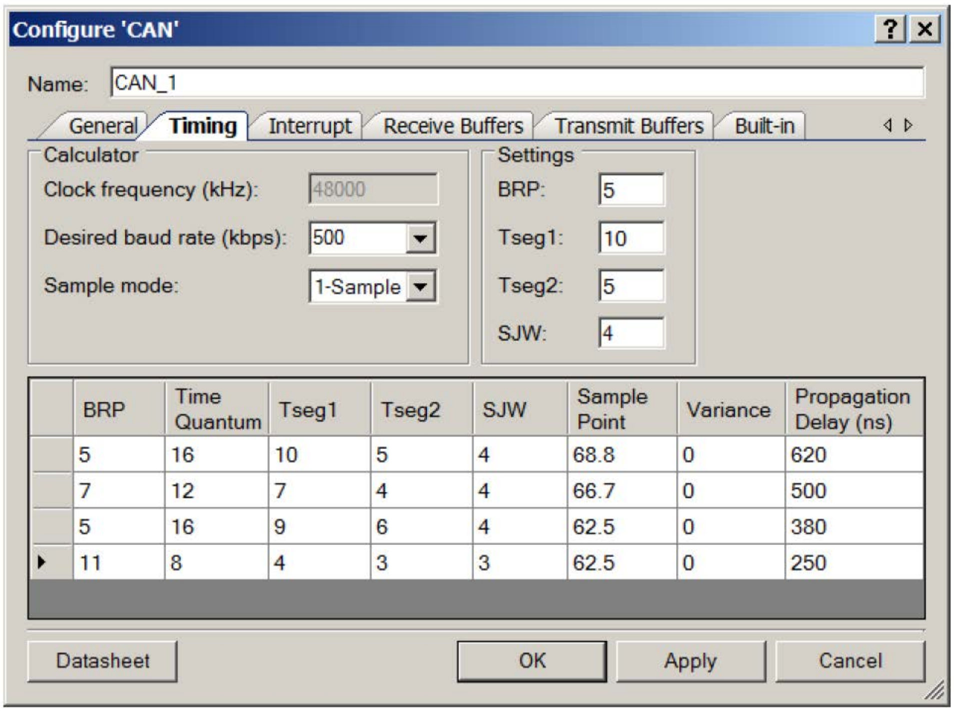


Figura 5.1.2: finestra del timing del componente CAN

In figura 5.1.2 si vede invece la seconda finestra di dialogo, quella di temporizzazione, dove andando a settare la velocità di scambio dati nel CAN bus (baud rate espresso in kbps, i.e. kbit per secondo) e la modalità di campionamento (sample mode) si possono definire le lunghezze dei quattro periodi temporali in cui un frame CAN è suddiviso (già nominati e descritti in precedenza). La temporizzazione dei bit è calcolata secondo le specifiche ISO e le impostazioni di registro proposte per i segmenti di tempo (BRP, Tseg1, Tseg2 e SJW) sono visualizzate nella tabella dei parametri nella parte bassa della finestra: con BRP si intende il valore del Bit Rate Prescaler necessario per calcolare un time quantum (15 bit totali, 0 indica 1 clock, 7FFFh indica 32768 cicli di clock) mentre SJW è l’acronimo di Synchronization Jump Width ovvero ampiezza del segmento di sincronizzazione, che deve assumere un valore minore o uguale sia di Tseg1 che di Tseg2. È possibile selezionare i valori da utilizzare facendo doppio clic sulla riga appropriata; in alternativa si può scegliere di inserire manualmente i valori per Tseg1, Tseg2, SJW e BRP nell'apposito campo valori d'ingresso. L’eventuale settaggio di impostazioni errate di temporizzazione dei bit potrebbero far sì che il controllore CAN rimanga in stato di errore permanente: quindi, se i valori inseriti manualmente non corrispondono a nessuno dei valori indicati nella tabella, il componente mostra sullo schermo il seguente messaggio di avvertimento: "An invalid set of BRP/Tseg1/Tseg2/SJW is entered. Please select a valid set from the table".

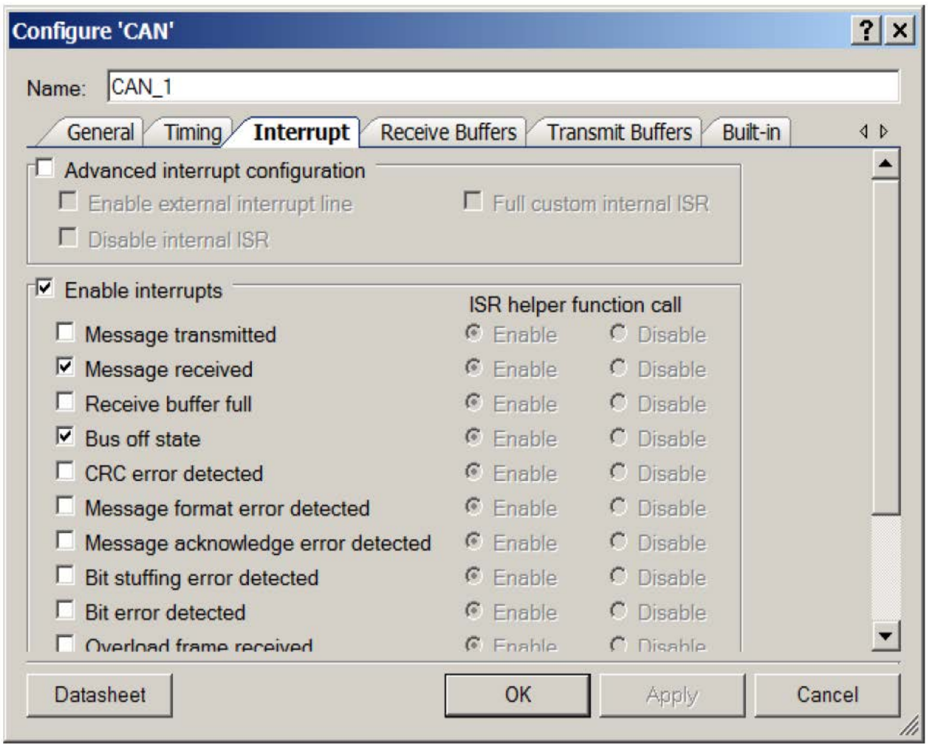


Figura 5.1.3: finestra per la gestione degli interrupt

La terza finestra di dialogo, raffigurata in figura 5.1.3, è quella che permette di abilitare o disabilitare gli interrupts globali del controllore CAN; di default l'opzione "Enable interrupts" è attiva e quindi una volta che nel main.c (codice principale con cui viene programmato da parte dell'utente il kit, deve essere implementato manualmente) presente tra i file del progetto in via di sviluppo nel PSoC Creator si dà il comando CAN_1_Start() gli interrupts vengono attivati. In tal caso spetta all'utente decidere a quale evento associare una ISR (Interrupt Service Routine) tra una lista di opzioni diverse; ciò permette di far coincidere ad un messaggio ricevuto dal CY8CKIT-059, e quindi all'interrupt corrispondente generato (se selezionato nella lista di figura 5.1.3), una determinata risposta (routine eccezionale) del controllore. Si deve ricordare che una qualsiasi ISR è completamente personalizzabile dall'utente, utilizzando i comandi dedicati e implementandoli nel main.c del progetto; questo argomento specifico verrà approfondito nel seguito della trattazione.

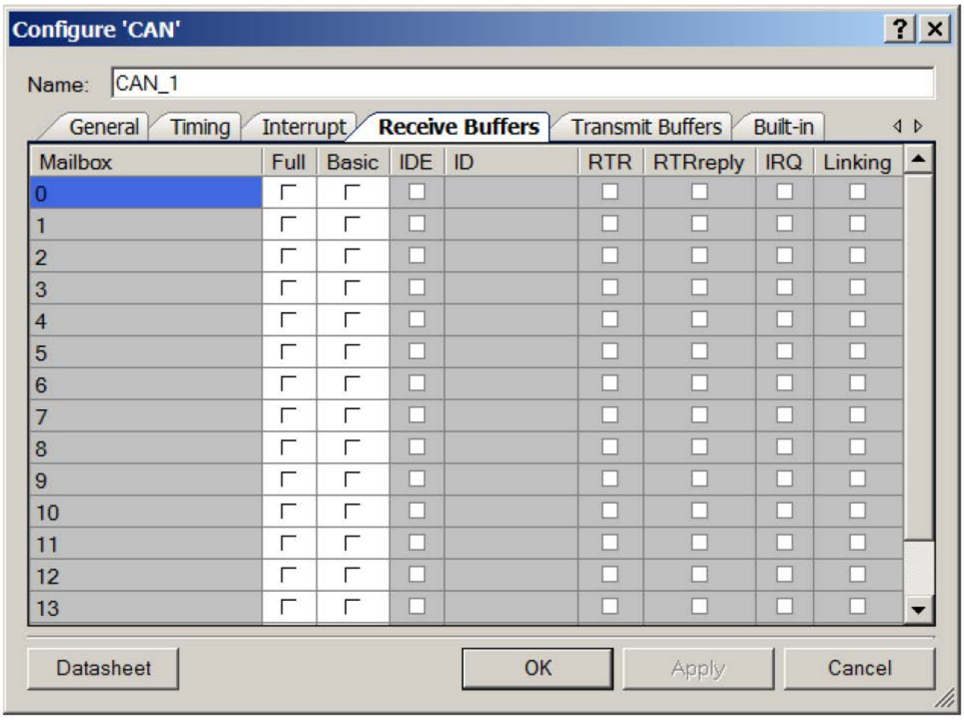


Figura 5.1.4: finestra per la gestione dei buffers di ricezione

La quarta finestra di dialogo permette invece la gestione dei buffers (i.e. memoria tampone, sono delle determinate zone di memoria utilizzate quando il microcontrollore deve lavorare con periferiche a velocità di trasferimento di dati diverse rispetto a quella della CPU stessa) dedicati alla ricezione di dati. Si distinguono infatti in figura 5.1.4 diverse colonne: la prima indica la particolare mailbox (casella di posta, intesa come sorgente di frame) che si sta settando e la seconda/terza colonna invece la tipologia di mailbox (full o basic rispettivamente). A seconda dell'opzione selezionata tra queste ultime due, si potrà (o meno) andare a configurare manualmente gli altri campi, quali IDE, ID, RTR, RTR Reply, IRQ (Interrupt Request) e Linking. Più in particolare, se una determinata mailbox è configurata Basic allora non sono modificabili le opzioni IDE, ID, RTR e RTR Reply; viceversa se configurata come Full è interamente personalizzabile. Per quanto concerne i diversi setting disponibili ad ogni riga della tabella, è fondamentale capirne la definizione e funzione poiché è in suddetta fase che di fatto si va a determinare la tipologia del frame che viene ricevuto. In dettaglio dunque si possono trovare:

- IDE, quando la casella IDE viene cancellata l'identificatore del messaggio è limitato a 11 bit; viceversa quando l'opzione è selezionata l'identificatore è esteso a 29 bit
- ID, l'identificatore del messaggio
- RTR (Remote Transmission Request), quando è selezionata in modalità Full CAN configura le impostazioni del filtro di accettazione per consentire solo la ricezione di messaggi il cui bit RTR sia impostato
- RTR Reply (Remote Transmission Request Auto Reply), disponibile solo per le mailbox impostate per ricevere i messaggi Full CAN, con il bit RTR impostato; se questa opzione è spuntata, il kit risponde automaticamente ad una richiesta RTR con il contenuto del buffer di trasmissione in questione
- IRQ (Interrupt Request), in concomitanza con la relativa opzione nella Interrupt Tab vista in precedenza, associa o meno un evento di interrupt in caso di ricezione di quel preciso messaggio
- Linking, consente di collegare più caselle di ricezione sequenziali per creare un array di caselle Che agisce come un FIFO (First In, First Out) di ricezione

Bisogna inoltre tenere in considerazione che ogni mailbox RX completa ha un'API predefinita. La lista delle funzioni è disponibile nel file di progetto CAN_1_TX_RX_func.c. Queste funzioni sono compilate in modo condizionato a seconda dell'impostazione della mailbox di ricezione da parte dell'utente; solo le mailbox definite come Full hanno le loro rispettive funzioni compilate. L'identificatore macro CAN_1_RXx_FUNC_ENABLE definisce se una funzione viene compilata: le definizioni sono elencate nel file di progetto CAN_1.h disponibili nella finestra di dialogo laterale al workspace di PSoC Creator.

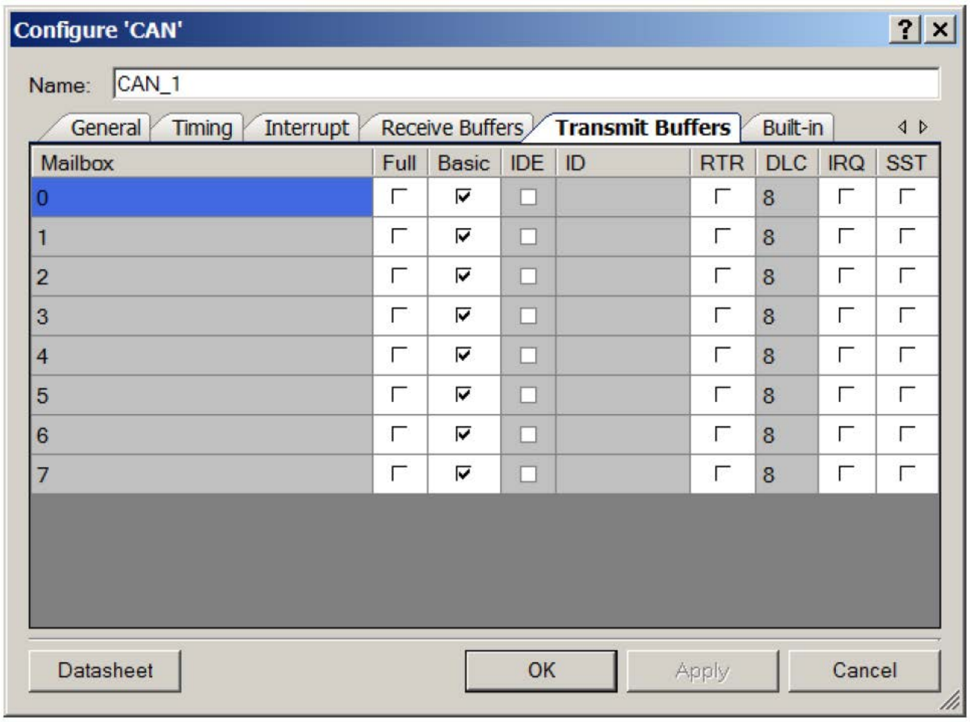


Figura 5.1.5: finestra per la gestione dei buffers di trasmissione

In figura 5.1.5 si può vedere la penultima finestra di dialogo, che permette la configurazione dei buffers di trasmissione. Analogamente alla Receive Buffers Tab descritta in precedenza, si possono definire le diverse mailbox (8 in totale) e settarle come Basic o Full, a seconda degli scopi d'utilizzo, oltre che modificare il campo Mailbox inserendo un nome univoco per ciascuna di esse. Anche la funzione per la gestione di questa mailbox avrà un nome univoco, uguale a quello inserito in tale campo. I caratteri accettati sono: A-Z, a-z, 0-9 e _. Se si inserisce un nome errato, il campo Mailbox ritorna automaticamente al valore predefinito. A differenza invece della precedente finestra di dialogo, dato che ora si sta trattando la trasmissione di dati e non più la ricezione, oltre a IDE, ID, RTR e IRQ, ora si possono definire (se la Mailbox è configurata come Full) le opzioni DLC e SST. Per DLC si intende Data Length Code, ovvero il numero di bytes dati che il messaggio da trasmettere contiene, mentre SST è l'acronimo di Single Shot Transmission: se abilitata viene impedita la ritrasmissione di un messaggio CAN a causa di una perdita di arbitraggio o di un generico errore del bus. Quest'ultima opzione non è tuttavia disponibile per i dispositivi PSoC 5LP, come il kit CY8CKIT-059 in uso in questa trattazione) e quindi non è rilevante per il caso d’interesse trattato nel seguito. Così come le mailbox RX Full, anche le TX Full hanno delle API predefinite associate, la cui lista di funzioni disponibili è presente in CAN_1_TX_RX_func.c.

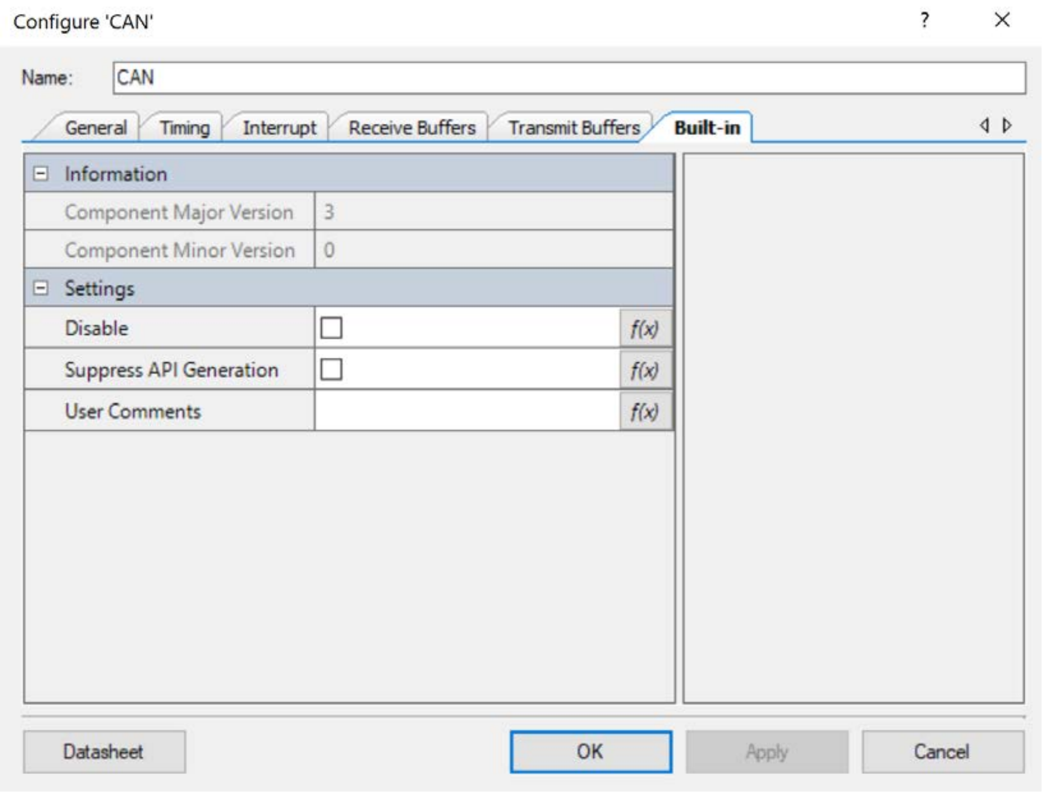


Figura 5.1.6: finestra di dialogo Built-in

L’ultima finestra di dialogo è quella di Built-in, visibile in figura 5.1.6, che permette all’utente di modificare i Settings, in particolare si può scegliere se disabilitare o meno la possibilità stessa di modificare i parametri del componente (come spiegato in precedenza tramite le altre finestre di dialogo già introdotte) ma anche sopprimere la generazione automatica delle API, eseguita di default in background dal PSoC Creator in seguito ad ogni settaggio implementato tramite interfaccia grafica.

Si devono approfondire con particolare attenzione anche le API (Application Programming Interface) implementabili per un componente, consultandone il datasheet che Cypress Semiconductor mette a disposizione; le routine dell’Application Programming Interface consentono di configurare il componente tramite software. La seguente tabella in figura 5.1.7, presa dal datasheet del componente CAN, elenca e descrive l'interfaccia di un numero limitato di funzioni tra tutte quelle disponibili. Per impostazione predefinita, PSoC Creator assegna il nome "CAN_1" alla prima istanza di un componente in un determinato progetto. È possibile rinominare l'istanza a qualsiasi valore univoco che segua le regole sintattiche per gli identificatori. Il nome dell'istanza diventa il prefisso di ogni nome di funzione globale, variabile e simbolo costante. Per la leggibilità, il nome dell'istanza usato nella seguente tabella è "CAN"; affinché una qualsiasi API funzioni correttamente, si deve ricordare di utilizzarne all'interno del listato il nome assegnato al componente da parte dell'utente e non il nome del componente in quanto tale (e.g. si può chiamare il componente CAN "xxxxx", quando si va a richiamarne le API deve essere questo il prefisso da utilizzare e non "CAN" in quanto tale).

Function	Description
CAN_Start()	Sets the initVar variable, calls the CAN_Init() function, and then calls the CAN_Enable() function.
CAN_Stop()	Disables the CAN.
CAN_GlobalIntEnable()	Enables global interrupts from CAN Component.
CAN_GlobalIntDisable()	Disables global interrupts from CAN Component.
CAN_SetPreScaler()	Sets prescaler for generation of the time quanta from the BUS_CLK/SYSCLK.
CAN_SetArbiter()	Sets arbitration type for transmit buffers.
CAN_SetTsegSample()	Configures: Time segment 1, Time segment 2, Synchronization Jump Width, and Sampling Mode.
CAN_SetRestartType()	Sets reset type.
CAN_SetSwapDataEndianness() [2]	Enables or disables the endian swapping of CAN data bytes.
CAN_SetEdgeMode()	Sets Edge mode.
CAN_RXRegisterInit()	Writes only receive CAN registers.
CAN_SetOpMode()	Sets Operation mode.
CAN_SetErrorCaptureRegisterMode() [2]	Sets the error capture register mode to free running or error capture mode.
CAN_ReadErrorCaptureRegister() [2]	This function returns the value of error capture register.
CAN_ArmErrorCaptureRegister() [2]	This function arms the error capture register when the ECR is in error capture mode.
CAN_GetTXErrorFlag()	Returns the flag that indicates if the number of transmit errors equals or exceeds 0x60.

Figura 5.1.7: lista parziale delle funzioni API che sono disponibili in PSoC Creator, operanti correttamente se il componente CAN è chiamato "CAN", visto il prefisso utilizzato in tabella

Oltre all'elenco completo delle funzioni disponibili e la loro descrizione, nel datasheet di ogni generico componente è presente una spiegazione più dettagliata di ognuna di esse, come si può notare in figura 5.1.7; e.g. per le prime due funzioni chiamate CAN_Start() e CAN_Stop() in figura 5.1.8 si può vederne l'approfondimento fornito da Cypress.

uint8 CAN_Start(void)

Description: Sets the initVar variable, calls the CAN_Init() function, and then calls the CAN_Enable() function. This function sets the CAN Component into run mode and starts the counter if polling mailboxes available.

Return Value: uint8: Indication whether register is written and verified

Value	Description
CYRET_SUCCESS	Function passed successfully.
CAN_FAIL	Function failed.

Side Effects: If the initVar variable is already set, this function only calls the CAN_Enable() function.

uint8 CAN_Stop(void)

Description: This function sets the CAN Component into Stop mode and stops the counter if polling mailboxes available.

Return Value: uint8: Indication whether register is written and verified

Value	Description
CYRET_SUCCESS	Function passed successfully.
CAN_FAIL	Function failed.

Side Effects: Pending message in the Tx buffer of PSoC 3/PSoC 5LP will not be aborted on calling the CAN_Stop() API. User has to abort all pending messages before calling the CAN_Stop() function to make sure that the block stops all the message transmission immediately.

Figura 5.1.8: descrizione più dettagliata delle funzioni CAN_Start() e CAN_Stop() disponibili per il componente CAN in PSoC Creator

Si può notare come, oltre ad una descrizione più accurata di come opera la funzione stessa, si spieghi anche l'argomento richiesto, il valore di ritorno che il CY8CKIT-059 fornisce quando viene eseguita ed infine gli effetti collaterali che ne conseguono. Più in dettaglio la funzione CAN_Start() è priva di argomento dato che non deve essere specificato alcunché all'interno delle parentesi, operativamente imposta la variabile initVar (indica se il CAN è stato inizializzato; la variabile viene inizializzata a 0 e impostata a 1 la prima volta che viene chiamato CAN_Start() e questo permette al componente di riavviarsi senza re-inizializzazione dopo la prima chiamata alla routine CAN_Start()), chiama la funzione CAN_Init() e poi chiama la funzione CAN_Enable() (imposta il componente CAN in modalità run e avvia il contatore se sono disponibili mailboxes per il polling) ritornando come valore CYRET_SUCCESS se superata con successo o CAN_FAIL in caso contrario. Tutto ciò comportando come effetto collaterale che se la variabile initVar è già impostata, questa funzione chiama solo la funzione CAN_Enable(). Un discorso del tutto analogo, ovviamente con diverse considerazioni, può essere fatto sia per la funzione CAN_Stop() che per ogni altra. Queste indicazioni fornite nel datasheet di ogni componente riguardanti le varie routine API sono dal punto di vista operativo fondamentali e necessarie: tramite queste funzioni infatti, richiamabili nel file main.c del progetto, si stabilisce la routine operativa stessa del componente e del CY8CKIT-059 di conseguenza. Tramite le varie finestre di dialogo si settano semplicemente le caratteristiche di funzionamento del componente ma non vengono in alcun modo fornite indicazioni software sulle operazioni che deve eseguire. Le variabili di ritorno sono molto utili per verificare la corretta implementazione del codice principale e quindi favorirne il debug: conoscendo infatti solo le varie routine API con la loro descrizione non si potrebbe capire il buon esito o meno delle operazioni. Nel caso specifico della tesi trattata, si potrebbero riscontrare degli errori nel funzionamento del bus e sarebbe molto più difficile comprenderne le ragioni se non si fosse in possesso anche dei possibili valori di ritorno (con relativi significati) che le funzioni richiamate possono originare; appare ovvio quindi come sia molto più facile in questo modo individuare il punto preciso dove l'utente ha commesso errori di implementazione nel main.c e, nel caso in cui non ve ne siano, escludere la parte software come possibile origine dell'errore.

5.2.ISR-Interrupt Service Routines

Nel contesto di applicazione del componente CAN del kit CY8CKIT-059 in un CAN bus caratterizzato da un elevato numero di nodi e ad alto scambio di dati (elevato numero di messaggi scambiato nel fieldbus), a maggior ragione se il microprocessore in dotazione alla schedina in questione debba compiere numerose operazioni in sequenza e gestire diverse periferiche, diventa fondamentale l'utilizzo delle ISR (Interrupt Service Routines). Esse non sono altro che una successione di istruzioni, più in particolare definite come funzioni informatiche di tipo callback, che il microprocessore esegue in risposta al verificarsi di un determinato interrupt; dovendo però la CPU avviare una ISR specifica in risposta a quel preciso evento, deve contemporaneamente abbandonare i compiti che stava eseguendo. Prima di avviare la interrupt service routine deve salvare il contesto di funzionamento (deve eseguire appunto una commutazione di contesto) affinché una volta terminata la ISR possa ripartire dal punto preciso in cui si era interrotta la routine principale; si dice allora che la routine di servizio dell'interrupt è perfettamente rientrante. Tutto questo processo deve impiegare poco tempo dato che non si può distogliere la CPU per troppo tempo dalla propria main routine; andranno allora salvate solamente quelle variabili e registri che verranno modificate dalla ISR. A tal considerazione però si aggiunge la problematica di conoscere a priori cosa salvare o meno; è opportuno allora, in questo contesto di incertezza, salvare almeno il registro PC (contiene l'indirizzo di memoria della prossima istruzione della main routine) e quello che definisce lo stato dell'interrompibilità del processore. Nel caso specifico del componente CAN in esame ci sono diverse sorgenti d'interrupt possibili, tra cui le principali sono:

- messaggio trasmesso, il messaggio in coda è stato inviato
- messaggio ricevuto, dispone di un gestore speciale che richiama le funzioni appropriate distinguendo le mailboxes Full da quelle Basic
- il buffer di ricezione è pieno, è stato ricevuto un messaggio ma non c'è più spazio per salvarlo
- CAN bus allo stato off
- rilevato errore CRC
- rilevato errore di Acknowledge
- rilevato errore di bit stuffing
- rilevata perdita di arbitraggio del bus mentre si stava inviando un messaggio.

Tutte queste sorgenti di interrupt hanno punti d'ingresso (funzioni) in modo da potervi inserire all'interno il codice desiderato. Tutte queste funzioni sono compilate in modo condizionato a seconda del customizer. Una volta che si è deciso quale dei precedenti eventi elencati costituirà la fonte di interrupt andando a spuntare la relativa opzione nella finestra di dialogo Interrupt Tab del componente, l’evento può essere utilizzato per diversi scopi, tra cui:

- controllo hardware della logica degli eventi di interrupt; la linea di interruzione hardware può essere utilizzata per eseguire semplici operazioni come la stima del carico del CAN bus. Abilitando gli interrupt Message Transmitted e Message Received nel customizer del componente CAN, e collegando la linea di interrupt ad un contatore, è possibile valutare il numero di messaggi che si trovano sul bus durante uno specifico intervallo di tempo. Inoltre, le azioni possono essere eseguite direttamente nell'hardware se il tasso di messaggi è superiore ad un certo valore
- interruzione dell'interazione di uscita con il DMA (Direct Memory Acces, permette a sottosistemi quali periferiche l’accesso diretto alla memoria senza coinvolgere la CPU nel processo); il componente CAN non supporta il funzionamento DMA internamente, ma è possibile collegare il componente DMA tramite finestra di lavoro del PSoC Creator alla linea di interrupt esterna (se è abilitata). L’utente è responsabile poi della configurazione e del funzionamento del DMA. Inoltre, si dovrebbe tenere presente che è necessario gestire alcune attività di housekeeping (e.g. la conferma del messaggio e la cancellazione dei flag di interrupt) in codice per una corretta gestione degli interrupt CAN. Con un trigger DMA hardware è possibile gestire i registri e i trasferimenti di dati quando si verifica un interrupt Message Received, senza che venga eseguito alcun firmware nella CPU. Questo è utile anche quando si gestiscono messaggi RTR. L'interrupt Message Transmitted può essere utilizzato per attivare un trasferimento DMA per ricaricare la mailbox dei messaggi con nuovi dati, senza l'intervento della CPU
- routine di servizio personalizzato per interrupt esterni; le ISR esterne personalizzate possono essere utilizzate in aggiunta o in sostituzione di quelle interne già compilate dal PSoC Creator. Quando quest’ultima opzione si verifica, la priorità di interrupt può essere impostata per determinare quale ISR deve essere eseguita per prima (interna o esterna), forzando così le azioni prima o dopo di quelle codificate nella routine di interrupt interna. Quando la ISR

esterna viene utilizzata in sostituzione di quella interna, l'utente si assume tutta la responsabilità per la corretta gestione dei registri e degli eventi della CAN.

Le impostazioni dell'output di interrupt del componente CAN consentono all'utente di abilitare o disabilitare una linea di interrupt esterna, disattivare, bypassare o personalizzare completamente la ISR interna ma anche abilitare o disabilitare la chiamata di specifiche funzioni di gestione di interrupt quando i relativi eventi generanti sono selezionati. La linea di interrupt esterna è visibile nello spazio di lavoro di PSoC Creator solo se abilitata nel customizer. Se è collegato un componente di interrupt esterno, esso non viene avviato come parte dell'API CAN_Start() e dovrà essere avviato al di fuori di tale routine. Se inoltre tale sorgente esterna è collegata al componente CAN e l'ISR interna non è disabilitata o bypassata, allora due generatori di eventi di interrupt risultano collegati alla stessa linea e quindi, seppur separati, gestiranno gli stessi eventi di interrupt. Questa è una situazione molto particolare ed inusuale, oltre che nella maggior parte dei casi indesiderabile. Se la ISR interna è disabilitata o bypassata (usando un'opzione di personalizzazione) il componente di interrupt interno sarà rimosso durante il processo di costruzione del componente CAN. Se si sceglie di disabilitare una chiamata di una funzione di interrupt nella ISR interna (per un evento di interrupt abilitato, utilizzando un'opzione del customizer), il blocco dell'interrupt CAN si attiva (quando si verifica l'evento rilevante), ma non viene eseguita alcuna chiamata di funzione interna nella routine interna CAN_ISR (parte di codice precompilata dal PSoC Creator quando si attivano degli interrupt per il componente CAN). Un esempio di caso d'uso è quando si vuole gestire un evento specifico (per esempio, un messaggio ricevuto) attraverso un percorso diverso dalla chiamata di funzione standard dell'utente (e.g. quando si vuole gestire l'interrupt attraverso DMA). Se si sceglie di personalizzare completamente la ISR interna (tramite l'opzione customizer) la funzione CAN_ISR non conterrà alcuna chiamata di funzione.

Per arrivare ad una comprensione ancora più profonda delle operazioni da eseguire per lavorare con il componente CAN messo a disposizione dal kit CY8CKIT-059, PSoC Creator mette a disposizione anche una serie di esempi scaricabili da browser. Non serve fare altro che andare nel workspace dove si trascinano i vari componente i che si vogliono utilizzare, si clicca con il tasto destro sul componente in uso e si seleziona la voce “Find Code Example” nella tendina che si apre. Si arriva quindi ad ottenere la finestra di dialogo di figura 5.2.1 che rappresenta l’intera lista degli esempi a disposizione, filtrati secondo il componente di interesse e scaricabili in pochi istanti.

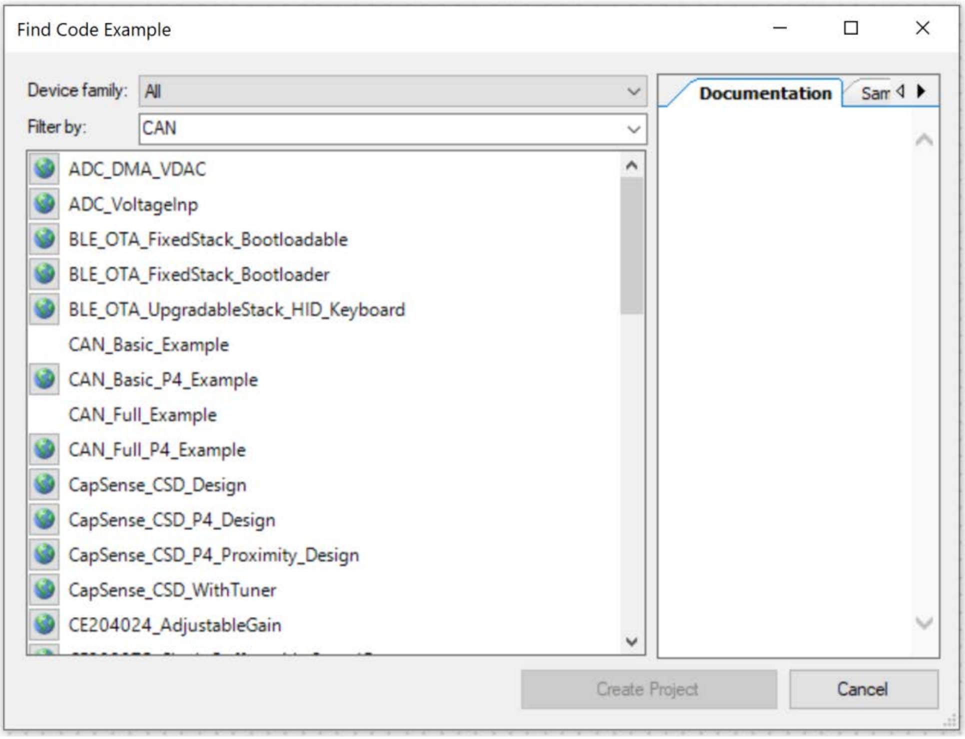


Figura 5.2.1: lista di esempi di codice che PSoC Creator mette a disposizione per avere una comprensione più chiara e approfondita di come poter utilizzare i componenti

Gli esempi di progetto includono non solo la parte di codice implementata nel file main.c, ma anche gli schemi adottati nel workspace, così da avere un’ampia panoramica sia sulla parte software (routines API, funzioni richiamate) che su quella hardware (collegamento dei pin, setting dei componenti).

6. UART-Universal Asynchronous Receiver/Transmitter

UART è l'acronimo di Universal Asynchronous Receiver/Transmitter; non è un protocollo di comunicazione come SPI e I2C, ma un circuito fisico in un microcontrollore, o un IC (Integrated Circuit, circuito integrato) stand-alone [10]. Lo scopo principale di questo componente è quello di trasmettere e ricevere dati seriali in modo asincrono, ovvero senza alcuna sincronizzazione fra i due moduli che stanno comunicando: trasmittente e ricevente non si inviano quindi alcun segnale che li mantenga sincronizzati. Se da un lato questa caratteristica comporta il vantaggio di una comunicazione più semplice, dall’altro può avere implicazioni negative quali la possibile deriva in frequenza della trasmissione stessa, entrambi concetti che verranno approfonditi nel seguito. In questa particolare tipologia di comunicazione due UART comunicano direttamente tra loro: l'UART trasmittente (i.e. ha dati da inviare) converte i dati paralleli di un dispositivo di controllo, come una CPU, in forma seriale, li trasmette in questa forma all'UART ricevente (i.e. ha dati da ricevere) che poi riconverte i dati seriali in dati paralleli per il dispositivo ricevente. Sono necessari solo due fili per trasmettere i dati tra due UART, uno di riferimento (massa) rispetto all’altro che viene invece sottoposto ad una certa tensione, e questo rappresenta una dei principali vantaggi dell’utilizzo di tale comunicazione. I valori di tale tensione applicata sono due: 0 V codifica lo zero logico, l’altro diverso da zero (solitamente di modulo attorno a qualche Volt) l’uno logico. La trasmissione seriale stessa allora si traduce dal punto di vista pratico nell’alternare tali valori di tensione sui cavi affinché il ricevente ricostruisca i bits del messaggio comunicato e ne decodifichi le informazioni. I dati fluiscono dal pin Tx (trasmissione) del componente trasmittente al pin Rx (ricezione) di quello ricevente. Gli UART trasmettono i dati in modo asincrono, il che significa che non c'è un segnale di clock per sincronizzare l'uscita dei bit dall'UART trasmittente al campionamento dei bit da parte dell'UART ricevente; ognuno dei due avrà un differente clock interno indipendente dall’altro e sono proprio tali clock a determinare l’alternanza tra due possibili valori di tensione sulla linea di trasmissione, andando così a stabilire la tempistica (velocità, frequenza) della comunicazione, i.e. il baud rate (bits per second). Il ricevitore allora deve sincronizzarsi su ogni frame ricevuto, sia un carattere o un byte, all’inizio del pacchetto dati stesso. Invece di un segnale di clock, l'UART trasmittente aggiunge un bit di avvio e uno di arresto al pacchetto di dati che viene trasferito, i cosiddetti bits di protocollo, fondamentali per garantire una corretta comunicazione fra dispositivi ma di fatto causa di un rallentamento della trasmissione stessa; la velocità effettiva dunque sarà minore del baud rate dichiarato e concordato fra i dispositivi, a causa di tale overhead (necessità di aggiungere dei bits superflui al messaggio ma necessari alla comunicazione). Questi bit definiscono l'inizio e la fine del pacchetto di dati in modo che l'UART ricevente sappia quando iniziare a leggere i bit. Quando l'UART ricevente rileva un bit di inizio, inizia a leggere i bit in entrata ad una frequenza specifica nota come baud rate. Il baud rate è una misura della velocità di trasferimento dati, espressa in bit al secondo (bps, bits per second). Entrambi gli UART devono operare all'incirca alla stessa velocità di trasmissione e questa è l’unica prerogativa da soddisfare per far operare correttamente questa tipologia di comunicazione. Il baud rate tra gli UART di trasmissione e di ricezione può differire solo del 10% circa prima che la temporizzazione dei bit si allontani troppo e il ricevente sbagli a campionare i bits in serie ricevuti. Sia ricevente che trasmittente inoltre devono essere configurati per trasmettere e ricevere una stessa struttura di pacchetto dati.

Per ricapitolare quindi le informazioni appena introdotte, la comunicazione UART è suddivisa nelle seguenti fasi:

- scelta del baud rate, ovvero della velocità (frequenza) di comunicazione (trasmissione/ricezione) da condividere su entrambi i dispositivi
- divisione del messaggio da inviare (costituito da un numero generico di bit) in stringhe più corte (a seconda della lunghezza scelta per il data field), creando così diversi pacchetti dati
- affiancamento ad ogni pacchetto dati (per ora composto dal solo data field) dei bits di protocollo (start bit, bit di parità opzionale e stop bit)

Ovviamente tutte le scelte intraprese in fase di configurazione e progettazione della comunicazione devono poi essere condivise dai dispositivi comunicanti, dato che la successiva decodifica dei dati da parte del ricevente deve rispettare il protocollo seguito dal trasmittente per impacchettare i dati.

6.1.Principio di funzionamento

L'UART che deve trasmettere generalmente riceve i dati da un bus dati, utilizzato per inviare dati all'UART da un altro dispositivo come una CPU, una scheda di memoria o un microcontrollore. Dopo che l'UART trasmittente riceve i dati in parallelo dal bus dati, aggiunge un bit di avvio, un bit di parità e un bit di stop, creando il pacchetto di dati secondo il formato richiesto da questa tipologia di comunicazione. Successivamente, il pacchetto dati viene emesso in serie, bit dopo bit, al pin Tx del trasmittente. L'UART ricevente legge il pacchetto di dati bit per bit al suo pin Rx, essendo al corrente della velocità (frequenza) di trasmissione. L'UART ricevente converte quindi i dati in forma parallela e rimuove il bit di avvio, il bit di parità e i bit di stop. Infine, l'UART ricevente trasferisce il pacchetto di dati in parallelo al bus dati sul lato ricevente:

I dati trasmessi da un UART sono organizzati in pacchetti, ognuno dei quali contiene uno start bit, da 5 a 9 bits di dati (a seconda del dispositivo UART in uso e dalla sua configurazione), un bit di parità (opzionale) e 1 o 2 bits di stop come si evince dalla figura 6.1.1.

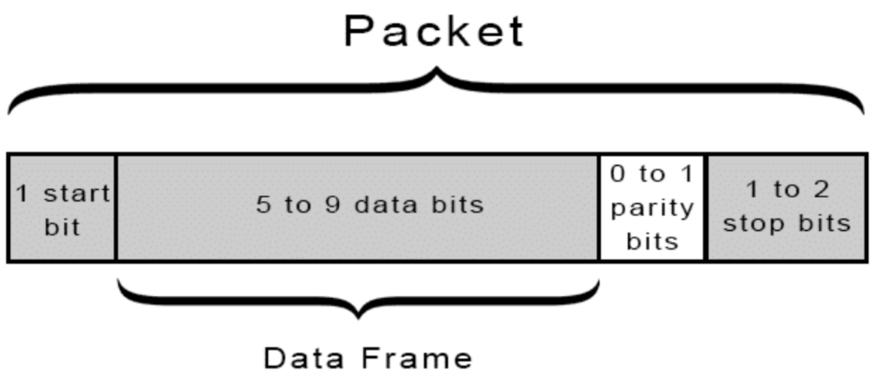


Figura 6.1.1: formato di un data frame nella comunicazione UART

La linea di trasmissione dati UART viene normalmente mantenuta ad un livello di alta tensione quando non trasmette dati. Per avviare il trasferimento dei dati, l'UART trasmittente crea sulla linea di trasmissione un fronte dall'alto al basso per la durata di un bit. Quando l'UART ricevente rileva la transizione da alta a bassa tensione (start bit), inizia a leggere i bit nel frame dei dati alla frequenza della velocità di trasmissione. Il data frame contiene i dati effettivi che vengono trasferiti: può essere lungo da 5 bit fino a 8 bit se viene utilizzato un bit di parità. Se non viene utilizzato alcun bit di parità, il data frame può essere lungo 9 bit. Nella maggior parte dei casi, i dati vengono inviati con il bit meno significativo in prima posizione.

Il parity bit descrive la parità o disparità di un numero; è un modo per l'UART ricevente di capire se i dati sono cambiati durante la trasmissione e questo può accadere per diversi motivi. I bit possono essere modificati da radiazioni elettromagnetiche, velocità di trasmissione non corrispondenti o trasferimenti di dati a lunga distanza. Dopo che l'UART ricevente legge il frame dei dati, conta il numero di bit con un valore di 1 e controlla se il totale è un numero pari o dispari. In fase di strutturazione del frame da inviare si decide che tipo di parità utilizzare. Se si sceglie parità pari si contano tutti i bit 1 all'interno del data field del messaggio: se tale numero è pari allora il parity bit viene impostato a 0, se invece è dispari a 1. In questo modo il numero di bit di valore 1 considerando sia il campo dati che il bit di parità risulta sempre essere un numero pari. Se si sceglie invece parità dispari si contano tutti i bit 1 all'interno del data field del messaggio: se tale numero è pari allora il parity bit viene impostato a 1, se invece è dispari a 0. In questo modo il numero di bit di valore 1 considerando sia il campo dati che il bit di parità risulta sempre essere un numero dispari. Quando il bit di parità corrisponde ai dati, l'UART sa che la trasmissione è stata esente da errori. Ma se il bit di parità è uno 0, e il totale è dispari; o il bit di parità è un 1, e il totale è pari, la UART sa che i bit nel data frame sono cambiati. Nella realtà questo non è un metodo di rilevazione infallibile, anche se nel complesso accurato: potrebbe infatti accadere che venga alterato più di un singolo bit, e.g. due bit di valore 1 per i motivi sopra indicati potrebbero mutare in 0 entrambi. Il controllo del bit di parità non rileverebbe l'errore nonostante sia presente. Ad ogni modo l'alterazione di un numero pari di bits è un evento molto raro e questo rende molto consolidato questo metodo di diagnostica del frame.

Per concludere il frame e segnalare la fine del pacchetto dati, l'UART trasmittente commuta la linea di trasmissione dati da una bassa tensione ad un'alta tensione per almeno la durata di un bit. Il bit di stop serve inoltre a riconoscere il bit di start del pacchetto successivo scambiato sulla linea seriale.

Per completare l'introduzione all'UART è necessario anche dibattere sui vantaggi e svantaggi di questa tipologia di comunicazione, dato che non esistono dei protocolli di comunicazione perfetti nella loro interezza. Da un lato i vantaggi che questo metodo presenta sono:

- utilizzo di due soli cavi per la comunicazione (cavo di terra escluso), uno di ricezione Rx e uno di trasmissione Tx
- invio del segnale di clock da parte del trasmittente insieme al pacchetto dati non necessario
- presenza del bit di parità come metodo di rilevazione dell'errore
- possibilità di variare la struttura del pacchetto dati fintanto che si può impostare ognuno dei due UART coinvolti nella comunicazione
- l'essere di fatto un metodo ben documentato e ampiamente utilizzato, e dunque di comprovata efficacia.

Tuttavia, questa tipologia di comunicazione presenta anche degli svantaggi, tra cui:

- grandezza del data frame limitata a nove bits
- supporta solo un master (padrone della comunicazione, decide quando inviare un messaggio e quando è pronto a riceverne uno) e uno Slave (subisce la comunicazione, risponde al master) e quindi non dei sistemi multipli
- la velocità di trasmissione di trasmittente e ricevente possono differire al massimo del 10% tra loro se non si vuole compromettere la comunicazione

7. Componente UART del kit CY8CKIT-059 di Cypress

Oltre al componente CAN approfondito precedentemente, è necessario focalizzare l'attenzione anche sul componente UART che il kit CY8CKIT-059 di Cypress Semiconductor mette a disposizione dell'utente, raffigurato in figura 7.1 [11].

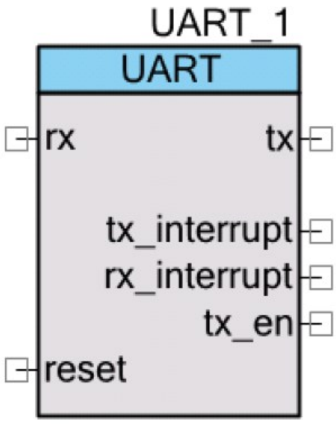


Figura 7.1: interfaccia grafica in PSoC Creator del componente UART

L'UART fornisce comunicazioni asincrone comunemente denominate standard RS232 o standard RS485. Il componente può essere configurato per le versioni Full Duplex (la trasmissione avviene contemporaneamente nella due direzioni Rx e Tx), Half Duplex (la trasmissione avviene alternativamente nella due direzioni Rx e Tx), oppure Simplex (solo Rx o solo Tx). Per facilitare l'elaborazione dei dati di ricezione e trasmissione da parte dell'UART, vengono forniti dei buffer (locazioni di memoria tampone) configurabili e di dimensioni indipendenti. Ciò consente alla CPU di dedicare più tempo alle attività critiche in tempo reale piuttosto che alla gestione e supporto dell'UART.

Per la maggior parte dei casi d'uso, è possibile configurare facilmente l'UART scegliendo il baud rate (bits per second), la parità, il numero di bit di dati e il numero di bit di partenza. La configurazione più comune per la RS232 è spesso elencata come "8N1", che costituisce l'abbreviazione di otto bit di dati, nessuna parità e un bit di stop; questa è la configurazione predefinita per il componente UART. Pertanto, nella maggior parte delle applicazioni è necessario impostare solo il baud rate e farlo combaciare con la frequenza di trasmissione del secondo dispositivo di comunicazione. Un secondo uso comune per gli UART è nelle reti RS485 multidrop. Il componente UART supporta un segnale di abilitazione tx_en dell'uscita TX per abilitare il trasmettitore TX durante le trasmissioni, ma fornisce anche un supporto di configurazione per il numero di bit di dati, bit di stop, parità, controllo del flusso hardware e generazione e rilevamento della parità.

Come si nota in figura 7.1, visivamente il componente mette a disposizione connessioni di diverso tipo, in particolare:

- Rx, input che porta i dati seriali in ingresso da un altro dispositivo sul bus seriale
- Tx, output che porta i dati seriali in uscita ad un altro dispositivo sul bus seriale
- Reset, input che resetta le macchine di stato UART (RX e TX) allo stato di inattività; in questo modo vengono eliminati tutti i dati attualmente trasmessi o ricevuti. Questo ingresso è un reset sincrono che richiede almeno un fronte di salita del clock interno. L'input di reset può essere lasciato flottante senza collegamento esterno: se non è collegato nulla alla linea di reset, il componente gli assegnerà una logica costante 0
- Rx_interrupt, output che costituisce l'OR logico del gruppo delle possibili sorgenti di interrupt; questo segnale va alto mentre una qualsiasi delle sorgenti di interrupt abilitate è vera
- Tx_interrupt, vedi Rx_interrupt
- Tx_en, output che viene utilizzato principalmente ad abilitare la trasmissione. Questa uscita va alta prima che inizi una trasmissione e bassa quando la trasmissione è completa; questo mostra un bus occupato al resto dei dispositivi collegati al bus stesso ed è un output visibile solo quando il parametro Hardware TX Enable è selezionato nell'interfaccia grafica del componente UART.

7.1.Parametri del componente

Una volta trascinato un componente UART nello spazio di lavoro diPSoC Creator, cliccandovi sopra due volte con il tasto sinistro del mouse si apre la finestra di Configure come si vede in figura 7.1.1.

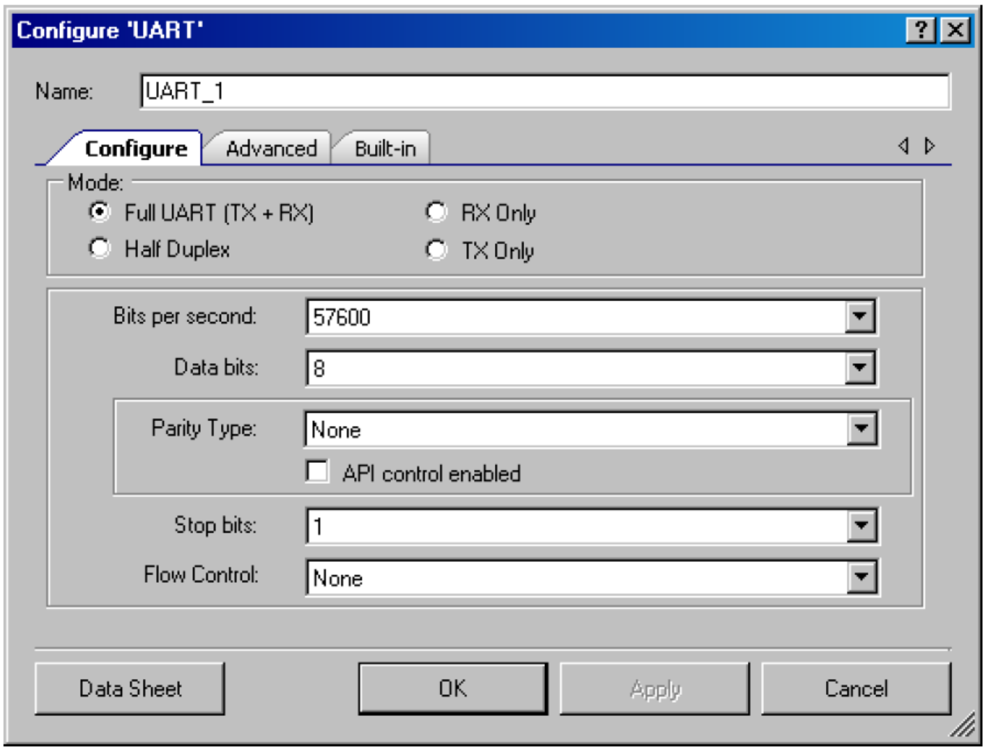


Figura 7.1.1: finestra di dialogo generale per settare il componente UART

Questa prima finestra di dialogo, nota come Configure Tab, è impostata in modo da assomigliare ad una finestra di configurazione dell'iper-terminale (programma per il debug della trasmissione dei dati sulle porte seriali) per evitare una configurazione errata di due lati del bus, perché il PC che utilizza l'iper-terminale è molto spesso l'altro lato del bus. Si può dapprima configurare il modo operativo dell'UART, scegliendo i componenti funzionali che si vogliono includere in esso; in particolare si può optare per una comunicazione bidirezionale Full Duplex (Tx + Rx, di default), Half Duplex (si utilizzano metà delle risorse) oppure per il protocollo unidirezionale fisso RS232 (solo Rx ricevente o Tx trasmittente). Si possono poi settare i bits per second, ovvero il baud-rate (velocità di trasmissione dati) che di default il valore impostato è 57600, ma anche i data bits (bits contenuti nel data field del data package tra quello di start e quello di stop), di default 8 (corrispondente di fatto un byte di dati). Vi è poi l'impostazione del tipo di parità; si può decidere se non utilizzare tale metodo di rilevazione dell'errore selezionando "None" oppure in caso contrario se affidarsi ad una parità pari ("Even") o dispari ("Odd"). L'utente può poi stabilire se utilizzare 1 o 2 bits di stop e se affidare il controllo di flusso all'hardware oppure non eseguirlo affatto.

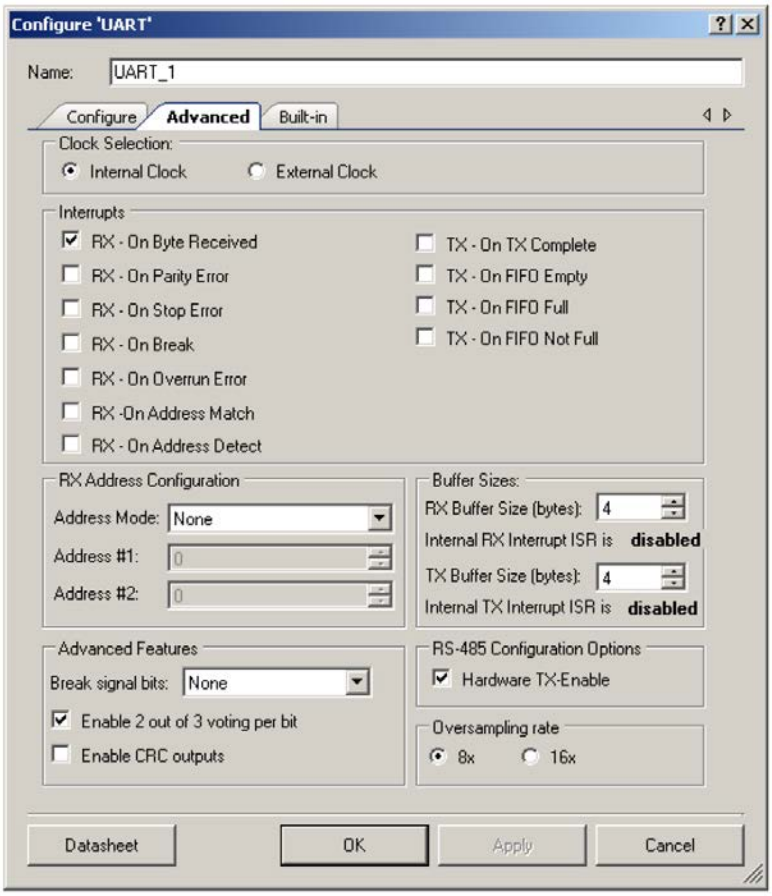


Figura 7.1.2: finestra avanzata dal componente UART

La seconda finestra di dialogo, che si può vedere in figura 7.1.2, è invece un'Advanced Tab per il settaggio avanzato del componente UART: consente la configurazione di opzioni sia hardware che software. Per quanto riguarda quelle hardware, è possibile selezionare il clock ovvero scegliere tra un clock configurato internamente o un clock-I/O configurato esternamente per la generazione del baud-rate. Si può poi selezionare la tipologia di Address Mode desiderata, ovvero il modo in cui hardware e software interagiscono per gestire gli indirizzi del dispositivo e i bytes di dati. Vi sono poi delle caratteristiche ancora più avanzate, tra cui la scelta del numero di bits del break signal (segnale di interruzione, composto da bits di valore 0), l'abilitazione o meno dell'algoritmo di compensazione dell'errore ("Enable 2 out of 3 voting per bit") e del segnale di output tx_en, utilizzato solitamente nelle comunicazioni RS485, oltre che la scelta del tasso di sovra-campionamento (Oversampling Rate, parametro che permette di scegliere il divisore di clock per la generazione del baud-rate). Per quanto concerne invece la configurazione delle opzioni software, l'Advanced Tab permette di configurare le diverse sorgenti di interrupt: si può gestire la ISR (Interrupt Service Routine) con un componente di interrupt esterno collegato all'output tx_interrupt o rx_interrupt. Il pin dell'uscita di interrupt è visibile a seconda del parametro Mode selezionato: esso emette lo stesso segnale all'interrupt interno in base agli interrupt di stato selezionati tra quelli disponibili in figura 7.1.2. Questi output possono poi essere utilizzati come sorgenti di richiesta DMA per il DMA (Direct Memory Access) stesso dal buffer Rx o Tx indipendentemente dall'interrupt, o come un generico altro interrupt, a seconda della funzionalità desiderata. Si possono inoltre configurare le dimensioni dei buffer Rx e Tx, definendo rispettivamente quanti bytes di memoria RAM allocare per quel determinato buffer. Per quanto riguarda la ricezione, i dati sono spostati dai registri FIFO di ricezione al buffer Rx mediante le funzioni API UART_GetChar() o UART_ReadRXData(), mentre per la trasmissione i dati sono scritti nel registro Tx mediante i comandi API UART_PutChar() e UART_PutArray(). Tuttavia, dato che sono utilizzati quattro bytes di hardware FIFO come buffer quando la dimensione di quest'ultimo è scelta essere di quattro bytes, la selezione di dimensioni maggiori di questo valore richiede obbligatoriamente l'utilizzo di interrupts per gestire lo spostamento dei dati tra hardware FIFO e il buffer, e viceversa.

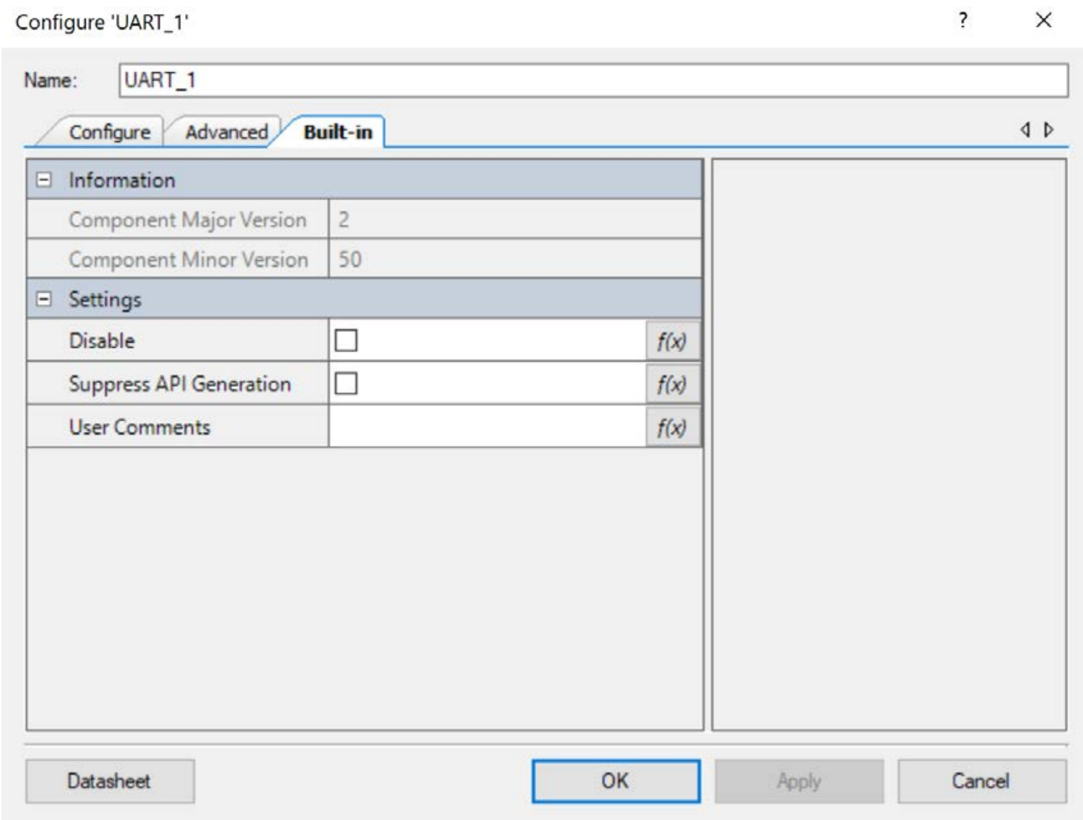


Figura 7.1.3: finestra di dialogo Built-in

L'ultima finestra di dialogo è quella di Built-in, visibile in figura 7.1.3, che, analogamente a quanto visto in precedenza per il componente CAN, permette all'utente di modificare i Settings, in particolare si può scegliere se disabilitare o meno la possibilità stessa di modificare i parametri del componente (come spiegato in precedenza tramite le altre finestre di dialogo già introdotte) ma anche sopprimere la generazione automatica delle API, eseguita di default in background dal PSoC Creator in seguito ad ogni settaggio implementato tramite interfaccia grafica.

Così come già illustrato a riguardo del componente CAN nel precedente capitolo, anche il componente UART messo a disposizione dal kit CY8CKIT-059 di Cypress Semiconductor può essere configurato via software sfruttando le routines API (Application Programming Interface) implementate per questo componente in particolare. Sono ancora valide le considerazioni sviluppate in precedenza, dato che le modalità operative, intese come scopo e concetto di applicazione, di tali funzioni non cambiano; servono sempre a determinare via software, in particolare includendole nel file main.c del progetto in esame in PSoC Creator, la routine operativa del relativo componente. Ciò che varia semplicemente da un componente all'altro sono le funzioni API disponibili, che saranno caratterizzate da diversi nomi, argomenti, valori di ritorno ed effetti collaterali. Pur non ripetendo quindi quanto già detto nel precedente capitolo sul componente CAN per quanto riguardava le routines API, per dovere di completezza si aggiunge in figura 7.1.4 una parte delle funzioni disponibili per il componente UART a titolo esemplificativo. La lista completa può facilmente essere visionata nel datasheet del componente stesso.

Function	Description
UART_DisableRxInt()	Disables the internal interrupt irq
UART_SetRxInterruptMode()	Configures the RX interrupt sources enabled
UART_ReadRxData()	Returns the data in the RX Data register
UART_ReadRxStatus()	Returns the current state of the status register
UART_GetChar()	Returns the next byte of received data
UART_GetByte()	Reads the UART RX buffer immediately and returns the received character and error condition
UART_GetRxBufferSize()	Returns the number of received bytes remaining in the RX buffer and returns the count in bytes
UART_ClearRxBuffer()	Clears the memory array of all received data
UART_SetRxAddressMode()	Sets the software-controlled Addressing mode used by the RX portion of the UART
UART_SetRxAddress1()	Sets the first of two hardware-detectable addresses
UART_SetRxAddress2()	Sets the second of two hardware-detectable addresses
UART_EnableTxInt()	Enables the internal interrupt irq
UART_DisableTxInt()	Disables the internal interrupt irq
UART_SetTxInterruptMode()	Configures the TX interrupt sources enabled
UART_WriteTxData()	Sends a byte without checking for buffer room or status
UART_ReadTxStatus()	Reads the status register for the TX portion of the UART
UART_PutChar()	Puts a byte of data into the transmit buffer to be sent when the bus is available
UART_PutString()	Places data from a string into the memory buffer for transmitting
UART_PutArray()	Places data from a memory array into the memory buffer for transmitting
UART_PutCRLF()	Writes a byte of data followed by a Carriage Return and Line Feed to the transmit buffer
UART_GetTxBufferSize()	Determines the number of bytes used in the TX buffer. An empty buffer returns 0
UART_ClearTxBuffer()	Clears all data from the TX buffer
UART_SendBreak()	Transmits a break signal on the bus
UART_SetTxAddressMode ()	Configures the transmitter to signal the next bytes as address or data
UART_LoadRxConfig()	Loads the receiver configuration. Half Duplex UART is ready for receive byte
UART_LoadTxConfig()	Loads the transmitter configuration. Half Duplex UART is ready for transmit byte
UART_Sleep()	Stops the UART operation and saves the user configuration

Figura 7.1.4: lista parziale delle funzioni API che sono disponibili in PSoC Creator, operanti solo se il componente UART è chiamato “UART”, visto il prefisso utilizzato in tabella

Un altro aspetto molto importante da considerare è la descrizione funzionale del componente UART; è implementato nei blocchi UDB (Universal Digital Block) secondo lo schema di figura 7.1.5.

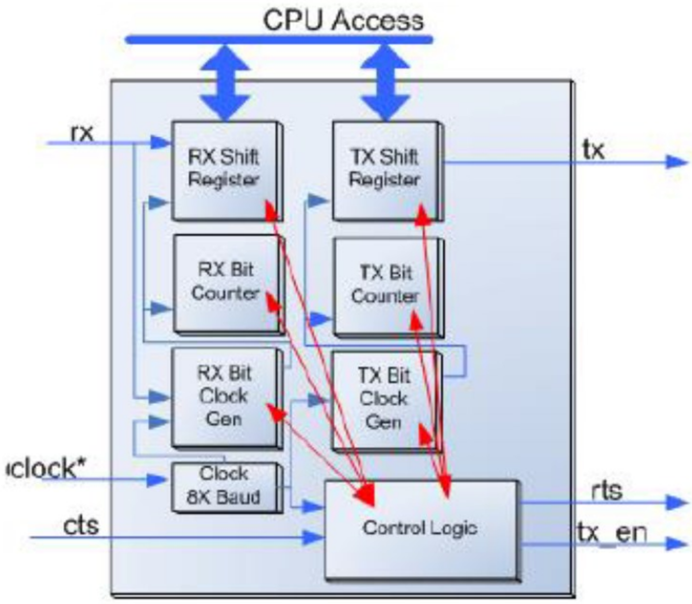


Figura 7.1.5: diagramma a blocchi del funzionamento del componente UART del kit CY8CKIT-059

La configurazione di default per l'UART è quella ad 8 bit senza controllo di flusso né parità, operante ad un baud-rate di 57600 bps (bits per second). I modi di funzionamento implementabili sono diversi, in particolare:

- Full UART (RX+TX), UART full-duplex composto da un ricevitore e un trasmettitore asincrono; in questa modalità è necessario un singolo clock per definire il baud rate sia per il ricevitore che per il trasmettitore
- Half Duplex, implementa un UART completo ma utilizza la metà delle risorse rispetto alla configurazione Full UART; in questa configurazione, l'UART può essere configurato per passare dal modo Rx al modo Tx, ma non può eseguire operazioni Rx e Tx contemporaneamente. La configurazione Rx o Tx può essere caricata chiamando la funzione UART_LoadRxConfig() o UART_LoadTxConfig()
- solo Rx, implementa solo la parte ricevente dell'UART; è necessario un singolo clock per definire il baud rate del ricevitore

- solo Tx, implementa solo la parte trasmittente dell'UART; è necessario un singolo clock per definire il baud rate del trasmittente
- nessun controllo hardware di flusso dell'UART; il controllo di flusso sull'UART fornisce linee di indicazione di stato Rx e Tx separate del bus esistente. Quando il controllo di flusso hardware è abilitato, tra un UART ed un altro sono disponibili una linea "Request to Send" (RTS, richiesta di invio) e una linea "Clear to Send" (CTS, pronto ad inviare); la linea CTS è un input dell'UART che viene impostato dall'altro UART nel sistema quando è pronto ad inviare i dati sul bus. La linea RTS è un output dell'UART che informa l'altro UART sul bus che è pronto a ricevere dati. La linea RTS di un UART è collegata alla linea CTS dell'altro e viceversa. Queste linee sono valide solo prima dell'avvio di una qualsiasi trasmissione: se il segnale viene impostato o cancellato dopo l'avvio di un trasferimento, la modifica avrà effetto solo sul trasferimento successivo
- nessuna parità, in questa modalità non viene inserito nei data package alcun bit di parità e quindi il flusso dati è costituito da partenza, dati e stop trasmissione
- parità pari o dispari, viene inserito un bit di parità uguale a 0 o 1 in base al numero di bit di valore 1 conteggiati all'interno del data field e alla parità scelta come illustrato in precedenza; metodo di rilevazione di errori di trascrizione/lettura dei bits in una trasmissione
- uno o due bits di stop; il numero di bit di stop è disponibile come meccanismo di sincronizzazione; nei sistemi più lenti, a volte è necessario che il comando di arresto occupi due tempi di bit per consentire al lato ricevente di elaborare i dati prima che ne vengano inviati altri. Inviando due bit del segnale di stop, il trasmettitore permette al ricevitore di avere un tempo supplementare per interpretare il byte e la parità dei dati. Il secondo bit di stop non viene controllato dal ricevitore come un errore di framing; il flusso di dati è lo stesso, "Start, Data, [Parità eventuale], Stop". Il tempo del bit di stop può essere configurato su una o due bit.

Le funzioni API descritte in precedenza forniscono il supporto per le comuni operazioni di routine richieste per la maggior parte delle applicazioni; per un utente più esperto, ma anche per completezza di trattazione, è utile approfondire anche i registri UART. I registri di stato, indipendenti per la linea Rx e Tx, sono di sola lettura e contengono i vari bits di stato definiti per l'UART. Al valore di tali registri si può accedere usando le function calls UART_ReadRxStatus() e UART_ReadTxStatus(). I segnali output di interrupt tx_interrupt e rx_interrupt sono generati facendo passare per un componente logico OR i campi di bits mascherati all'interno di ogni registro; le maschere possono essere configurate usando le API UART_SetRxInterruptMode() e UART_SetTxInterruptMode(). Una volta ricevuto un interrupt, se ne può recuperare la sorgente leggendone il rispettivo registro di stato tramite UART_GetRxInterruptSource() e UART_GetTxInterruptSource(), che mantiene l'informazione riguardo la sorgente di interrupt fintanto che l'utente non richiama una delle functions precedentemente citate per leggerne il valore. I dati di stato vengono registrati ad ogni fronte del clock di input dell'UART. Molti di questi bit sono sticky e vengono cancellati alla lettura del registro di stato; essi sono assegnati come leggibili per essere utilizzati come output di interrupt per l'UART. Tutti gli altri bit sono configurati come trasparenti e rappresentano i dati direttamente dagli input del registro di stato; non sono sticky e quindi non sono clear-on-read. Oltre ai registri di stato esiste il registro di controllo, che permette all'utente di controllare e gestire il funzionamento generale dell'UART; per scrivere tale registro si deve ricorrere alla funzione UART_WriteControlRegister() mentre per leggerlo si deve usare UART_ReadControlRegister(). Questo particolare registro tuttavia non viene usato se l'utente seleziona opzioni UART semplici nel customizer; il suo utilizzo diretto è infatti consigliato solo a programmatori dotati di una certa esperienza e confidenza nella programmazione ed utilizzo del componente UART e con PSoC Creator. Gli ultimi due registri disponibili sono quelli dei dati Rx e Tx, distinti ed indipendenti tra loro. Il registro dati Tx (8 bits) contiene i dati da trasmettere ed è implementato come un FIFO (First In First Out); c'è una macchina di stato software per controllare i dati provenienti dal buffer di memoria di trasmissione per gestirne quantità maggiori. Tutte le funzioni che si occupano della trasmissione dei dati devono passare attraverso questo registro per poter inserire i dati sul bus; se ci sono dati in questo registro e il controllo di flusso indica che i dati possono essere inviati, allora i dati vengono trasmessi. Non appena questo registro (FIFO) è vuoto, non vengono trasmessi altri dati sul bus fino a quando non ne vengono aggiunti di nuovo (non vengono mai ritrasmessi gli stessi dati, a meno che non lo implementi l'utente via software nella routine del componente). Il DMA può essere impostato per riempire questo registro di dati Tx quando è vuoto, utilizzando l'indirizzo del registro dati TX definito mediante UART_TXDATA_PTR. Per quanto riguarda invece il registro dati Rx, implementato anch'esso come un FIFO, conterrà i dati ricevuti dall'UART, il cui movimento è regolato da una macchina di stato software. In genere l'interrupt Rx indica che i dati sono stati ricevuti; a quel punto possono essere recuperati sia con la CPU che con il DMA. Quest'ultimo può essere impostato per recuperare i dati da questo registro ogni volta che il FIFO non è vuoto utilizzando l'indirizzo del registro dati RX definito mediante UART_RXDATA_PTR.

La comunicazione seriale UART tra un kit di prototipazione CY8CKIT-059 ed un PC, caso d'interesse per questa tesi, avviene mediante il KitProg, un sistema multifunzionale che permette l'alimentazione del dispositivo target oltre che la sua programmazione e debug, come rappresentato in figura 7.1.6.

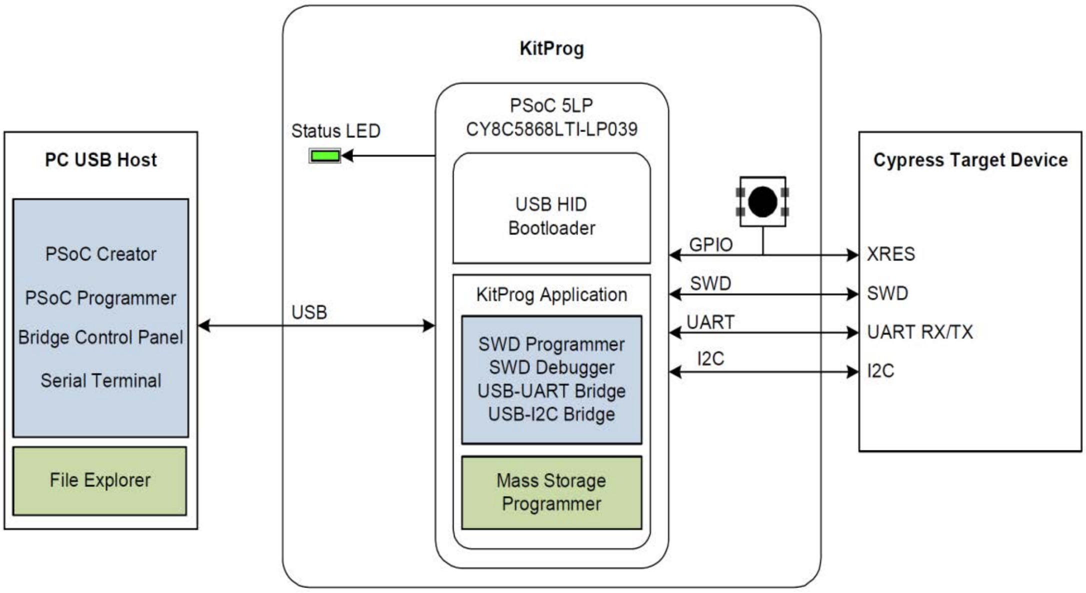


Figura 7.1.6: ecosistema del KitProg

Il KitProg in dotazione può agire da ponte USB-UART tra i due protocolli di trasmissione seriale (USB è l'acronimo infatti di Universal Serial Bus): a livello hardware le linee UART tra il KitProg e il dispositivo target sono cablate sulla scheda tra le due porzioni di PCB (Printed Circuit Board, circuito stampato) attraverso l'area distaccabile, con UART_RX assegnato a P12[6] e UART_TX assegnato a P12[7] su

PSoC 5LP (target); in questo modo crea una connessione RS-232 con un programma terminale del PC. Le linee UART non sono comprese nei cinque pin (SWDIO, SWDCLK, RSD, GND e VTARG) che collegano il KitProg al dispositivo target ma sono instradate al di fuori di essi; nel caso in cui il KitProg viene distaccato dalla scheda non è più possibile realizzare la comunicazione UART poiché vengono a mancare i collegamenti fisici tra CY8CKIT-059 e PC. A livello software l'unico requisito per instaurare correttamente la comunicazione UART mediante il ponte USB-UART sul KitProg è l'utilizzo di un qualsiasi programma emulatore di terminale (e.g. HyperTerminal, PuTTY oppure altro a seconda del sistema operativo del Personal Computer); le velocità di comunicazione supportate sono 1200, 2400, 4800, 9600, 19200, 38400, 57600 e 115200 Baud (bits per second) e tali valori devono essere utilizzati in fase di configurazione sia del componente UART del kit di prototipazione che poi del terminale del PC.

8. Sistema di monitoraggio distribuito basato sulla comunicazione CAN bus: progettazione e realizzazione del circuito hardware

Il primo circuito progettato e realizzato in questa tesi è una rete di comunicazione basata sul CAN bus per un sistema di monitoraggio distribuito della carica e scarica di una batteria. Nella sua interezza si può considerarlo concettualmente composto da tre sezioni, ognuna adibita a svolgere una specifica funzione:

- Circuito di acquisizione dei segnali analogici di tensione e corrente di ogni singola batteria tramite un kit CY8CKIT-059
- Circuito per la comunicazione e condivisione, da parte delle schede CY8CKIT-059 operanti da Slave, dei dati acquisiti dalle batterie tramite CAN bus
- Interfaccia seriale UART-USB tra il kit CY8CKIT-059 implementato come Master del CAN bus e il PC che restituisce poi i dati acquisiti in real time sotto forma di grafico

E' necessario evidenziare che le prime due sezioni costituiscono di fatto delle strutture modulari replicabili: nell'ottica di voler aumentare (diminuire) i nodi connessi sul CAN bus, e quindi il numero di batterie monitorate, si dovrà aggiungere (sottrarre) un equivalente numero di circuiti di acquisizione dei segnali analogici e circuiti di interfaccia tra kit di prototipazione e CAN bus senza dover riprogettare il sistema di monitoraggio distribuito, bensì applicando delle modifiche nella programmazione delle schede CY8CKIT-059. Questo consente di fatto un agevole utilizzo e gestione del sistema così progettato, facilmente modificabile e adattabile secondo le funzioni che deve soddisfare.

Ognuna delle tre sezioni appena introdotte, fondamentali per il corretto funzionamento del sistema di monitoraggio, è costituita da una parte hardware e una software, rispettivamente l'insieme dei componenti fisici e delle istruzioni necessari per l'acquisizione e la memorizzazione di segnali e dati, oltre che per lo svolgimento di funzioni specifiche.

Nel seguito della trattazione vengono dunque approfondite tali sezioni nello specifico, sia dal punto di vista del circuito fisico progettato e realizzato, sia per quanto concerne la programmazione in linguaggio C dei kit CY8CKIT-059 utilizzati.

8.1. Circuito di acquisizione dei segnali analogici di tensione e corrente della batteria

Il circuito necessario per acquisire la corrente e la tensione di ogni singola batteria è stato realizzato dal collega Samuel Matrella nella sua tesi “Studio di un sistema per il monitoraggio della carica degli accumulatori di un veicolo elettrico”, come già riportato in precedenza nell’antefatto di questa trattazione. Se ne riprenderà ora il principio di funzionamento e lo schema circuitale; la componente software verrà invece analizzata in seguito quando verrà presentato il sistema di monitoraggio distribuito nella sua interezza. Per quanto riguarda la scelta e la descrizione dei componenti utilizzati in tale circuito, si rimanda al documento appena citato in quanto non d’interesse per la presente trattazione.

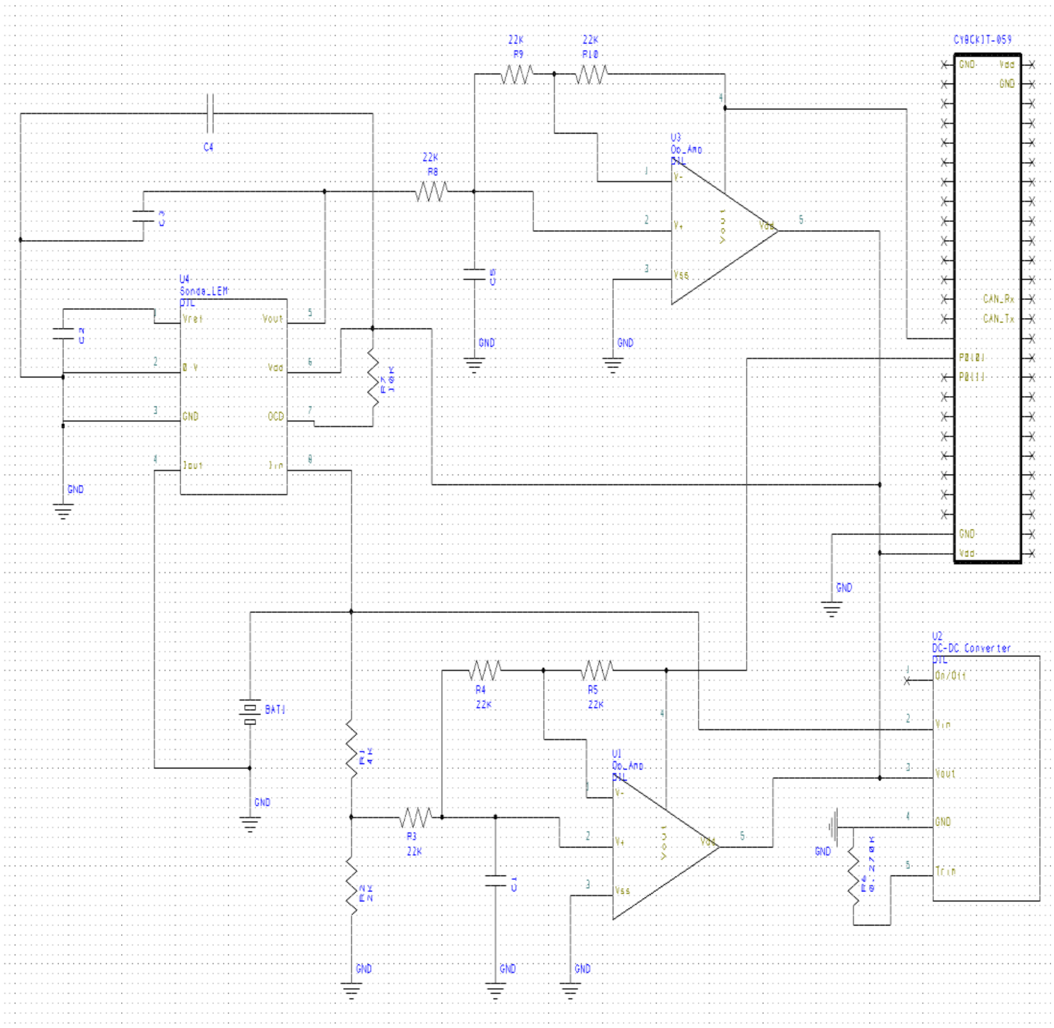


Figura 8.1.1: schema circuitale della sezione di acquisizione dei segnali analogici di tensione e corrente della batteria

I convertitori analogico-digitali ADC_SAR_Seq messi a disposizione dal kit CY8CKIT-059 utilizzato permettono l'acquisizione e la conversione di segnali analogici con un range di tensione in input compreso tra V_{SS} e V_{DD} della scheda. Per quanto concerne l'acquisizione del segnale di tensione, ricordando che le batterie al piombo acido a disposizione in laboratorio sono caratterizzate da tensione nominale pari a 12V ciascuna e tensione massima a fine ricarica di 14.9V, è necessario interporre tra esse e il kit di prototipazione di Cypress un partitore di tensione con guadagno $\frac{V_{out}}{V_{in}} = \frac{5V}{14.9V} = \frac{1}{3}$ realizzato utilizzando due resistenze. Il valore del guadagno del partitore impone la scelta del valore delle resistenze e la tensione in uscita viene filtrata da rumori e disturbi ad alte frequenze che possono interferire con l'acquisizione del segnale mediante il filtro passa-basso rappresentato in figura 8.1.2 e composto di tre resistenze, un condensatore collegato alla connessione di terra del circuito e un amplificatore operazionale. Il pin V_{DD} di quest'ultimo deve essere connesso all'alimentazione, che non deve superare i 5V per non danneggiare il kit CY8CKIT-059, mentre il pin V_{SS} viene collegato alla connessione di terra del sistema.

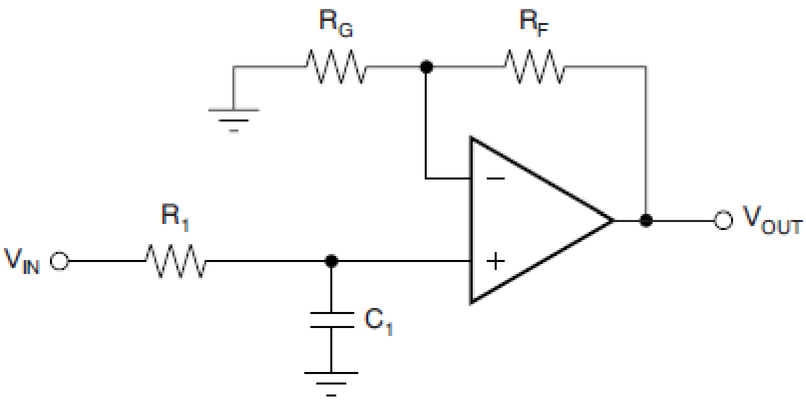


Figura 8.1.2: rappresentazione schematica del filtro passa-basso utilizzato

La frequenza di taglio scelta è pari a 100Hz e ciò impone i valori delle resistenze e del condensatore utilizzati per realizzare il filtro passa-basso; per maggiori informazioni riguardo i componenti scelti e i valori utilizzati si rimanda alla tesi realizzata dal collega Samuel Matrella [1]. Il segnale di tensione V_{OUT} in uscita dall'amplificatore operazionale, abbassato entro il range richiesto dal CY8CKIT-059 (5V tensione massima ammissibile in input), viene quindi portato in ingresso al pin P0[0] del kit stesso, cui è collegato il primo ingresso del convertitore ADC_SAR_Seq messo a disposizione dalla scheda.

Per quanto concerne invece l'acquisizione del valore di corrente della batteria si deve tenere a mente che nella prima fase della ricarica dell'accumulatore la corrente raggiunge un picco di 15.2A. Per misurarla si utilizza una sonda di tipo LEM dimensionata al primario per una I_{nom} di 15A e I_{max} di 37.5A; il valore di tensione che la sonda restituisce in risposta ad una corrente di ingresso varia linearmente secondo l'andamento riportato in figura 8.1.3.

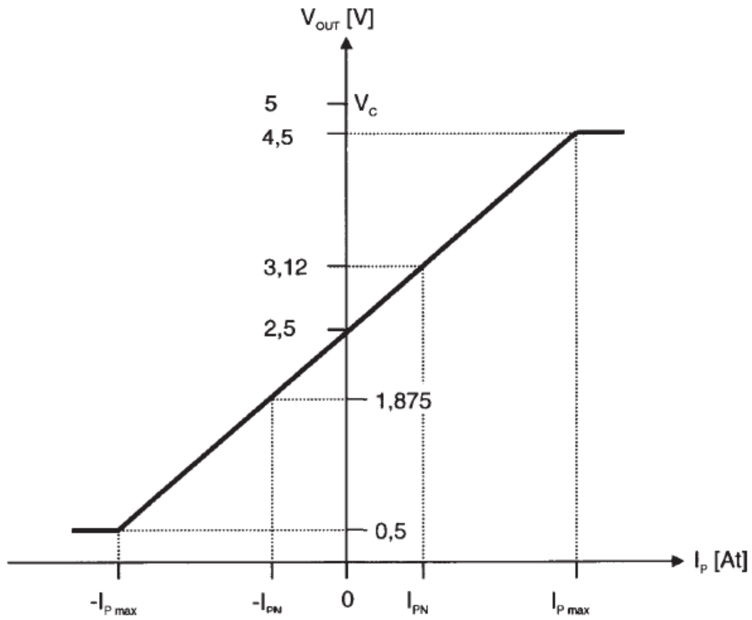


Figura 8.1.3: caratteristica V_{OUT}-$I_{INGRESSO}$ della sonda di tipo LEM utilizzata

È bene evidenziare come il segnale di tensione V_{OUT} restituito dalla sonda sia compreso in un range tra gli 0.5V e i 4.5V compatibile con i livelli di tensione che il kit CY8CKIT-059 ammette in ingresso ai suoi pin I/O. La sonda, così come l'amplificatore operazionale impiegato nel circuito di acquisizione della tensione della batteria, viene alimentata a 5V dalla batteria stessa tramite la scheda. Il segnale di tensione V_{OUT} corrispondente al valore di corrente rilevato dalla sonda viene quindi portato al pin P0[1] del kit CY8CKIT-059 cui fa capo il secondo ingresso del componente ADC_SAR_Seq della scheda.

Il kit CY8CKIT-059 connesso come appena illustrato ai due diversi circuiti di acquisizione è ora in grado di monitorare lo stato della batteria, acquisendone i valori di tensione e corrente in real time; si procede dunque ad implementare un circuito che permetta alla scheda operante da Slave di trasmettere i dati raccolti ad un dispositivo agente da Master del CAN bus del sistema distribuito di gestione della carica e scarica di un pacco batterie.

8.2. Circuito di comunicazione dei dati sul CAN bus

Una volta che ogni nodo Slave della rete, costituito da batteria, circuito di acquisizione di tensione e corrente e kit CY8CKIT-059, ha a disposizione i parametri monitorati, deve procedere a comunicarli alla scheda Master. Nell'ambito dei veicoli elettrici stradali, e più in generale nel settore automotive, per garantire al mezzo di trasporto un'elevata autonomia, a parità di tipologia di spazio disponibile e accumulatore scelto si deve massimizzare il numero di batterie utilizzate, aumentano di conseguenza, per quanto concerne il progetto qui trattato, i nodi della rete di comunicazione da realizzare e per tale motivazione si è scelto di implementare in questo progetto un CAN bus come mezzo di comunicazione e condivisione dei dati. Il protocollo CAN e i fondamenti teorici che ne stanno alla base sono già stati illustrati ed approfonditi nel capitolo 4; si procede a trattare allora la progettazione e realizzazione della circuiteria hardware costituente il CAN bus.

Affinché ogni kit CY8CKIT-059 operante da Slave possa comunicare e condividere messaggi sul bus, è necessario aggiungere al circuito un transceiver CAN isolato. Il componente CAN messo a disposizione dalla scheda infatti non rende possibile interfacciarla direttamente al CAN bus dato che non implementa la codifica dei dati secondo il protocollo CAN; il transceiver agisce quindi da ponte di comunicazione tra ogni nodo della rete e il CAN bus stesso, permettendo ai vari kit Slave e a quello Master di scambiare messaggi utilizzando quello stesso linguaggio di codifica dei frame previsto dal protocollo CAN.

Un altro componente indispensabile è un regolatore di tensione: ogni batteria al piombo acido disponibile in laboratorio è caratterizzata infatti da una tensione nominale di 12V e tensione massima di 14.9V, mentre il kit CY8CKIT-059 può operare in un range di tensione V_{DD} di alimentazione tra i 3V e i 5.5V. Dato che deve essere la batteria stessa ad alimentare la scheda, il regolatore di tensione interviene adattando la tensione in uscita dalla prima e fissandola a 5V costanti, garantendo di conseguenza la corretta alimentazione del kit. Si deve tenere in considerazione che la comunicazione tra i vari nodi della rete deve essere isolata, poiché essi sono connessi e alimentati da batterie diverse e quindi operano a tensioni differenti; è dunque necessario utilizzare anche un convertitore DC-DC isolato e regolato per fornire in uscita 5V. In questo modo è possibile alimentare il kit e il lato non isolato del transceiver (quello che si affaccia al kit CY8CKIT-059 cui è collegato) tramite il regolatore di tensione ed in parallelo alimentare il lato isolato (quello che si affaccia al CAN bus) tramite l'uscita del convertitore DC-DC, garantendo l'isolamento della comunicazione tra i diversi nodi della rete.

Nel seguito di questo capitolo sono maggiormente approfonditi i componenti scelti per svolgere le funzioni sopra elencate e costituenti una parte fondamentale della rete di comunicazione progettata e realizzata.

8.2.1. Transceiver CAN isolato ISO1050DWR

Per quanto riguarda il transceiver CAN isolato la scelta è ricaduta sul componente ISO1050DWR di Texas Instruments, raffigurato in figura 8.2.1.1.

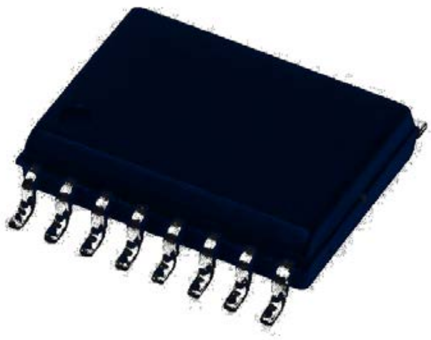


Figura 8.2.1.1: componente ISO1050DWR di Texas Instrument

Il componente ISO1050DWR è un transceiver CAN digitalmente isolato che soddisfa i requisiti dello standard ISO1198-2; la separazione tra buffer di input e output è realizzata mediante una barriera di ossido di silicio (SiO_2) che fornisce isolamento galvanico fino a 5000V_{RMS} [12]. Se utilizzato in combinazione con un’alimentazione isolata, il dispositivo previene l’ingresso di correnti di disturbo sul bus dati o su altri componenti che possano interferire o danneggiare circuiti sensibili: essendo un transceiver CAN, questo componente fornisce la capacità di instaurare una trasmissione differenziale sul CAN bus e una ricezione differenziale verso il controllore CAN con velocità fino a 1Mbps (megabit per second). Più nello specifico il componente utilizzato nel progetto è realizzato in formato SOIC (Small Outline Integrated Circuit) ovvero è un circuito integrato di piccole dimensioni (10.30 mm × 7.50 mm) con 16 pin, 8 per lato, come si può vedere in figura 8.2.1.2.

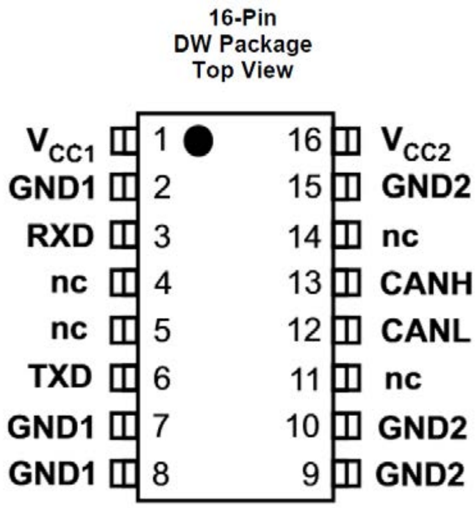


Figura 8.2.1.2: pin del componente ISO1050 in formato SOIC

I pin da 1 a 8 nel lato sinistro della figura 8.2.1.2 costituiscono il lato non isolato del transceiver, ovvero quello che si affaccia al kit CY8CKIT-059 (lato digitale): il pin V_{CC1} va connesso con la tensione di alimentazione fornita dal kit stesso (nel range 3V-5.5V), i pin GND1 costituiscono la connessione di terra lato digitale, il pin RXD rappresenta i dati ricevuti dal CAN bus (tensione low per stato dominante del bus, high per stato recessivo), il pin TXD rappresenta l’input dei dati trasmessi verso il CAN bus (la codifica dei bit per questo pin è uguale a quello RXD) mentre i pin NC (No Connect) non sono connessi. I pin da 9 a 16 nel lato destro della figura 8.2.1.2 invece costituiscono il lato transceiver, ovvero quello che si affaccia al CAN bus: il pin V_{CC2} va connesso con la tensione di alimentazione isolata pari a 5V fornita dal convertitore isolato DC-DC regolato a 5V, i pin GND2 costituiscono la connessione di terra lato transceiver, il pin CANH rappresenta la linea high-level del mezzo fisico del CAN bus, il pin CANL rappresenta la linea low-level del mezzo fisico del CAN bus mentre i pin NC non sono connessi.

Tra le peculiarità di questo componente c'è il DTO (TXD Dominant Time-Out), un circuito interno che impedisce al nodo connesso al transceiver ISO1050DWR di bloccare la comunicazione sul CAN bus qualora per un guasto hardware o software il pin TXD rimanga dominante per un tempo superiore al periodo di time-out t_{TXD_DTO}. In particolare, tale funzione di protezione disabilita il driver del CAN bus se non nota in esso un gradino di salita prima che il t_{TXD_DTO} finisca, garantendo la continuità di comunicazione tra gli altri nodi; il driver del nodo in questione viene riattivato solamente nel momento in cui un segnale recessivo viene percepito nel pin TXD. Questo concetto è rappresentato schematicamente in figura 8.2.1.3.

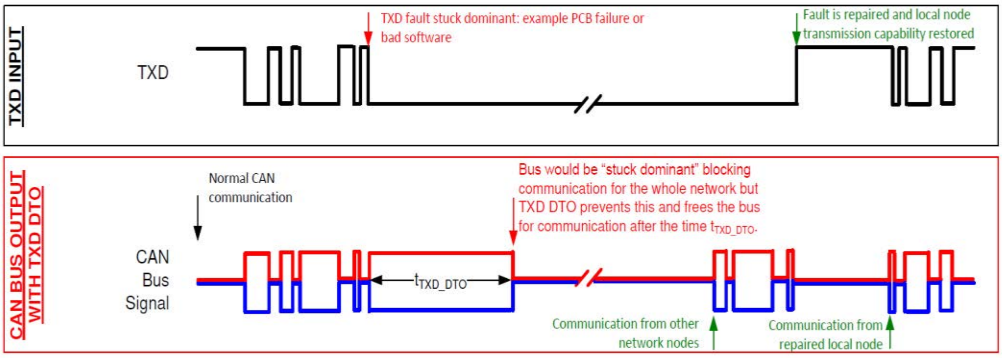


Figura 8.2.1.3: diagramma temporale del funzionamento del dispositivo durante DTO

Si deve però prestare attenzione alla presenza di un effetto collaterale dovuto al DTO: limitando il tempo per cui il pin TXD può rimanere allo stato dominante si va ad aumentare parallelamente la minima velocità di trasmissione dei dati possibile dell'apparecchio. Il protocollo CAN consente infatti un massimo di undici bit 0 dominanti successivi sul pin TXD nel caso peggiore, dove cinque bit dominanti successivi sono seguiti immediatamente da un frame di errore. Questa condizione, associata al valore minimo del periodo temporale t_{TXD_DTO}, impone una limitazione della velocità minima di trasmissione dei dati, che può essere calcolata sfruttando la formula $11/t_{TXD\_DTO\_minimo}$. Tenendo in considerazione che secondo il datasheet del componente ISO1050DWR il periodo $t_{TXD\_DTO\_minimo}$ vale 300µs, si ottiene che la minima velocità di trasmissione di dati sul CAN bus di cui tenere conto in fase di progettazione deve essere pari a 37kbps.

Un altro accorgimento cui prestare molta attenzione riguarda sia i tre pin GND1 lato digitale che i tre pin GND2 lato transceiver; essi infatti non sono internamente connessi tra loro come si potrebbe pensare (GND1-GND1-GND1, GND2-GND2-GND2). Per ottenere quindi un corretto funzionamento del componente ISO1050DWR si deve ricordare di cortocircuitare tra loro i pin di terra appartenenti ad uno stesso lato del dispositivo esternamente mediante cavo e/o saldatura.

8.2.2. Regolatore di tensione LM7805CT/NOPB

Per quanto riguarda il regolatore di tensione la scelta è ricaduta sul componente LM7805CT/NOPB di Texas Instruments, raffigurato in figura 8.2.2.1.

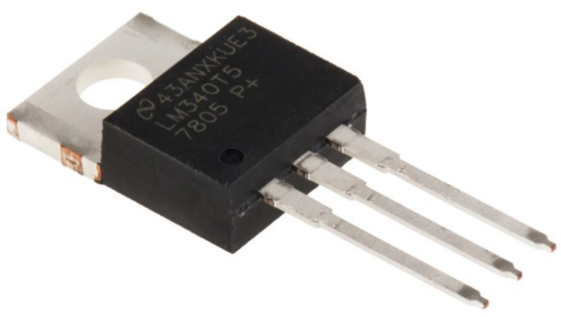


Figura 8.2.2.1: componente LM7805CT/NOPB di Texas Instrument

Il componente LM7805CT/NOPB è un regolatore di tensione monolitico a tre terminali, con tensione di uscita costante e pari a 5V [13]. È solitamente utilizzato come regolatore di tensione in un'ampia gamma di applicazioni, tra cui regolazione locale per l'eliminazione di disturbi e problemi di distribuzione associati alla regolazione single-point; può essere applicato inoltre come componente esterno per ottenere un output modificabile di tensione e corrente, ruolo che ricopre nel progetto illustrato in questa trattazione, nonostante questo non rappresenti il suo utilizzo più comune. Più nello specifico il componente utilizzato nel progetto è realizzato in formato TO-220, ovvero un particolare stile di package elettronico (dimensioni nominali del corpo principale 14.986 mm × 10.16 mm) con 3 pin come si può vedere in figura 8.2.2.1 e 8.2.2.2.

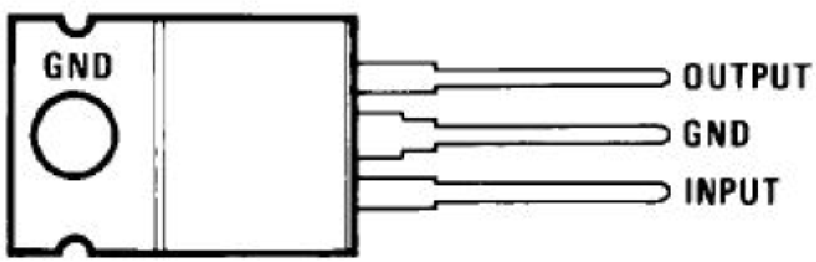


Figura 8.2.2.2: schema circuitale del componente LM7805CT/NOPB in formato TO-220

Il primo pin di figura 8.2.2.2 dal basso è quello di input, cui va connessa la tensione d'ingresso da regolare; il secondo pin, insieme alla aletta del corpo principale del componente, rappresenta la connessione di terra mentre il terzo pin è la tensione d'uscita. L'output può essere regolato, a seconda del componente scelto tra quelli appartenenti alla famiglia LM340 e LM7805, per assumere valori dai 3V fino ai 5V, accettando in ingresso valori di tensione fino a 35V e fornendo una corrente massima in output di 1.5A qualora gli si fornisca un'adeguata dissipazione del calore.

L'unica accortezza che si deve prestare quando si utilizza il componente LM7805CT/NOPB riguarda il range della tensione in input: per mantenere efficacie la regolazione della tensione di linea infatti il dispositivo deve essere alimentato con V_{IN} pari ad almeno 7.5V altrimenti non opera correttamente e non garantisce di fornire la tensione di output al valore stabilito dalla sua famiglia si appartenenza e per cui viene utilizzato.

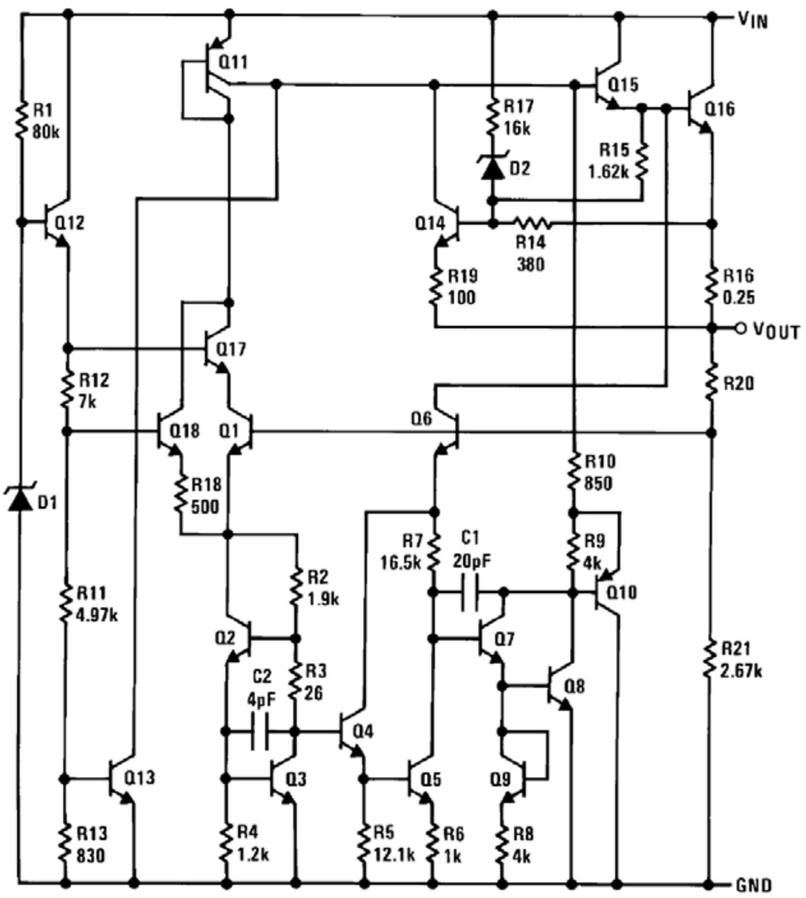


Figura 8.2.2.3: schema a blocchi funzionali del componente LM7805CT/NOPB

In figura 8.2.2.3 si può vedere lo schema a blocchi funzionali del componente LM7805CT/NOPB: non è comunque d'interesse per questa trattazione approfondirne nello specifico il principio e il circuito di funzionamento. Si lascia quindi al lettore un eventuale approfondimento del componente andandone a consultare con maggior attenzione il datasheet.

8.2.3. Convertitore isolato DC-DC regolato a single output SPA01A-05

Per quanto riguarda il convertitore isolato DC-DC regolato la scelta è ricaduta sul componente SPA01A-05 di Mean Well, raffigurato in figura 8.2.3.1.

Figura 8.2.3.1: package outline del componente SPA01A-05

Il componente SPA01A-05 è un convertitore isolato DC-DC regolato a 4 pin che fornisce isolamento tra le tensioni input e output fino a 1500V_{DC}, filtro built-in contro le emissioni elettromagnetiche e raffreddamento ad aria per convenzione naturale [14]. La tensione di output +V_{OUT} è impostata a 5V fissi con corrente di output pari a 0.2A. Più nello specifico il componente utilizzato nel progetto è realizzato in formato SIP (System In a Package), con i quattro terminali disposti in un'unica linea e accoppiati a due a due per poter distinguere la tensione d'ingresso da quella d'uscita. Le misure nominali del componente sono 7.75mm × 17.4mm: la casa costruttrice Mean Well non ne fornisce alcuno schema circuitale o a blocchi funzionali e comunque non sarebbero stati approfonditi nella presente trattazione in quanto non rilevanti per lo scopo ultimo del progetto.

L'unica accortezza che si deve prestare quando si utilizza il componente SPA01A-05 riguarda la tensione di input con cui alimentarlo, così come già visto in precedenza per il componente LM7805CT/NOPB: secondo il relativo datasheet infatti affinché ne sia garantito un corretto funzionamento la tensione di alimentazione deve essere compresa tra i 9V e i 18V.

8.2.4. Circuito hardware complessivo

Introdotti e approfonditi dunque i componenti scelti ed utilizzati per la realizzazione della sezione in esame, si può procedere illustrando lo schema circuitale hardware necessario per far interfacciare ogni singolo CY8CKIT-059 operante da Slave, e quindi responsabile dell'acquisizione dei segnali analogici di tensione della batteria monitorata da esso, al CAN bus cui deve comunicare i dati in suo possesso. In altre parole, si studieranno con maggiore attenzione i collegamenti fisici da realizzare per costituire quello che sarà di fatto un nodo della rete complessiva di comunicazione.

In precedenza, nel corso del capitolo 8, è stato anticipato come sia necessario l'utilizzo di un regolatore di tensione LM7805CT/NOPB per alimentare il kit CY8CKIT-059 tramite la batteria monitorata da esso e un convertitore isolato DC-DC regolato per alimentare il lato isolato del transceiver ISO1050DWR sempre tramite la batteria. Dal punto di vista circuitale si può vedere una rappresentazione di questi collegamenti fra i diversi componenti in figura 8.2.4.1.

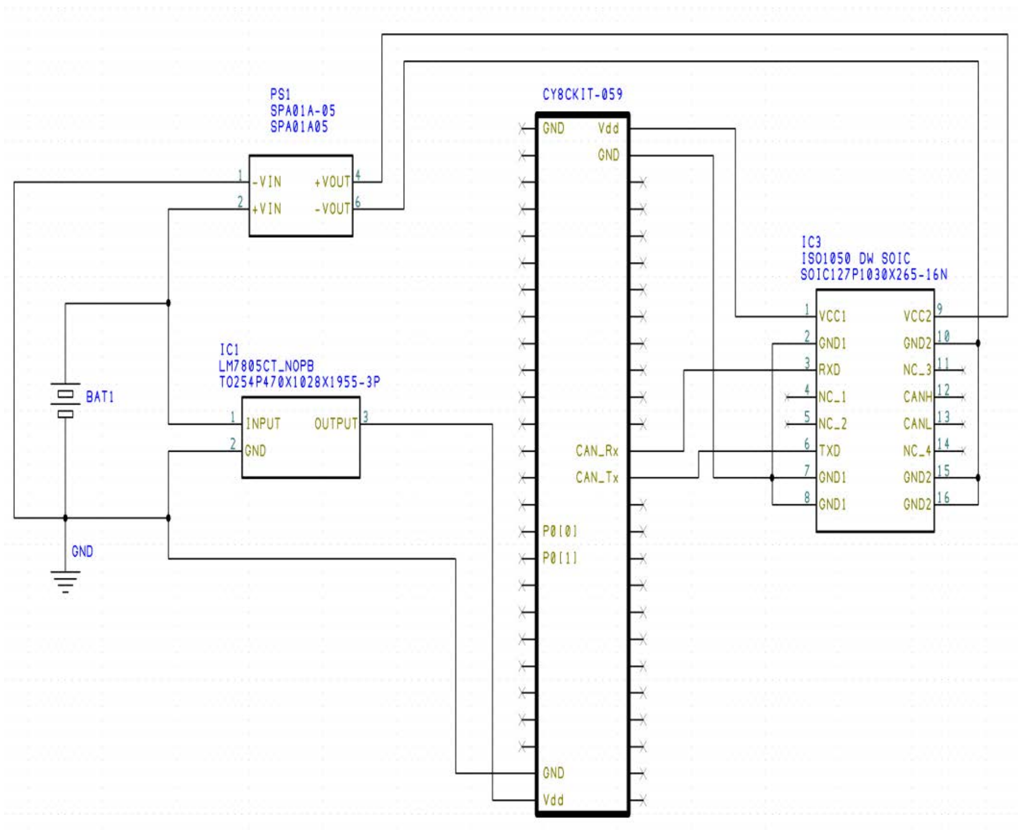


Figura 8.2.4.1: schema circuitale della sezione di trasmissione sul CAN bus dei dati di ogni nodo Slave della rete

I morsetti positivo e negativo della batteria sono collegati in primo luogo ai pin INPUT e GND rispettivamente del regolatore di tensione LM7805CT/NOPB e in secondo luogo ai pin $+V_{IN}$ e $-V_{IN}$ del convertitore isolato DC-DC, così da poter alimentare alla tensione corretta entrambi i componenti secondo le indicazioni fornite dai rispettivi datasheet. Il pin della tensione d'uscita regolata a 5V costanti del componente LM7805CT/NOPB, connesso al pin V_{DD} del kit CY8CKIT-059, funge da sorgente di potenza per la scheda. Per richiudere il circuito anche il pin GND del kit CY8CKIT-059 va connesso alla connessione di terra comune alla batteria. I pin $+V_{OUT}$ e $-V_{OUT}$ in uscita dal convertitore isolato DC-DC regolato a 5V sono connessi ai pin V_{CC2} e GND2 rispettivamente del transceiver CAN isolato ISO1050DWR, alimentandone quindi il lato isolato rivolto verso il CAN bus. La scheda CY8CKIT-059 alimenta a sua volta il lato digitale del transceiver stesso, connettendo i pin V_{DD}

e GND del kit ai pin V_{CC1} e GND1 rispettivamente dell'ISO1050DWR. Da notare, come già specificato in precedenza nel capitolo 8.2.1, che i tre pin GND1 lato digitale del transceiver CAN isolato sono collegati tra loro esternamente poiché non connessi internamente al componente; la stessa operazione va effettuata per i tre pin GND2 del lato isolato. Questa accortezza, seppur possa apparire banale, è fondamentale e necessaria: lo stesso autore infatti ha riscontrato il non funzionamento del componente ISO1050DWR qualora anche solo uno di questi pin di terra non fosse connesso correttamente nel modo appena indicato. Infine si connettono i pin CAN_Rx e CAN_Tx del kit CY8CKIT-059, attribuibili a due qualsiasi pin I/O generici del componente tramite PSoC Creator come verrà mostrato nel seguito, ai pin RXD e TXD rispettivamente dell'ISO1050DWR affinché il kit possa trasmettere i dati al transceiver CAN isolato e quest'ultimo possa inviarli nel CAN bus una volta impostata la struttura del frame secondo le specifiche stabilite dal protocollo CAN. I pin NC_1, NC_2, NC_3 e NC_4 devono essere lasciati privi di qualsiasi connessione.

Una volta assemblata la sezione circuitale in esame secondo i collegamenti e le specifiche appena descritte, ogni kit CY8CKIT-059 incaricato di acquisire i valori di tensione in real time da una batteria è in grado di trasmettere e ricevere nel bus CAN a cui è connesso i dati a sua disposizione o inviati da altri nodi, garantendo l'isolamento tra sé stesso e gli altri dispositivi collegati al bus. In questo modo si realizza di fatto un nodo della rete di comunicazione stessa in fase di progettazione.

Il circuito appena illustrato è il medesimo per ogni nodo che si vuole implementare, semplificando quindi le operazioni di gestione e modifica del CAN bus qualora fosse necessario; tale sezione può essere vista quindi come una struttura modulare facilmente replicabile, così come lo era il circuito di acquisizione dei segnali di tensione e corrente della batteria.

Una volta realizzati i circuiti hardware relativi alle due sezioni appena descritte nei capitoli 8.1 e 8.2 si procede a progettare e realizzare l'interfaccia seriale di comunicazione tra il nodo Master del CAN bus e il PC, rendendo così possibile la visualizzazione dei parametri acquisiti dal pacco batterie in due grafici real time mediante un opportuno script Matlab che verrà illustrato nel seguito della trattazione.

8.3. Interfaccia seriale tra il kit CY8CKIT-059 operante da Master del CAN bus e il PC

L'ultima sezione dello schema circuitale del progetto realizzato implementa la raccolta dei dati inviati sul CAN bus dai diversi nodi della rete sotto forma di frame CAN e la loro successiva trasmissione attraverso interfaccia seriale al PC così che tensione e corrente dell'intero pacco di batterie possano essere visualizzate in real time. I componenti utilizzati per questa struttura hardware sono un kit CY8CKIT-059 operante da Master del sistema di monitoraggio distribuito, un convertitore isolato DC-DC regolato SPA01A-05, un transceiver CAN ISO1050DWR e una prolunga USB così da sfruttare il KitProg della scheda come ponte UART-USB e connetterlo al PC. Lo schema circuitale relativo a tale sezione è rappresentato in figura 8.3.1.

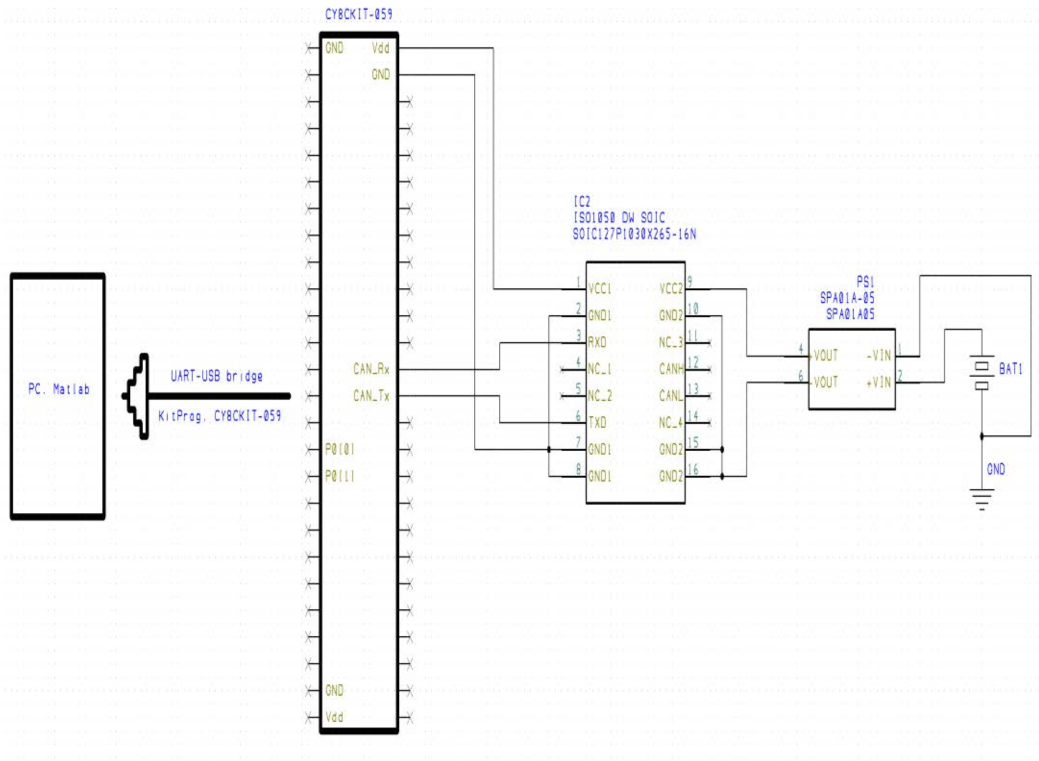


Figura 8.3.1: schema circuitale della sezione di interfaccia seriale UART-USB tra kit CY8CKIT-059 e PC

Più nel dettaglio il lato digitale del transceiver CAN è alimentato dal kit CY8CKIT-059 connettendo i pin V_{CC1} e GND1 del componente ISO1050DWR ai pin V_{DD} e GND rispettivamente della scheda; il lato isolato invece non può essere alimentato tramite il kit e il convertitore isolato DC-DC regolato SPA01A-05 poiché quest'ultimo per operare correttamente deve essere alimentato secondo le proprie specifiche con tensione di input nel range $9\text{V} < V_{IN} < 18\text{V}$ che la scheda non può mettere a disposizione (opera infatti a tensione V_{DD} massima di 5V). È quindi necessario che il lato bus del transceiver ISO1050DWR collegato al CY8CKIT-059 operante da Master sia alimentato direttamente da una qualsiasi delle batterie monitorate dal sistema di monitoraggio distribuito, interponendo ovviamente un convertitore isolato DC-DC regolato SPA01A-05 così da garantire un'alimentazione isolata a 5V del lato bus del transceiver. Si connettono allora i morsetti positivo e negativo della batteria ai pin $+V_{IN}$ e $-V_{IN}$ del componente SPA10A-05 e i due pin di uscita $+V_{OUT}$ e $-V_{OUT}$ del convertitore isolato DC-DC regolato a 5V ai pin V_{CC2} e GND2 del componente ISO1050DWR rispettivamente. Inoltre, si devono connettere i pin RXD e TXD del transceiver CAN isolato lato digitale ai pin CAN_Rx e CAN_Tx (assegnati come già introdotto in precedenza) rispettivamente del kit CY8CKIT-059 operante da Master affinché possa comunicare con il CAN bus e acquisire i dati dai vari nodi connessi alla rete di comunicazione. L'ultimo step riguarda l'interfaccia seriale vera e propria: il kit CY8CKIT-059 mette a disposizione nel KitProg evidenziato in figura 8.3.2 un connettore PCB USB (Printed Circuit Board Universal Serial Bus).

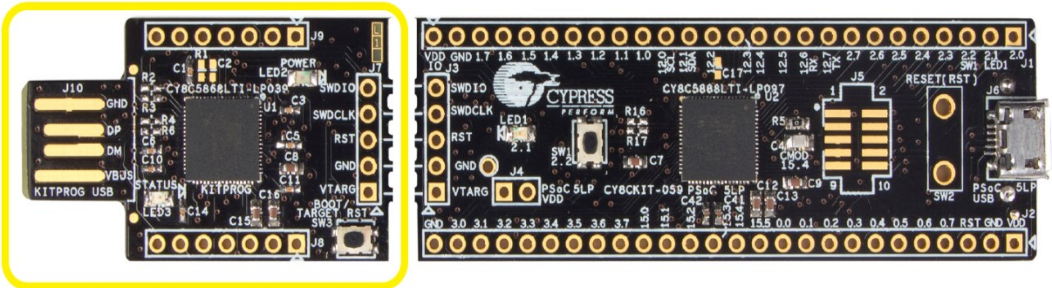


Figura 8.3.2: il KitProg, manualmente dal Target board, è evidenziato in giallo sulla sinistra

Il KitProg stesso può agire infatti da ponte USB-UART: le linee UART tra KitProg e Target board sono realizzate a livello di hardware nella zona distaccabile tra le due parti, tenendo in considerazione che le linee UART_RX e UART_TX sono assegnate rispettivamente ai pin P12_6 e P12_7 del dispositivo Target e tali assegnazioni non possono essere modificate. In questo modo il KitProg permette di inviare e ricevere dati tra il dispositivo Target e il PC, commutando la comunicazione UART implementata dalla scheda secondo le indicazioni del protocollo USB e scambiando informazioni mediante il mezzo fisico costituito dal connettore USB. Per quanto riguarda allora il kit CY8CKIT-059 operante da Master del CAN bus si deve ricordare di non distaccare le due parti che lo compongono se si vuole comunicare con il PC utilizzando il connettore USB presente nel KitProg; nel caso in cui tuttavia si dovesse verificare la separazione accidentale o volontaria tra KitProg e dispositivo Target, si può comunque trasmettere e ricevere dati dal computer sfruttando il connettore USB Type-B Mini messo a disposizione dal Target board ed evidenziato in figura 8.3.3.

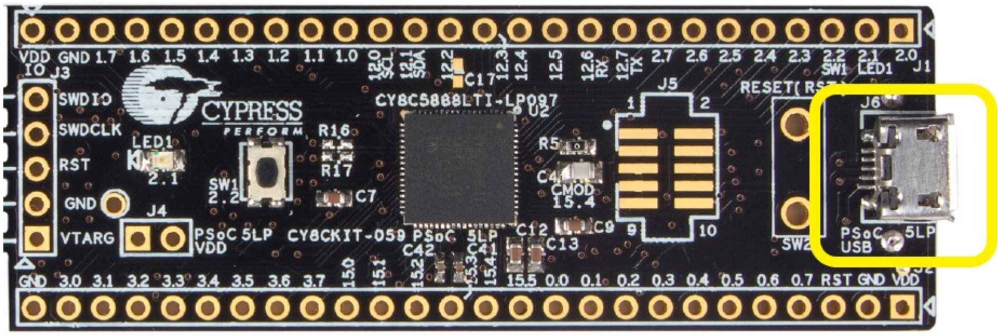


Figura 8.3.3: connettore USB Type-B Mini del Target board

Nel progetto in esame si è scelto di utilizzare il connettore USB Type-A messo a disposizione dal KitProg del CY8CKIT-059 per poter sfruttare i cavi USB a disposizione nel laboratorio.

8.4. Circuito hardware complessivo

Illustrate ed analizzate in dettaglio le tre sezioni in cui concettualmente è stato diviso il progetto per la realizzazione della rete di comunicazione di un sistema di monitoraggio distribuito della carica e scarica di una batteria, si procede ora a visualizzarne lo schema circuitale hardware nella sua interezza in figura 8.4.1. Si deve tenere in considerazione che i circuiti hardware presentati per le tre sezioni distinte servivano in primo luogo per mostrare i componenti necessari per realizzarli e studiarne a fondo i criteri di funzionamento, in secondo luogo per progettare step by step un circuito molto complesso nella sua struttura finale, consentendo all’autore di essere meno soggetto ad effettuare errori sia per quanto riguarda il corretto utilizzo dei componenti che per quanto concerne le numerose connessioni da realizzare per collegare tra loro i componenti stessi.

Quando si passa alla realizzazione del circuito hardware nella sua interezza, gli schemi circuitali delle tre sezioni singole devono essere leggermente modificati in quanto parti integranti di una macrostruttura e dunque vanno collegati tra loro nel modo più opportuno come evidenziato in figura 8.4.1. In particolare, in questo passaggio della progettazione del sistema si deve introdurre ed implementare nel circuito anche il mezzo trasmissivo fisico del CAN bus, composto secondo le indicazioni del protocollo CAN da una coppia di cavi ritorti, solitamente un doppino, per realizzare quello che è un singolo canale bidirezionale differenziale. Vanno aggiunte inoltre due resistenze da 120Ω, una ad ogni terminazione del CAN bus, per ottenere così una resistenza equivalente da 60Ω. Questa accortezza diventa sempre più importante all'aumentare della lunghezza dei cavi che costituiscono il mezzo fisico del bus stesso, in quanto realizzano sostanzialmente un adattamento d'impedenza necessario allo scambio di informazioni poiché evitano la riflessione del segnale sul CAN bus qualora l'impedenza caratteristica di linea differisca da quelle di ingresso dei ricevitori e del trasmettitore. Inoltre, la misura della resistenza ohmica complessiva del bus può essere utilizzata come verifica dell'integrità del bus. Qualora i due cavi CAN_L e CAN_H fossero infatti cortocircuitati fra loro, la resistenza misurata sarebbe 0Ω; se invece il circuito fosse interrotto si misurerebbero 120Ω. La tolleranza solitamente applicata su tali misure è ±5%.

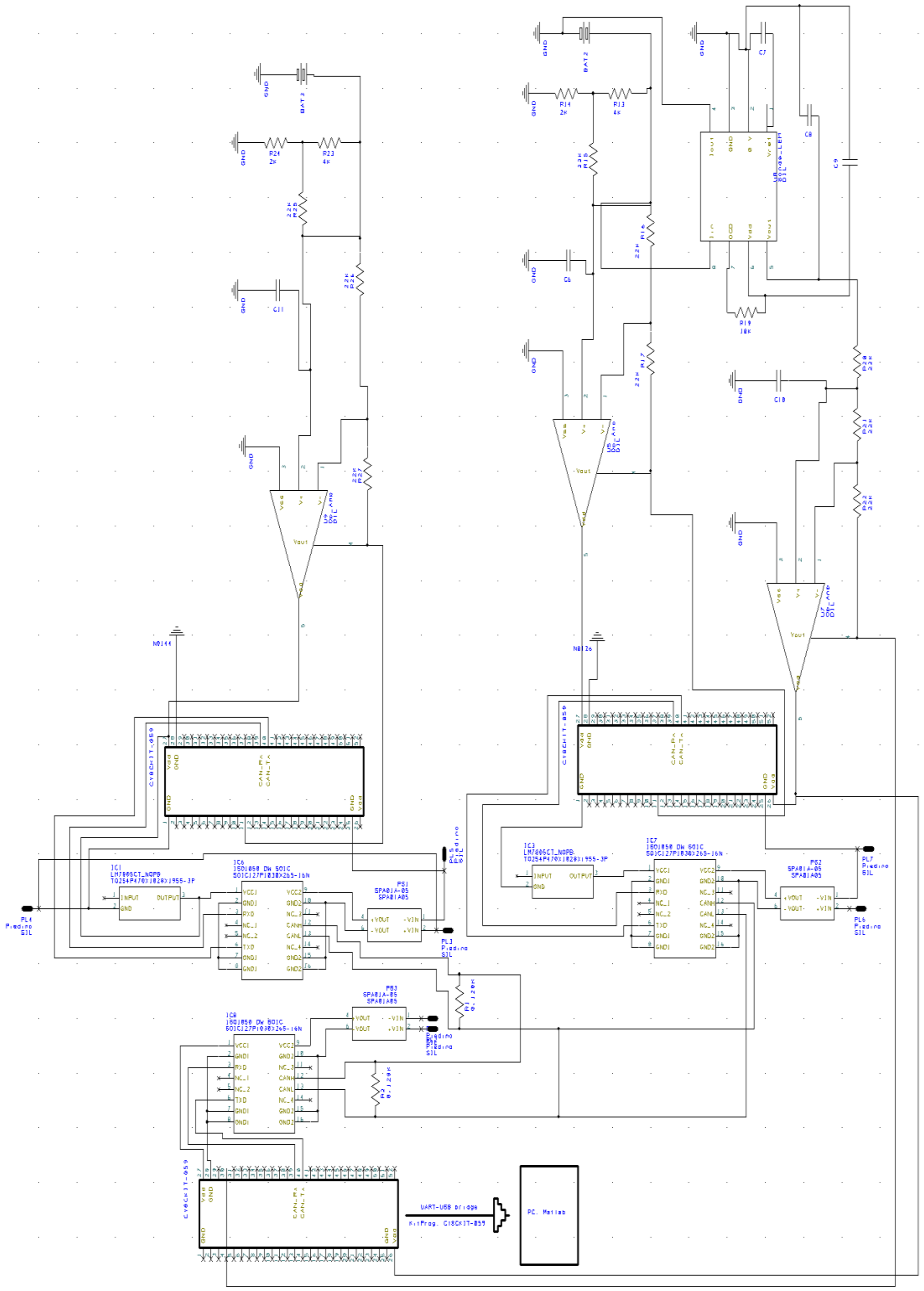


Figura 8.4.1: circuito hardware di una rete di comunicazione basata sul CAN bus di un sistema di monitoraggio distribuito della carica e scarica di un pacco batterie composto da due accumulatori

La sezione in alto di figura 8.4.1 rappresenta i due nodi della rete di comunicazione implementata operanti da Slave, ognuno dedicato al monitoraggio di una singola batteria e costituito da un kit CY8CKIT-059 e il relativo sistema di acquisizione dei segnali analogici di tensione illustrato nel capitolo 8.1. Nella parte centrale invece è raffigurato il cuore vero e proprio della rete di comunicazione, ovvero il mezzo fisico del CAN bus e l'interfaccia necessaria ai diversi nodi per poterci comunicare; in questa sezione si rende necessario l'utilizzo di transceiver CAN isolati ISO1050DWR, regolatori di tensione LM7805CT/NOPB e convertitori isolati DC-DC regolati a 5V SPA01A-05 come illustrato nel capitolo 8.2 e sotto capitoli. La parte inferiore della figura è costituita infine dal nodo Master della comunicazione, connesso al CAN bus mediante opportuna interfaccia; il kit, ricevuti i valori di tensione trasmessi da ogni nodo, acquisisce il segnale di corrente da una delle batterie monitorate e lo trasmette insieme a quelli di tensione ricevuti dai nodi Slave mediante UART (Universal Asynchronous Receiver/Transmitter) al PC sfruttando il ponte USB-UART e il connettore USB standard messi a disposizione dal KitProg del CY8CKIT-059 in questione come illustrato nel capitolo 8.3. La trasmissione seriale si conclude quindi nel PC: il programma scelto ed utilizzato per la visualizzazione grafica dei dati acquisiti in real time è Matlab, come verrà successivamente illustrato nei prossimi capitoli, che permette di ricevere e trasmettere dati mediante comunicazione seriale sfruttando la porta USB del PC e fornisce gli strumenti necessari per elaborare e visualizzare l'andamento dei parametri d'interesse.

Conclusa quindi la descrizione e l'approfondimento del circuito hardware progettato e realizzato, con relativa analisi approfondita delle sub-strutture che lo compongono e dei componenti elettronici impiegati, si può procedere con l'illustrazione della parte software, altrettanto fondamentale e necessaria per implementare una rete di comunicazione per un sistema di monitoraggio distribuito della carica e scarica di una batteria funzionante correttamente.

9. Sistema di monitoraggio distribuito basato sulla comunicazione CAN bus: progettazione e implementazione del software

Se il circuito hardware costituisce di fatto il cuore della rete di comunicazione, il software ne rappresenta in qualche modo il cervello; è infatti l'insieme di istruzioni e quindi programmi, oltre che delle componenti logico-digitali, che permettono di gestire e regolare le operazioni di un dispositivo. Lo stesso funzionamento del kit CY8CKIT-059 dipende interamente dallo script con cui viene programmato; è l'utente per mezzo dell'ambiente di lavoro PSoC Creator, in particolare nel file 'main.c' del progetto d'interesse, a fornire indicazioni alla scheda riguardanti le funzioni da svolgere e le azioni da eseguire. Per programmare il kit CY8CKIT-059 si deve semplicemente sviluppare e scrivere in modo corretto, a seconda dello scopo da ottenere, il codice scritto in linguaggio C e una volta collegato il dispositivo al PC mediante il connettore USB del KitProg cliccare il pulsante 'Program (Alt + F5)' nella barra in alto degli strumenti, come mostrato in figura 9.1.

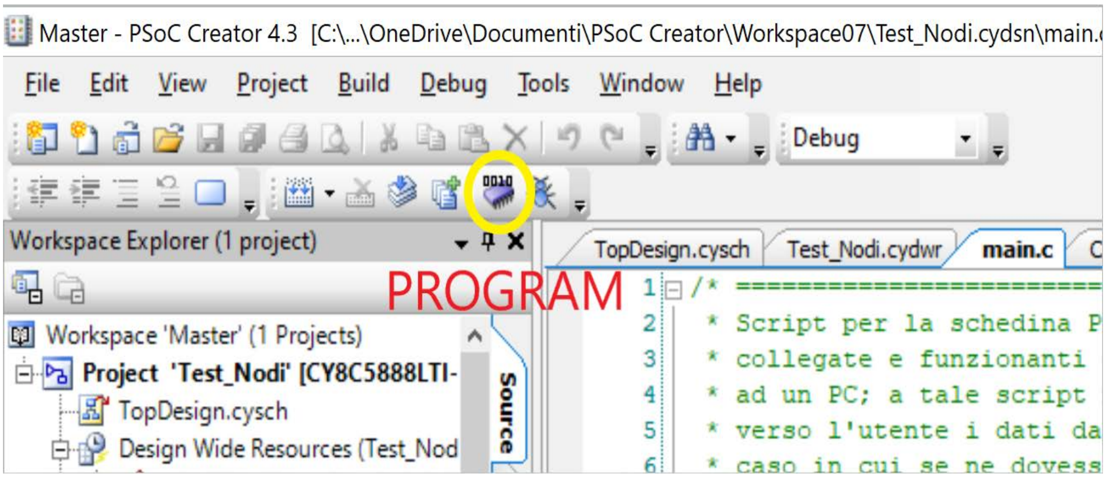


Figura 9.1: pulsante 'Program' in PSoC Creator

Come già illustrato in precedenza nel capitolo 3, oltre alla compilazione del file 'main.c' è necessario nella finestra 'TopDesign.cysch' inserire e configurare i componenti (periferiche) che il kit deve utilizzare per eseguire correttamente lo script secondo cui è stato programmato. Il workspace traduce automaticamente, quando si esegue la costruzione del progetto, il settaggio dei componenti e trasforma le connessioni del design sviluppato in collegamenti analogici e digitali internamente al kit CY8CKIT-059 stesso, oltre a generare i codici relativi ad ognuno di essi, esplorabili e modificabili dall'utente nella finestra di sinistra dell'ambiente di lavoro, come si può vedere in figura 9.2.

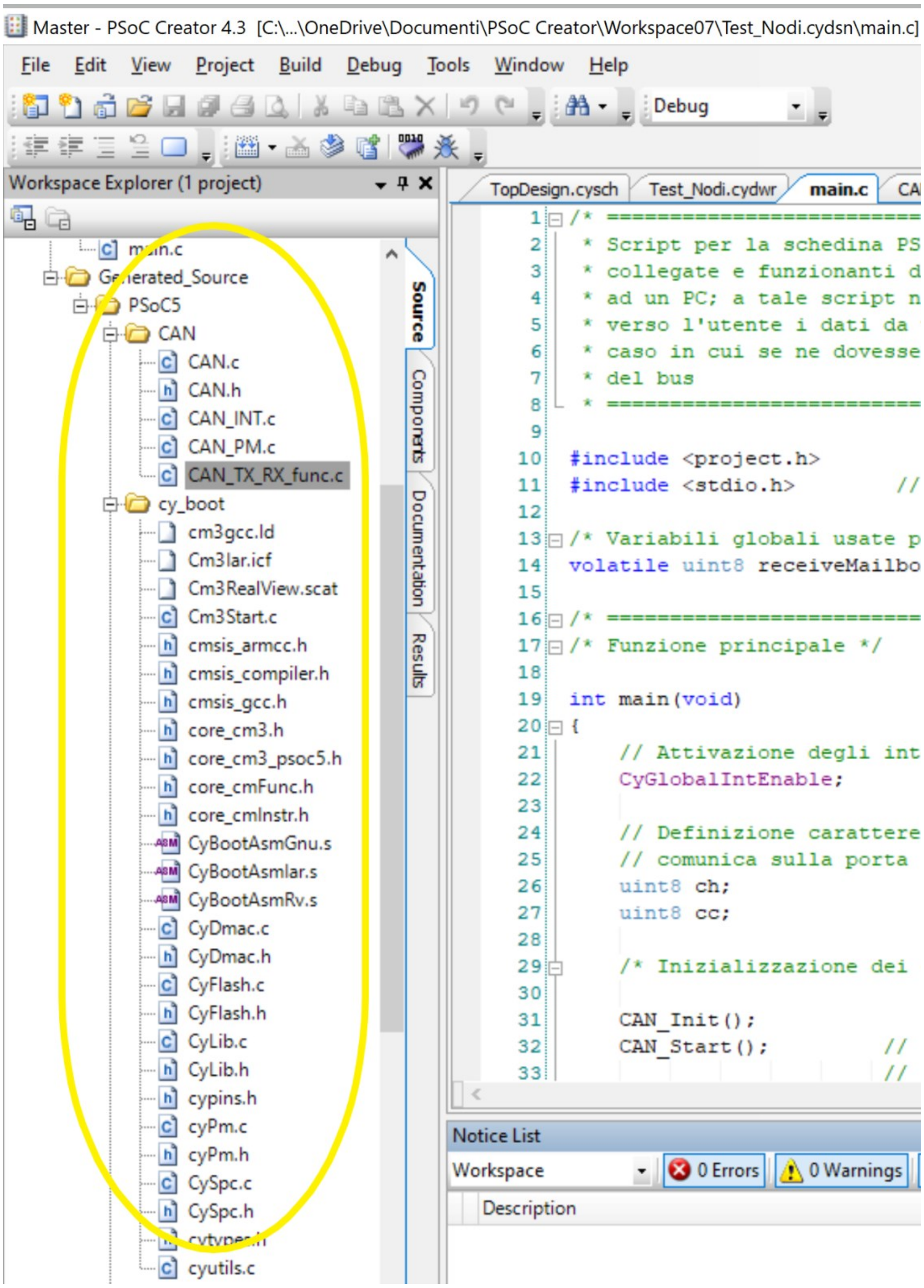


Figura 9.2: finestra che permette di accedere ai codici relativi ai diversi componenti utilizzati nel progetto, oltre che modificarli a seconda delle esigenze dell'utente

Le diverse possibilità di configurazione dei componenti CAN e UART utilizzati nel progetto in esame sono state introdotte nei capitoli 5.1 e 7.1 rispettivamente; per quanto riguarda il componente ADC_SAR_Seq, impiegato nel circuito di acquisizione dei segnali di tensione e corrente della batteria, si rimanda alla tesi “Studio di un sistema per il monitoraggio della carica degli accumulatori di un veicolo elettrico” di Samuel Matrella [1].

Si procede ora ad illustrare le configurazioni dei diversi componenti utilizzati, oltre che i codici scritti ad hoc, sia per quanto riguarda i kit CY8CKIT-059 operanti da Slave che per quello operante da Master del CAN bus. Inoltre, si illustra nel seguito anche lo script Matlab utilizzato per acquisire ed elaborare i dati dalla porta seriale (ponte USB-UART) e generare in grafici degli andamenti delle grandezze in real time.

9.1. Configurazione software dei kit CY8CKIT-059 operanti da Slave della rete di comunicazione

Il primo passo da compiere è riferirsi agli schemi circuitali rappresentati in figura 8.1.1 e 8.2.4.1 e cogliere il principio secondo cui sono stati ideati e realizzati. I componenti utilizzati per implementare un nodo Slave sono un convertitore analogico-digitale ADC_SAR_Seq e un driver CAN; il kit CY8CKIT-059 facente parte di tale nodo infatti deve convertire i segnali analogici di tensione acquisiti (capitolo 8.1) dalla batteria che monitora, per poi inviarli mediante un’opportuna interfaccia (descritta nel capitolo 8.2) sul CAN bus.

9.1.1. Configurazione del componente ADC_SAR_Seq

Il primo componente da configurare è il convertitore ADC_SAR_Seq, secondo le impostazioni mostrate in figura 9.1.1.1.

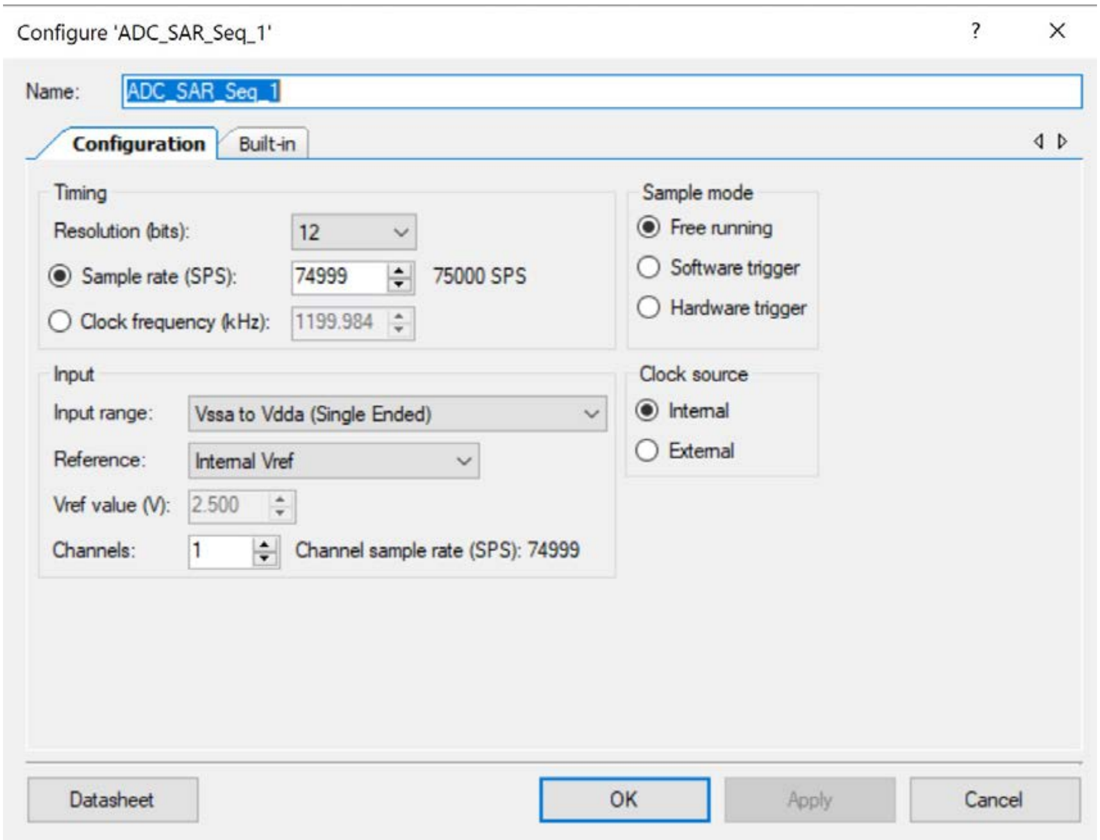


Figura 9.1.1.1: finestra di configurazione del componente ADC_SAR_Seq

Una volta applicate tali impostazioni il componente si presenta visivamente nel workspace di PSoC Creator come in figura 9.1.1.2.

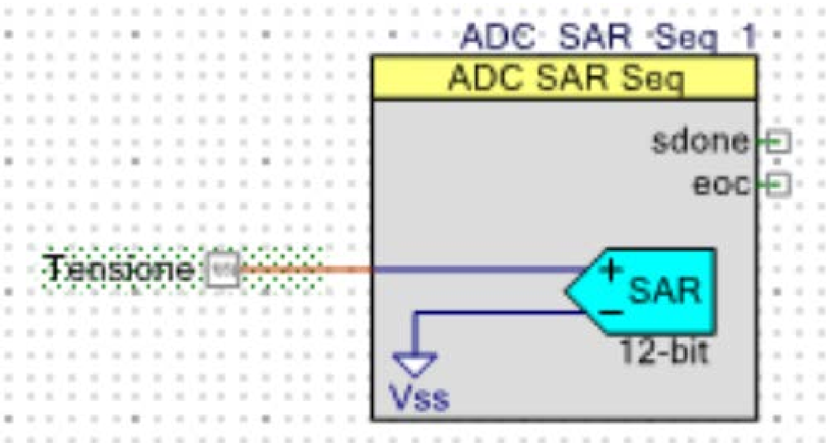


Figura 9.1.1.2: interfaccia grafica del componente ADC_SAR_Seq una volta configurato correttamente

Il canale di entrata corrisponde al pin I/O del kit CY8CKIT-059 selezionato per acquisire i segnali analogici di tensione in ingresso, provenienti dal partitore di tensione e filtro passa-basso rappresentati nel circuito illustrato nel capitolo 8.1. Tali segnali sono riferiti al pin V_{SS} di terra poiché il componente lavora in modalità Single Ended; inoltre la conversione analogico-digitale che applica è continuativa nel tempo e senza alcuna interruzione poiché impostata tramite interfaccia grafica in modalità Free Running.

Una volta convertiti in segnali digitali i valori analogici di tensione ricevuti in input, diverse Application Program Interface associate al convertitore ADC_SAR_Seq permettono all'utente di ottenere come risultato finale della conversione un segnale codificato nel sistema decimale. In particolare, risulta utile per tale scopo la funzione ADC_SAR_Seq_1_CountsTo_mVolts che verrà approfondita nel seguito della trattazione nell'Appendice A.

9.1.2. Configurazione del componente CAN

Il secondo componente da configurare è il componente CAN, secondo le impostazioni mostrate in figura 9.1.2.1 e 9.1.2.2.

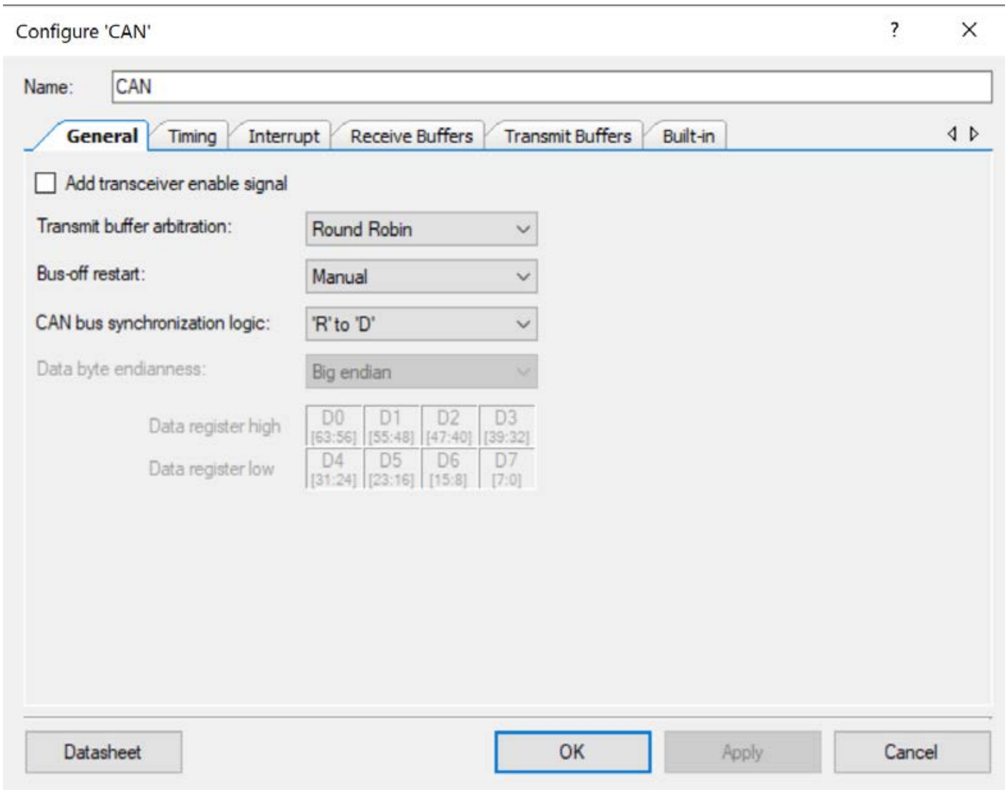


Figura 9.1.2.1: finestra di configurazione del componente CAN

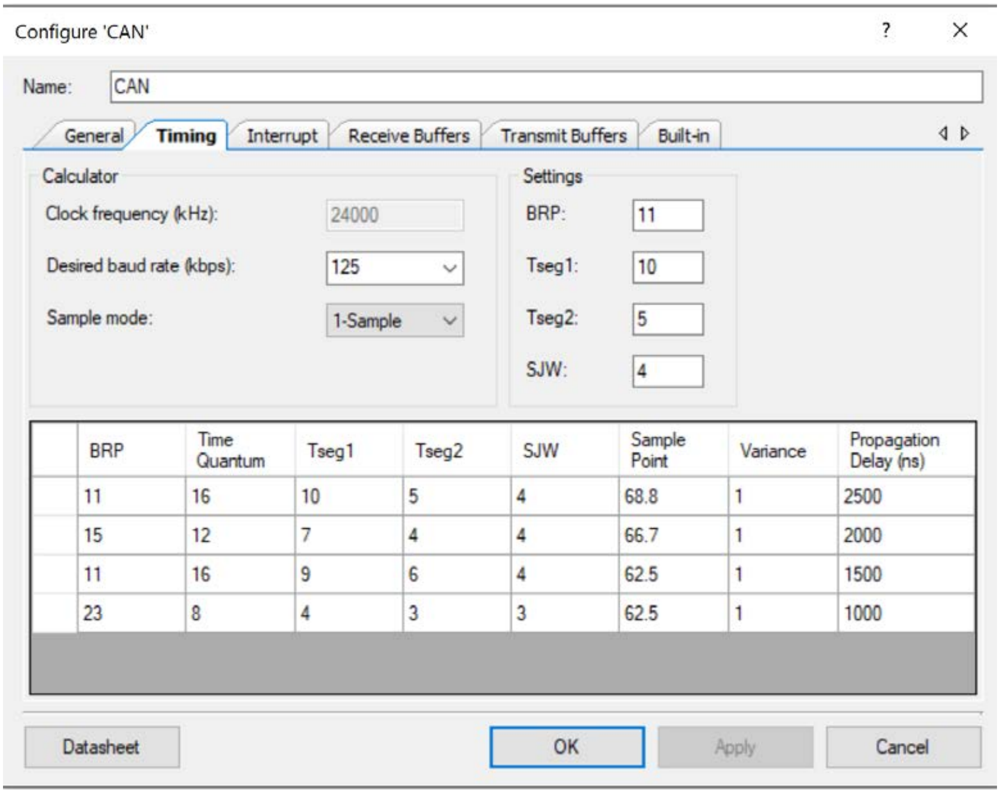


Figura 9.1.2.2: finestra di configurazione del timing del componente CAN

Come si può notare in figura 9.1.2.1 il componente CAN è configurato in modo tale che non sia presente un segnale esterno di abilitazione del transceiver CAN cui il componente è connesso, che l’ordine di invio dei messaggi delle mailbox di trasmissione sia 0-1-2 … 7-0-1, che la riattivazione del CAN bus qualora entri nello stato off debba essere eseguito manualmente dall’utente all’interno della main routine del kit CY8CKIT-059, che la logica di sincronizzazione del CAN bus sia ‘Recessive’ to ‘Dominant’, ovvero avvenga in corrispondenza di un fronte a gradino recessivo-dominante ed infine che l’ordine dei byte di dati dei frame sia Big Endian (impostazione di default). Inoltre, la velocità di comunicazione dei dati sul CAN bus, che deve essere unica e condivisa da tutti i nodi della rete affinché la comunicazione fra essi funzioni correttamente, è impostata a 125Kbps, essendo di fatto la massima velocità di comunicazione utilizzabile sfruttando il clock interno IMO del kit CY8CKIT-059 e rispettando parallelamente i requisiti di tolleranza per un CAN clock, pari all’1.58% per velocità fino a 125Kbps; qualora si voglia implementare una rete di comunicazione operante a velocità maggiori, ovvero comprese tra i 125Kbps e 1Mbps, si rende necessario l'utilizzo di un clock esterno per poter garantire il requisito di tolleranza del CAN clock pari allo 0.5% previsto per questo range. Una volta selezionata la velocità di comunicazione si impostano anche, cliccando su una delle righe della tabella in basso di figura 9.1.2.2, le lunghezze temporali dei diversi segmenti (BRP, Tseg1, Tseg2, SJW) proposte dal componente stesso a seconda della velocità di trasmissione scelta e che secondo le specifiche CAN costituiscono di fatto la temporizzazione del bit; l'impostazione manuale di valori non conformi a quelli presenti in tabella comporta un malfunzionamento del componente CAN e quindi l'impossibilità per questo nodo di comunicare con la rete.

Si configurano poi i buffer di ricezione e trasmissione, illustrati precedentemente nel capitolo 5.1; secondo la struttura implementata nel progetto in esame ogni nodo Slave della rete deve leggere sul CAN bus il messaggio proveniente dal nodo Master che coincide con la richiesta dei valori di tensione acquisiti dal nodo Slave stesso. In risposta quest'ultimo invia un frame dati sul CAN bus affinché il nodo Master possa acquisire i valori dei parametri di suo interesse; si devono allora configurare una mailbox di ricezione, impostata secondo le caratteristiche del frame inviato dal nodo Master (tipologia del messaggio e ID devono coincidere per tutti i nodi, altrimenti il messaggio non viene letto), e una di trasmissione caratterizzata da uno specifico ID (rappresentante il numero del nodo Slave che risponde alla richiesta del Master della comunicazione) e tipologia del messaggio. Nel progetto corrente si è deciso di utilizzare messaggi di tipologia Full CAN, di attribuire al messaggio del nodo Master l'ID 0x001 e ai messaggi dei nodi Slave 0x002 e 0x003. Le tabelle così configurate dei buffer di ricezione e trasmissione, nel caso specifico del nodo Slave 1, sono rappresentate in figura 9.1.2.3 e 9.1.2.4 rispettivamente.

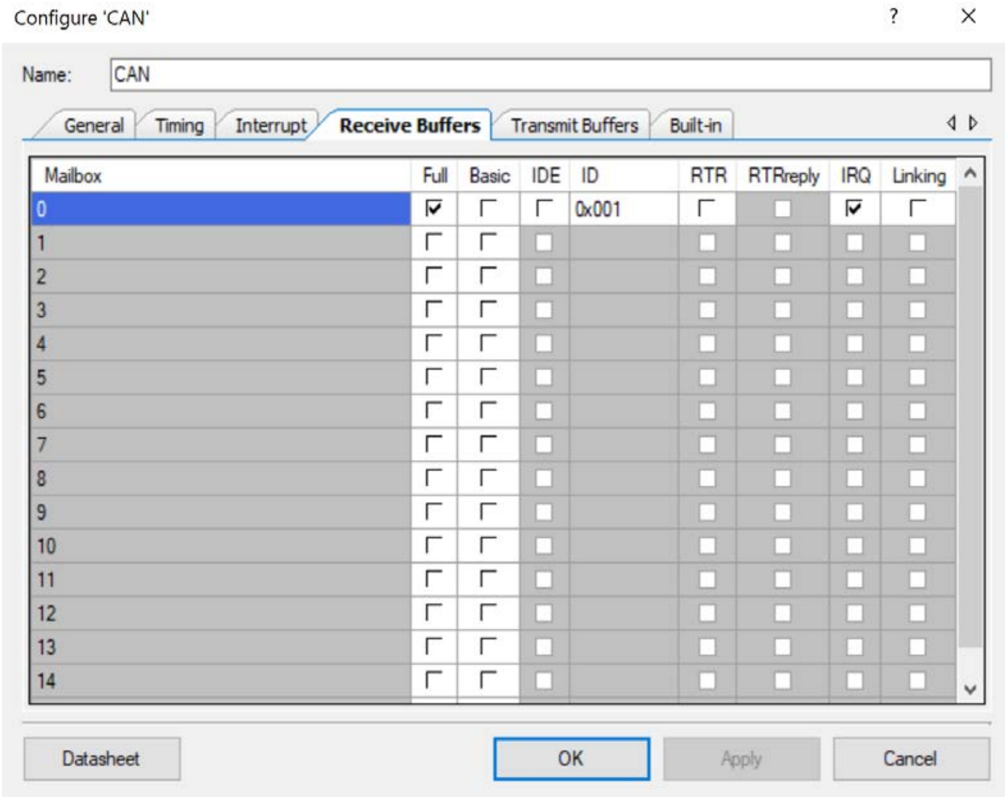


Figura 9.1.2.3: buffer di ricezione del componente CAN del nodo Slave 2 di rete

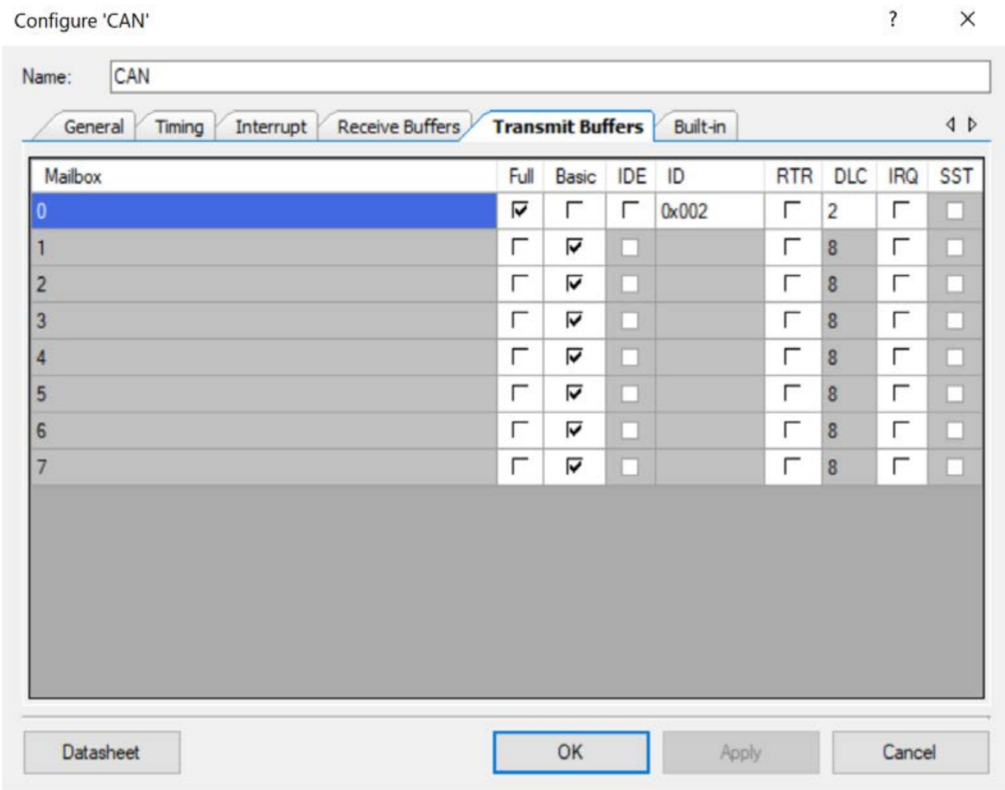


Figura 9.1.2.4: buffer di trasmissione del componente CAN del nodo Slave 2 di rete

In particolare il DLC (Data Lenght Code, numero di byte del data field del messaggio) è impostato a 2 byte poiché, come verrà illustrato in seguito, i valori di tensione acquisiti dalla batteria e restituiti dal componente ADC_SAR_Seq sono salvati dal kit CY8CKIT-059 come int16, ovvero interi salvati in 16 bit (2 byte) l'uno; ogni nodo Slave della rete invia nel CAN bus, quando richiestogli dal nodo Master (ovvero quando legge nel bus l'ID del messaggio di richiesta dei dati trasmesso dal nodo Master) un data frame contente 2 byte di dati, codificanti la tensione della rispettiva batteria che monitora.

9.1.3. Programmazione dei kit CY8CKIT-059 operanti da nodi Slave del CAN bus

Dopo aver configurato i componenti ADC_SAR_Seq e CAN del kit CY8CKIT-059 secondo le indicazioni descritte nel precedente capitolo, affinché la scheda Cypress operi da nodo Slave della rete occorre che il file 'TopDesign.cysch' del relativo progetto di PSoC Creator appaia come in figura 9.1.3.1.

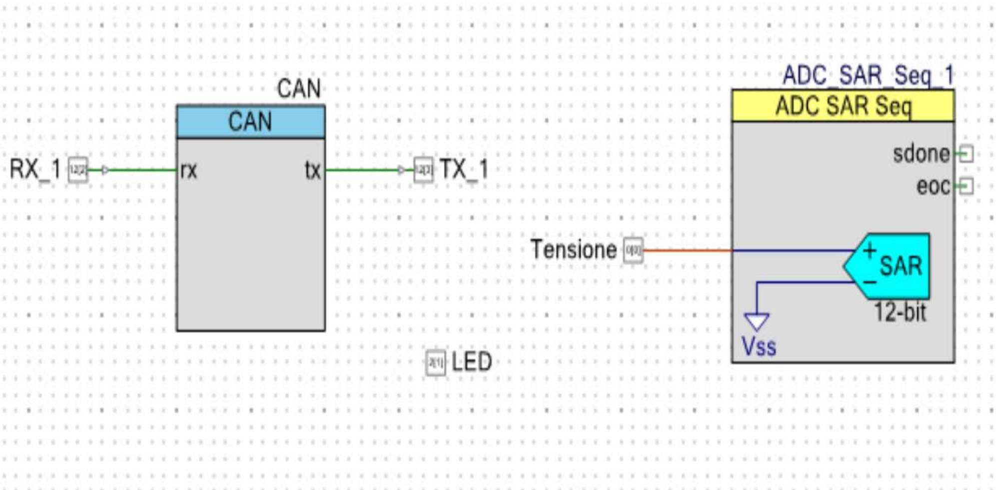


Figura 9.1.3.1: file 'TopDesign.cysch' di PSoC Creator per implementare un nodo Slave della rete di comunicazione del CAN bus

L'assegnazione dei pin digitali RX_1 e TX_1 del componente CAN, oltre che di quello analogico di tensione del componente ADC_SAR_Seq, può essere effettuata liberamente, l'unico vincolo è rispettare la tipologia di pin che il kit

CY8CKIT-059 mette a disposizione; si consiglia l'utilizzo dei pin I/O generici che possono essere utilizzati sia come digitali che analogici.

Il passo successivo è sviluppare il codice in linguaggio C per programmare il kit CY8CKIT-059 affinché svolga le operazioni richieste da un nodo Slave del CAN bus: nel progetto in esame questo si traduce nell'implementazione di una main routine che prevede l'acquisizione dei segnali analogici di tensione dalla batteria, mediante pin dedicati, la loro conversione in millivolt e la trasmissione nel bus in seguito alla ricezione della richiesta dei dati da parte del nodo Master mediante un frame CAN caratterizzato da un data field di 2 byte. L'invio dei dati sul bus avviene ad intervalli periodici programmabili a discrezione del progettista; nel contesto particolare del monitoraggio della carica e scarica di una batteria non è necessario acquisire i dati centinaia o migliaia di volte al secondo. La tensione dell'accumulatore infatti cambia lentamente ed è caratterizzata da variazioni apprezzabili solo nell'arco temporale di diversi minuti durante i processi di carica e scarica. Questa considerazione si traduce nell'inserimento di un delay (mediante opportuno comando 'CyDelay(ritardo_in_millisecondi)' nel file 'main.c') cosicché il kit CY8CKIT-059 ricominci ad eseguire la routine per cui è stato programmato solo dopo aver atteso il periodo temporale di ritardo stabilito. L'intero procedimento appena descritto deve essere tradotto in linguaggio C ed essere inserito, rispettando il lessico e le funzioni API messe a disposizione per i dispositivi Cypress, nel file 'main.c' del progetto in PSoC Creator.

Prima di mostrare e descrivere il codice scritto ed utilizzato per programmare un kit CY8CKIT-059 affinché operi da Slave, si procede ad analizzare la configurazione software del kit CY8CKIT-059 operante da Master del CAN bus e dialogante mediante porta seriale con il PC per la visualizzazione dei dati ottenuti in real time.

9.2. Configurazione software del kit CY8CKIT-059 operante da Master della rete di comunicazione

Similmente ad un nodo del CAN bus operante da Slave, il kit CY8CKIT-059 da programmare come Master della comunicazione necessita l'utilizzo del componente ADC_SAR_Seq poiché deve acquisire direttamente il segnale analogico di corrente da una batteria; la corrente che scorre in più batterie poste in serie fra loro è unica ed uguale per ognuna di esse e può quindi essere acquisita e monitorata direttamente dal Master della comunicazione. Si deve poi occupare della raccolta dell'insieme dei valori di tensione acquisiti dai nodi Slave e trasmessi sul CAN bus per poi inviarli al PC mediante porta seriale. Anche in questo caso allora il kit deve utilizzare il componente CAN per potersi interfacciare con il CAN bus; si rende necessario poi aggiungere il componente UART (Universal Asynchronous Receiver/Transmitter) cosicché possa avvenire lo scambio di dati tra scheda e PC mediante porta seriale. Si approfondisce nel seguito la configurazione di questi due componenti necessari per la corretta programmazione e funzionamento del kit CY8CKIT-059 operante da Master, mentre non viene approfondita la configurazione del componente ADC_SAR_Seq già indicata in precedenza per i nodi Slave nel sotto-capitolo 9.1.1.

9.2.1. Configurazione del componente CAN

Il componente CAN va configurato in modo analogo a quanto visto nel capitolo 9.1.2 per quanto riguarda parametri generali e bit timing (riferirsi dunque alla figura 9.1.2.1 e 9.1.2.2.), con l'unica differenza associata alle diverse impostazioni specifiche dei buffer di ricezione e trasmissione che si possono vedere in figura 9.2.1.1 e 9.2.1.2 rispettivamente.

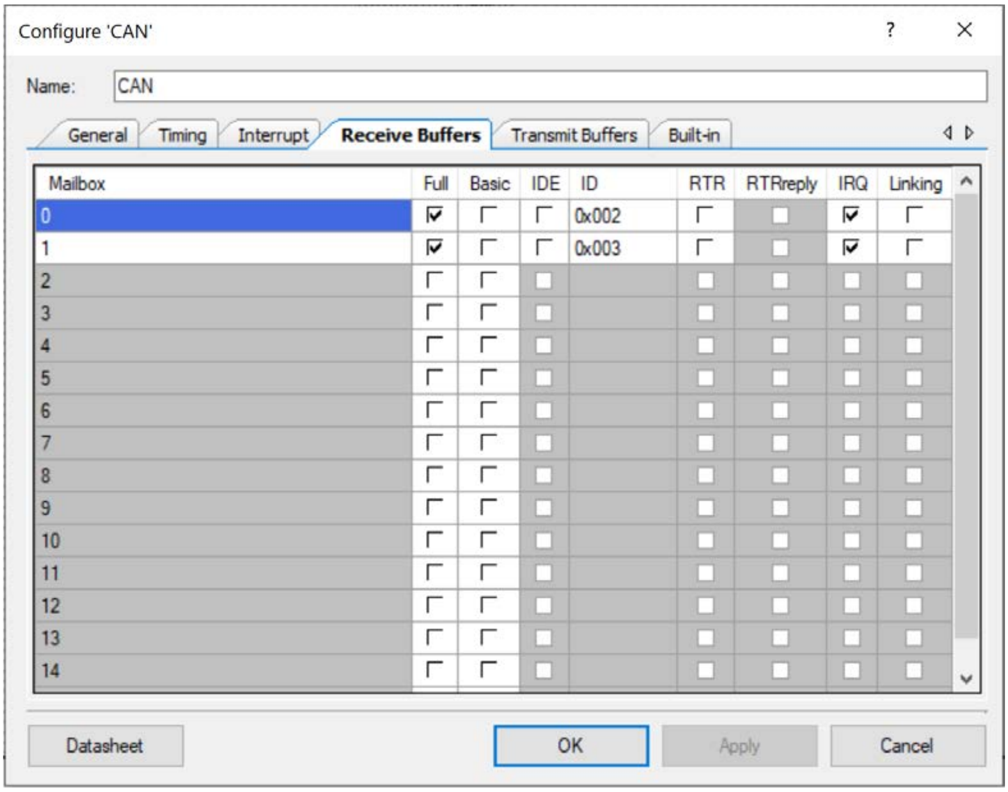


Figura 9.2.1.1: buffer di ricezione del componente CAN del nodo Master di rete

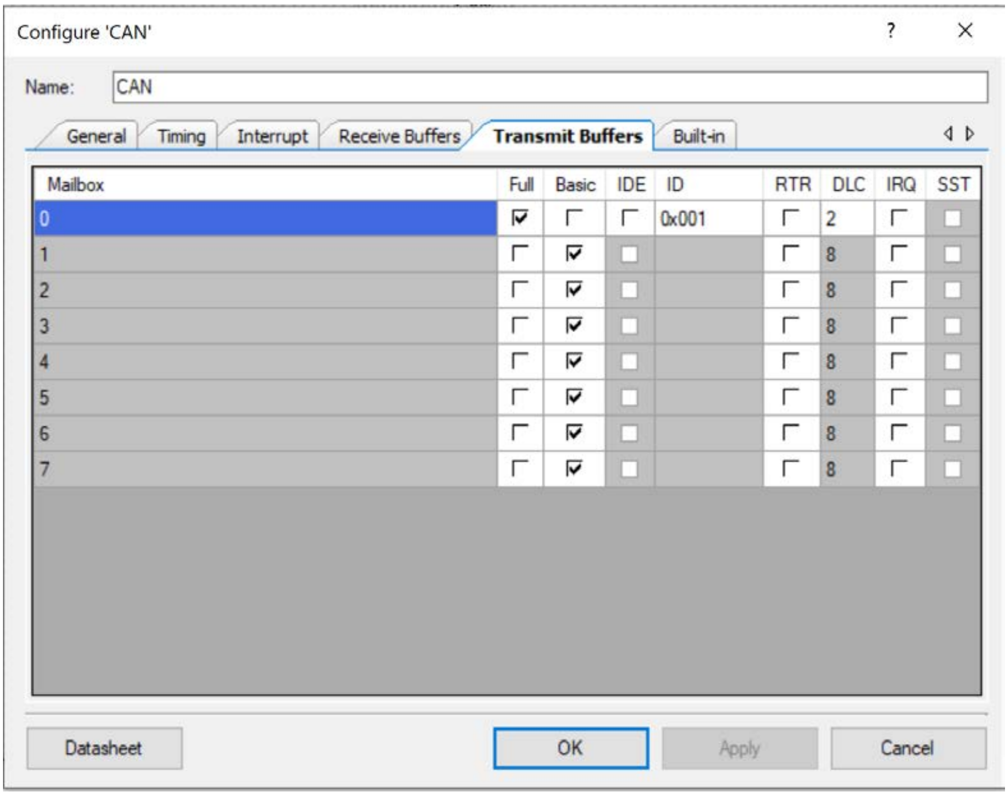


Figura 9.2.1.2: buffer di trasmissione del componente CAN del nodo Master di rete

Il CAN bus realizzato nel presente progetto è composto da tre nodi, due Slave e uno Master, necessari per simulare il collegamento al bus stesso di più nodi Slave; si è tuttavia scelto di implementarne un numero contenuto perché, nel caso di futura applicazione della rete di comunicazione progettata in un contesto pratico reale, l'aggiunta di più nodi richiederebbe semplicemente la realizzazione della circuiteria hardware già illustrata e l'attuazione di semplici modifiche software. L'interesse dell'autore è quello di progettare e sviluppare una rete di comunicazione funzionante, replicabile e modificabile: per soddisfare tali propositi il collegamento di due soli nodi Slave al CAN bus è più che sufficiente.

In figura 9.2.1.1 si deve notare come sia utilizzata e configurata una diversa mailbox di ricezione a seconda dell'ID del messaggio che si vuole leggere sul bus; la mailbox 0 viene infatti riservata alla ricezione del messaggio Full CAN caratterizzato dall'identificativo 0x002 (e quindi proveniente da un nodo specifico della rete), mentre la mailbox 1 a quello caratterizzato dall'identificativo 0x003 (proveniente quindi da un differente nodo della rete). In tal modo non solo il kit CY8CKIT-059 operante da Master è in grado di leggere e ricevere i messaggi sul CAN bus specificati da tali identificativi, ma anche di distinguerli fra loro e riconoscere da che particolare nodo della rete provengano. In figura 9.2.1.2 invece si può notare come sia stata configurata solamente la mailbox 0 di trasmissione; il nodo Master infatti non invia sul CAN bus alcun messaggio se non quello contraddistinto dall'ID 0x001 che rappresenta la richiesta ai nodi Slave della rete di inviare i valori di tensione da loro acquisiti dalle rispettive batterie che monitorano.

Si deve prestare particolare attenzione in questa fase della progettazione e realizzazione del CAN bus all'associare un diverso identificativo, sia nelle mailbox di ricezione che di trasmissione, ad ogni messaggio CAN scambiato sul bus per assicurare la corretta esecuzione della fase di arbitraggio del bus secondo il protocollo CSMA/CA; l'ID definisce infatti in modo univoco un particolare messaggio e l'assegnazione di uno stesso valore all'identificativo a messaggi diversi comporta non tanto un malfunzionamento del CAN bus, quanto un'errata ricezione da parte dei diversi nodi della rete, siano Slave o Master, del messaggio stesso.

9.2.2. Configurazione del componente UART

Una volta configurato il componente CAN secondo le indicazioni appena fornite, si procede alla caratterizzazione dei parametri del componente UART come evidenziato in figura 9.2.2.1 e 9.2.2.2.

Il componente UART è configurato in modalità Full UART (TX + RX) e quindi abilitato sia alla ricezione che trasmissione di dati sulla porta seriale che connette il kit CY8CKIT-059 operante da Master al PC. La velocità di trasmissione impostata è pari a 9600 bps (bit al secondo), i bits contenuti nel data field tra quello di start e quello di stop del data package sono impostati al valore di 8 (corrispondente quindi a un byte) e non viene configurata alcuna particolare parità per la rilevazione di eventuali errori di comunicazione che possano alterare i dati dei messaggi scambiati. Per concludere si imposta l'aggiunta di un solo bit di stop per indicare la terminazione di un messaggio.

Nella finestra mostrata in figura 9.2.2.2 invece si possono configurare delle impostazioni avanzate del componente UART, sia a livello software che hardware; per maggiori informazioni si rimanda al capitolo 7.1 in cui si sono approfondite le diverse scelte possibili.

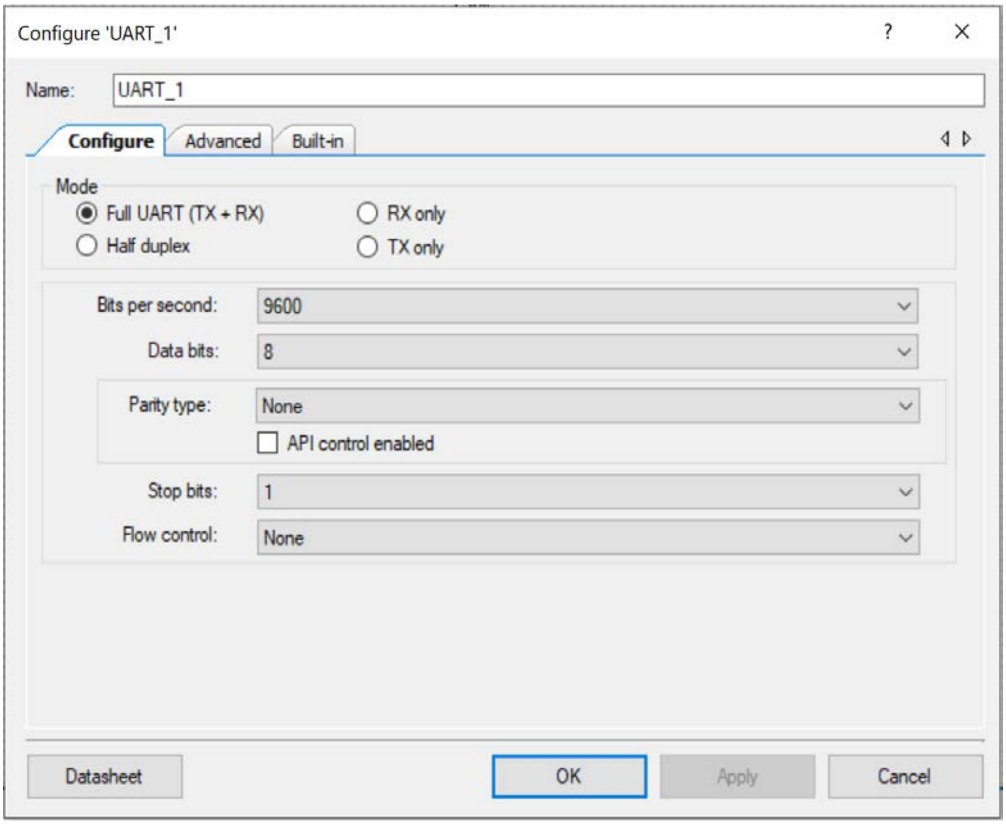
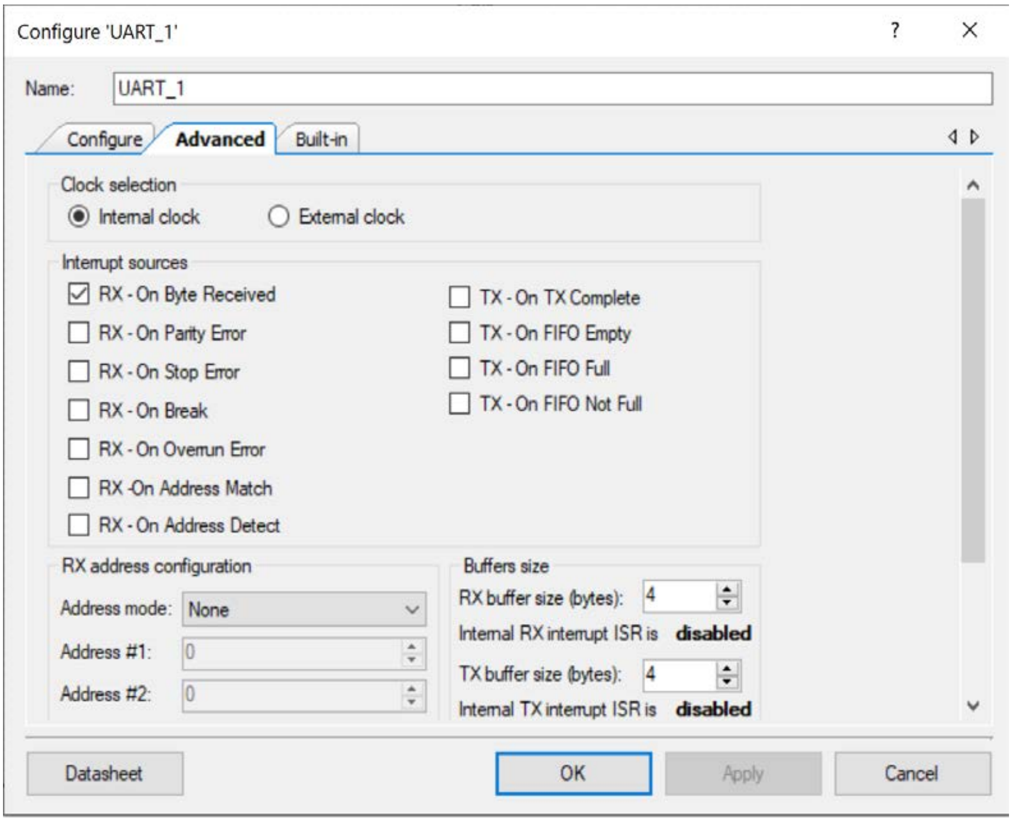


Figura 9.2.2.1: finestra di configurazione del componente UART



9.2.2.2: finestra di configurazione avanzata del componente UART

9.2.3. Programmazione del kit CY8CKIT-059 operante da nodo Master del CAN bus

Dopo aver configurato i componenti UART e CAN del componente CY8CKIT-059 secondo le indicazioni descritte nel precedente capitolo, affinché la scheda Cypress operi da nodo Master della rete occorre che il file 'TopDesign.cysch 'del relativo progetto di PSoC Creator appaia come in figura 9.2.3.1.

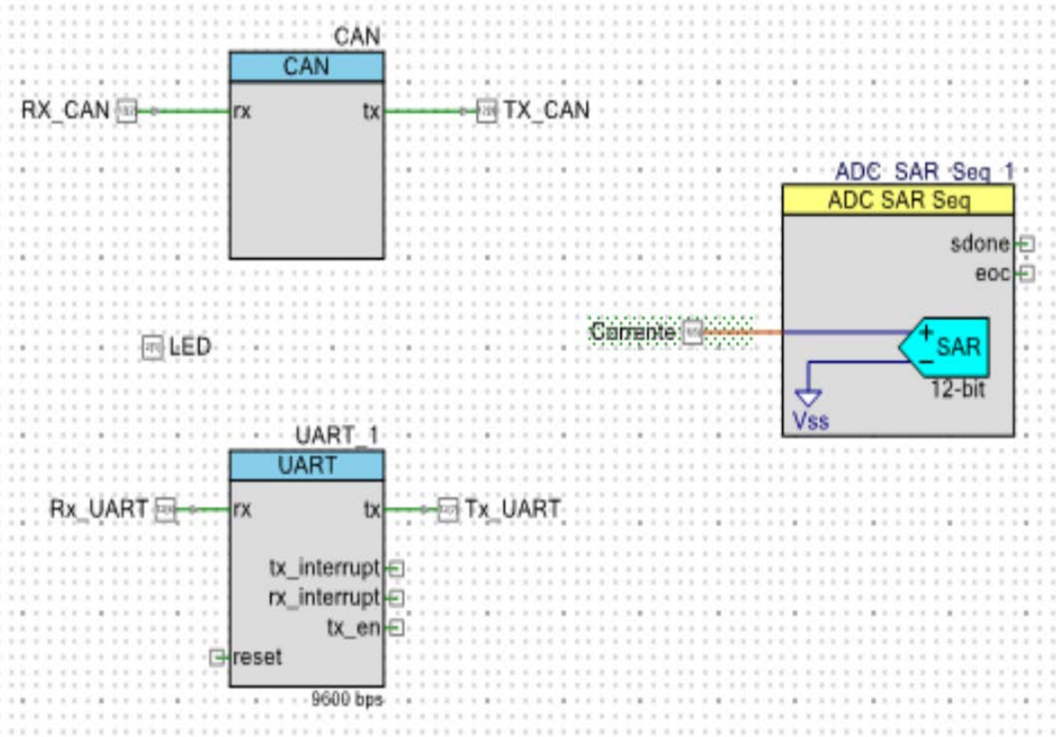


Figura 9.2.3.1: file 'TopDesign.cysch' di PSoC Creator per implementare un nodo Master della rete di comunicazione del CAN bus

L'assegnazione dei pin digitali RX_1 e TX_1 del componente CAN, oltre che di quelli digitali Rx_2 e Tx_2 del componente UART, può essere effettuata liberamente, l'unico vincolo è rispettare la tipologia di pin che il kit CY8CKIT-059 mette a disposizione; si consiglia l'utilizzo dei pin I/O generici che possono essere utilizzati sia come digitali che analogici.
Così come illustrato nel capitolo 9.1 per un nodo Slave del CAN bus, anche il kit CY8CKIT-059 in esame deve essere programmato opportunamente affinché operi da Master della rete di comunicazione e ciò richiede lo sviluppo di un appropriato codice in linguaggio C all'interno del file 'main.c' nel progetto corrispondente in PSoC Creator. Secondo il principio di

funzionamento sviluppato dall'autore della trattazione, la scheda legge sulla porta seriale un particolare carattere ASCII inviato dal PC indicante che quest'ultimo è pronto all'acquisizione e alla rappresentazione grafica in real time dei parametri delle batterie monitorate; il nodo Master allora invia sul CAN bus un frame codificato come richiesta dei valori di tensione acquisiti da ogni singolo nodo Slave connesso al bus. In risposta, secondo la loro programmazione illustrata nel capitolo 9.1, tali nodi inviano nel CAN bus i dati a loro disposizione; il nodo Master legge ogni singolo messaggio, conoscendo l'ordine secondo cui i dati sono stati inseriti in ogni byte del CAN frame, e ricostruisce il valore di tensione corrispondente ad ogni batteria. Tale ricostruzione da singoli byte a int16 è necessaria poiché il componente CAN obbliga lo sviluppatore a definire byte per byte il data field del messaggio CAN che si vuole inviare o ricevere, mentre ogni valore di tensione ottenuto dalla conversione operata dai componenti ADC_SAR_Seq dei nodi Slave è salvato in due byte (int16 in linguaggio C). Il nodo Master si fa quindi carico di ottenere nuovamente le tensioni della batteria in questo formato, acquisire il valore di corrente della batteria monitorata dal kit stesso e successivamente di trasmetterli al PC mediante porta seriale, sfruttando il componente UART del kit CY8CKIT-059 configurato secondo le indicazioni fornite nel presente capitolo. È necessario parallelamente lo sviluppo di un unico codice Matlab che consenta al PC di acquisire i valori dalla porta seriale e rappresentarli in un grafico per ottenere degli andamenti facilmente interpretabili delle grandezze delle batterie.

10. Test sperimentali effettuati sul sistema di monitoraggio distribuito basato sul CAN bus

Per convalidare il progetto illustrato fino a questo punto sono stati effettuati tre esperimenti: in primo luogo un test per verificare il corretto utilizzo dei componenti elettronici impiegati e una corretta realizzazione dei collegamenti tra di essi; in secondo luogo una verifica della rete di comunicazione nella sua interezza sfruttando degli alimentatori in continua e un reostato variabile; infine una prova di monitoraggio della carica e scarica di quattro batterie al piombo acido presenti in laboratorio, per ottenere consapevolezza dell'effettivo funzionamento della rete di comunicazione progettata e realizzata nell'ambito di applicazione di un sistema di monitoraggio distribuito. Nel seguito della trattazione vengono quindi presentati e approfonditi gli script di PSoC Creator, relativi sia ai nodi Slave che a quello Master del CAN bus, e quello Matlab utilizzati per eseguire i diversi test, e viene anche condotta un'analisi dei risultati sperimentali ottenuti, nel futuro contesto di applicazione di tale rete di comunicazione ad un sistema di monitoraggio distribuito della carica e scarica di una batteria di un veicolo elettrico.

10.1. Verifica della circuiteria hardware e del software della rete di comunicazione

Una volta realizzata la circuiteria hardware e sviluppata la parte software, prima di procedere con il test finale si è convenuto di testare il corretto funzionamento dei componenti elettronici e dei circuiti elettrici. L'idea alla base è alimentare mediante generatori di tensioni isolati in continua i due nodi Slave connessi al CAN bus e di far inviare a quest'ultimi sul CAN bus dei messaggi contenenti sequenze numeriche che variano a dente di sega, differenti l'uno dall'altro; in particolare tali sequenze hanno un diverso valore iniziale a cui poi viene applicato l'incremento di un contatore. Il Master acquisisce le sequenze e le invia al PC tramite porta seriale; visivamente il circuito hardware si presenta come in figura 10.1.1.

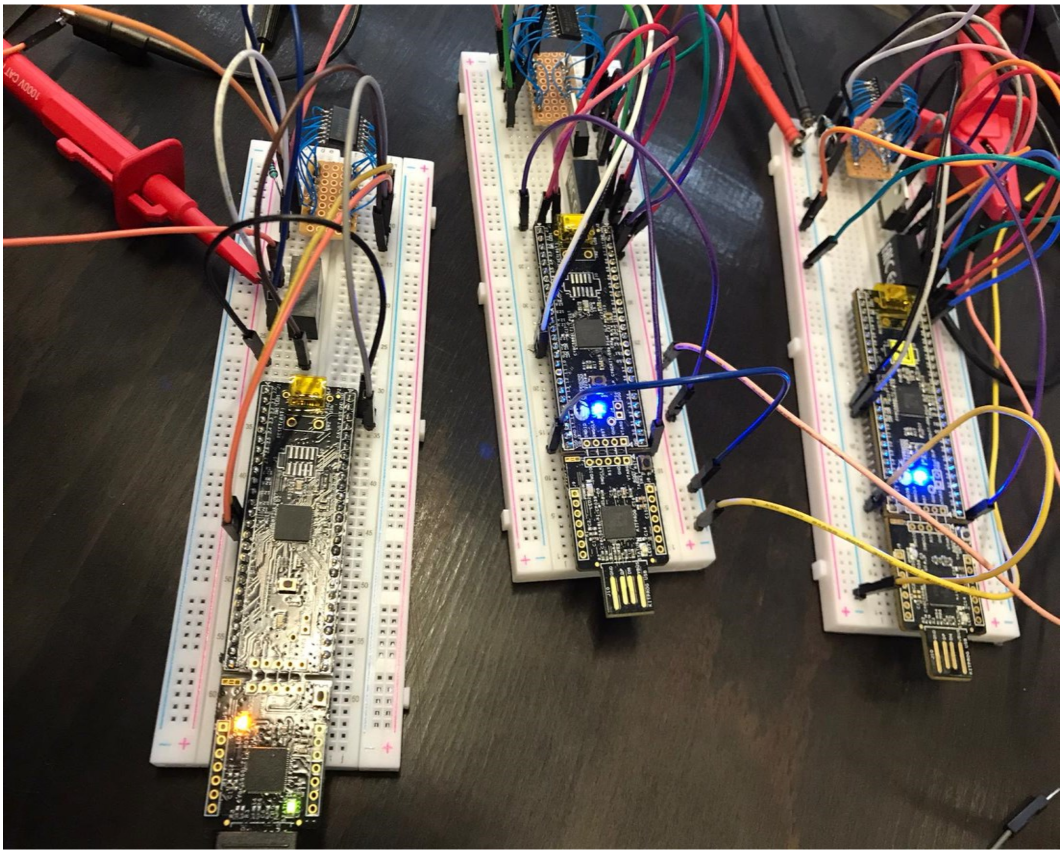


Figura 10.1.1: setup hardware dei componenti per il test

I circuiti associati ai due kit CY8CKIT-059 contraddistinti dal LED blu illuminato costituiscono i nodi Slave del CAN bus, mentre sulla sinistra è visibile il nodo Master connesso al PC mediante una prolunga USB tra KitProg del primo e porta USB del secondo.

Il circuito complessivo appena descritto, che non necessita della configurazione dei componenti ADC_SAR_Seq dei kit CY8CKIT-059 operanti da Slave poiché non si usano in tale esperimento delle batterie da monitorare, si traduce nei seguenti script con cui programmare rispettivamente i due kit Slave e il kit Master.

Il primo codice riportato è quello relativo alla programmazione del kit CY8CKIT-059 nell’ambiente di lavoro PSoC Creator configurato per agire da nodo Slave 1 e inviare sul bus una sequenza a dente di sega con valore iniziale 0 e incremento del valore di 1 ad ogni ciclo della main routine.

```c
/* ========================================*/
#include <project.h>
#include <stdio.h>
/* ===================================== */
/* Funzione principale */
int main(void)
{
    CyGlobalIntEnable;

    int16 tensione;
    int16 corrente;

    /*  Inizializzazione del componente CAN */

    CAN_Init();
    CAN_Start();
    int16 rampa;
    rampa=0;

    for(;;)
       {
        LED_Write(1);
        CAN_ReceiveMsg0();
        int8 richiesta;
        richiesta=CAN_RX_DATA_BYTE1(CAN_RX_MAILBOX_0);

        if (richiesta==1)
        {
            if(rampa<=9)
            {
                int8 rampa_array[2];
                rampa_array[0]=rampa >>8 & 0xFF;
                rampa_array[1]=rampa & 0xFF;
               CAN_TX_DATA_BYTE1(CAN_TX_MAILBOX_0) = rampa_array[0];
               CAN_TX_DATA_BYTE2(CAN_TX_MAILBOX_0) = rampa_array[1];
                CAN_SendMsg0();
                LED_Write(0);
            }

            if(rampa>9)
            {
            rampa=0;
            CAN_TX_DATA_BYTE1(CAN_TX_MAILBOX_0) = 0;
            CAN_TX_DATA_BYTE2(CAN_TX_MAILBOX_0) = 0;
            LED_Write(0);
            CAN_SendMsg0();
            }
            richiesta=0;
            rampa++;
        }
        CAN_RX_BUF_DISABLE(0);
        CAN_RX_BUF_ENABLE(0);
        CyDelay(5000);
    }
}

   /* [] END OF FILE */
```

Il secondo codice riportato in seguito invece è quello relativo alla programmazione del kit CY8CKIT-059 configurato per agire da nodo Slave 2 e inviare sul bus un segnale a dente di sega con valore iniziale 2 e incremento del valore di 1 ad ogni ciclo della main routine, così da ottenere un andamento diverso da quello del nodo 1 che permetta di verificare mediante grafico Matlab che i due segnali acquisiti siano differenti e che quindi l'intero sistema sviluppato operi correttamente.

```
/* ========================================*/
#include <project.h>
#include <stdio.h>
/* ===================================== */
/* Funzione principale */
int main(void)
{
    CyGlobalIntEnable;

    int16 tensione;
    int16 corrente;

    /*  Inizializzazione del componente CAN */

    CAN_Init();
    CAN_Start();
    int16 rampa;
    rampa=2;

    for(;;)
       {
        LED_Write(1);
        CAN_ReceiveMsg0();
        int8 richiesta;
        richiesta=CAN_RX_DATA_BYTE1(CAN_RX_MAILBOX_0);

        if (richiesta==1)
        {
            if(rampa<=9)
            {
                int8 rampa_array[2];
                rampa_array[0]=rampa >>8 & 0xFF;
                rampa_array[1]=rampa & 0xFF;
               CAN_TX_DATA_BYTE1(CAN_TX_MAILBOX_0) = rampa_array[0];
               CAN_TX_DATA_BYTE2(CAN_TX_MAILBOX_0) = rampa_array[1];
                CAN_SendMsg0();
                LED_Write(0);
            }

            if(rampa>9)
            {
            rampa=0;
            CAN_TX_DATA_BYTE1(CAN_TX_MAILBOX_0) = 0;
            CAN_TX_DATA_BYTE2(CAN_TX_MAILBOX_0) = 0;
            LED_Write(0);
            CAN_SendMsg0();
            }
```

```
            richiesta=0;
            rampa++;
        }
        CAN_RX_BUF_DISABLE(0);
        CAN_RX_BUF_ENABLE(0);
        CyDelay(5000);
    }
}

/* [] END OF FILE */
```

Queste due routine, dopo aver inizializzato il componente CAN del kit CY8CKIT-059 corrispondente, implementano un ciclo ‘for’ infinito che prevede, in seguito alla ricezione dei nodi Slave del messaggio trasmesso dal nodo Master sul bus per richiedere i dati in loro possesso (segnali fittizi a dente di sega), di inviare mediante un frame CAN il valore definito nel codice come “rampa” (int16), opportunamente suddiviso nei due singoli byte che lo compongono per simulare la scomposizione dei valori di tensione necessaria per il test finale. Ogni nodo Slave viene programmato per incrementare il valore di “rampa” fino ad un limite stabilito (rampa $\leq$ 9; se si verifica la condizione rampa $>$ 9 il suo valore viene riportato a 0) ogni qualvolta il nodo invia un messaggio sul CAN bus, così da ottenere l’andamento caratteristico di un segnale a dente di sega.

Oltre ai due codici appena illustrati relativi ai nodi Slave, è necessario programmare opportunamente anche il kit CY8CKIT-059 che si intende far funzionare come Master del CAN bus; si riporta nel seguito lo script sviluppato a tal proposito.

```
/* ========================================*/
#include <project.h>
#include <stdio.h>
/* ======================================== */
/* Funzione principale */
int main(void)
{
    // Attivazione degli interrupt globali
    CyGlobalIntEnable;
    uint8 cc;

    /* Inizializzazione dei componenti CAN e UART */
    CAN_Init();
    CAN_Start();
    UART_1_Start();

    for(;;)
```

```
        cc = UART_1_GetChar();

        if('t' == cc)
        {
        LED_Write(1);

        int8 richiesta=1;

        CAN_TX_DATA_BYTE1(CAN_TX_MAILBOX_0)=richiesta;
        CAN_SendMsg0();

        CAN_ReceiveMsg0();
        CAN_ReceiveMsg1();

        int8 rampa_bytes1[2];
             rampa_bytes1[0] = CAN_RX_DATA_BYTE1(CAN_RX_MAILBOX_0);
             rampa_bytes1[1] = CAN_RX_DATA_BYTE2(CAN_RX_MAILBOX_0);

        int16 rampa1 = (rampa_bytes1[0] << 8) + rampa_bytes1[1];

        int8 rampa_bytes2[2];
             rampa_bytes2[0] = CAN_RX_DATA_BYTE1(CAN_RX_MAILBOX_1);
             rampa_bytes2[1] = CAN_RX_DATA_BYTE2(CAN_RX_MAILBOX_1);

        int16 rampa2= (rampa_bytes2[0] << 8) + rampa_bytes2[1];

        char TransmitBuffer[20];

        // indice per trasmettere il buffer via UART al PC

        UART_1_PutChar('x');
        sprintf(TransmitBuffer, "\r%d %d", rampa1, rampa2);

        UART_1_PutString(TransmitBuffer);

        LED_Write(0);

        cc='a';
        }
    }
}

/* [] END OF FILE */
```

La main routine appena riportata prevede, dopo aver inizializzato il componente CAN e UART, di leggere la porta seriale; se il kit CY8CKIT-059 riceve il carattere ASCII ‘t’ allora riconosce la richiesta da parte del PC di trasmettere i valori in possesso dei nodi Slave della rete. Il suddetto nodo agente da Master del CAN bus invia un CAN frame con ID 0x001 per indicare ai nodi Slave di trasmettere nel bus i valori di tensione di cui dispongono. Mediante opportune API, consultabili nel corrispettivo datasheet dei diversi componenti utilizzati ed approfondite nell'Appendice A, il nodo Master riceve i frame CAN inviati dai due nodi Slave collegati al bus, ricostruisce il valore dei due segnali “rampa1” e “rampa2” salvandoli come int16 e li invia tramite porta seriale al PC.

Per completare il processo, come già anticipato in precedenza, è necessario sviluppare parallelamente uno script Matlab che in primo luogo indichi al nodo Master di essere pronto a ricevere i dati, in secondo luogo riceva tali valori e infine li renda disponibili visivamente all'utente mediante due opportuni grafici aggiornati in real time. Si riporta qui in seguito il codice Matlab sviluppato a tal proposito.

```
clc
clear

s = serialport("COM4", 9600);

n=1;

rampa1=[1];
rampa2=[1];

while(1)

    writeline(s,'t');
    pronto = read(s,1,'char');
    if(pronto == 'x')
        messaggio = read(s,4,'string');
        messaggio_numeri=str2num(messaggio);

        rampa1(n,1)=messaggio_numeri(1);
        rampa2(n,1)=messaggio_numeri(2);

        tiledlayout(2,1)

        % real time di tutti i dati
        nexttile
        plot(rampa1,'b')
        hold on
        plot(rampa2,'r')
        hold on
        grid on
        title('Plot 1')

        % Bottom plot
        nexttile
        axis([n-3 n+1 0 9])
        plot(rampa1,'b')
        hold on
        axis([n-3 n+1 0 9])
        plot(rampa2,'r')
        hold on
        grid on
```

```
        title('Plot 2')
        n=n+1;
        pause(4.98)
        end
end
```

Nello script appena riportato si definisce dapprima una porta seriale ‘s’ mediante la funzione ‘serialport’ che richiede in input il numero del terminale USB (visualizzabile nel computer su ‘Gestione dispositivi’→’Porte (COM e LPT)’) cui il kit CY8CKIT-059 Master è connesso al PC e la velocità di trasmissione dati che deve coincidere con quella impostata sul componente UART del nodo Master. Si procede poi definendo i due vettori “rampa1” e “rampa2” in cui si salvano i rispettivi valori real time dei segnali associati (inviati al Master tramite CAN bus dai nodi Slave) mediante la funzione specifica ‘read’ che richiede come input il nome della porta seriale e la quantità, oltre alla tipologia dei dati che deve leggere. In particolare, si è implementato un ciclo ‘if’ affinché Matlab inizi a leggere i dati solo quando riceve lo specifico carattere ASCII ‘x’ dalla porta seriale, inviatogli dal kit CY8CKIT-059 Master per comunicare che dispone dei dati precedentemente richiesti. Una volta acquisiti i valori dei due segnali mediante i comandi ‘tiledlayout’ e ‘nexttile’ Matlab realizza due grafici, visibili in figura 10.1.2 corrispondenti all’acquisizione di una trentina di valori inviati dai nodi Slave (dopodiché si è interrotta l’esecuzione del codice poiché sono già una quantità di dati sufficiente per dimostrare il corretto funzionamento della rete di comunicazione realizzata). Il grafico in alto mostra l’evoluzione dei due segnali nel tempo, considerando i due vettori “rampa1” e “rampa2” nella loro interezza e quindi utilizzando tutti i loro valori dal primo all’ultimo acquisito; il grafico viene sovrascritto ad ogni nuovo valore letto dalla porta seriale. Il secondo grafico in basso invece restituisce l’andamento dei due segnali considerando solo i loro cinque ultimi valori, fornendo così una prospettiva istantanea e più dettagliata della loro evoluzione nel breve periodo; ciò si traduce dunque in un grafico scorrevole, cha aggiorna in real time l’andamento più recente dei segnali.

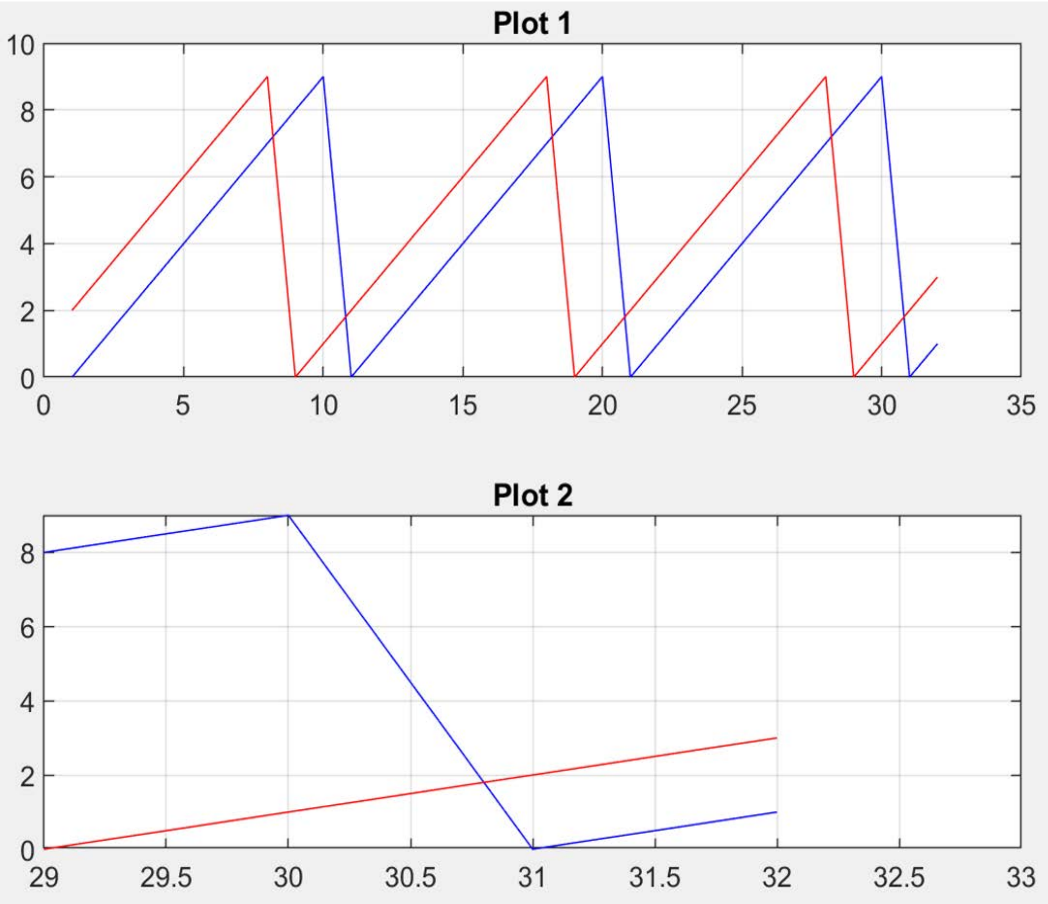


Figura 10.1.2: figura restituita da Matlab e aggiornata in real time dei segnali acquisiti dalla porta seriale

Per maggiore chiarezza visiva, il segnale blu ("rampa1") corrisponde a quello generato dal nodo Slave 1 del CAN bus mentre il rosso ("rampa2") a quello generato dal nodo Slave 2. I due segnali, come specificato all'inizio del capitolo ed evidenziato negli script con cui sono stati programmati i due kit CY8CKIT-059 agenti da Slave, sono stati realizzati affinché la loro differenza in modulo fosse sempre uguale a 2 e permettesse di distinguerli; visivamente si può notare come questa specifica sia rispecchiata dai grafici restituiti da Matlab.

Questo risultato costituisce la prova che la rete di comunicazione progettata e realizzata è perfettamente funzionante sia dal punto di vista del circuito hardware che per quanto concerne la programmazione software; non solo infatti è garanzia della corretta comunicazione seriale tra PC e nodo Master, ma anche della comunicazione secondo protocollo CAN tra quest'ultimo e i nodi Slave connessi al CAN bus. La corretta visualizzazione dei segnali indica nel complesso che tutti i componenti elettronici scelti sono utilizzati rispettando le loro caratteristiche elettriche e funzionali e che i vari kit CY8CKIT-059 sono programmati rispettando il proposito dell'utente. In definitiva il primo risultato sperimentale ottenuto costituisce un'importante e fondamentale riprova della progettazione concettuale della rete di comunicazione basata sul CAN bus, oltre che della realizzazione fisica del circuito. Il passo successivo del progetto è l'applicazione del sistema di acquisizione e comunicazione così implementato alla misura delle tensioni generate da alimentatori in corrente continua applicate ad un reostato variabile, al fine di simulare le condizioni operative che deve garantire il sistema di monitoraggio distribuito della carica e scarica di più batterie; esso deve essere infatti in grado di acquisire correttamente tensioni e correnti variabili nel tempo con elevata precisione. Tale test viene riportato nel seguente sotto-capitolo.

10.2. Applicazione della rete di comunicazione ad un sistema di alimentatori DC e un reostato variabile

Prima di applicare la rete di comunicazione ad un sistema di monitoraggio distribuito della carica e scarica di una batteria, è opportuno verificare il corretto funzionamento del progetto nella sua interezza; rispetto al precedente test condotto nel sotto-capitolo 10.1 vengono collegati alla rete anche i circuiti di acquisizione della tensione e della corrente illustrati nel capitolo 8.1. In particolare, il circuito di acquisizione della corrente è invariato rispetto a quello sviluppato dal collega Samuel Matrella, mentre quello di acquisizione della tensione è leggermente modificato; si sono aggiunti infatti nel filtro passa-basso dei condensatori in parallelo alla resistenza tra V- e V_{OUT} per migliorare la capacità di filtraggio dei disturbi legati alla tensione degli alimentatori.

A livello funzionale nella configurazione sviluppata per il suddetto test ognuno dei due nodi Slave della rete di comunicazione acquisisce la tensione, variabile manualmente da parte dell'utente, in uscita da un diverso canale dell'alimentatore in continua per poi trasmetterla nel CAN bus, mentre il nodo Master acquisisce la corrente in uscita da un terzo canale dell'alimentatore che fornisce 5V costanti con corrente massima di 3A, raccoglie i dati sul CAN bus e invia i due valori di tensione e quello di corrente tramite interfaccia USB-UART al PC. Visivamente il circuito hardware si presenta come in figura 10.2.1.

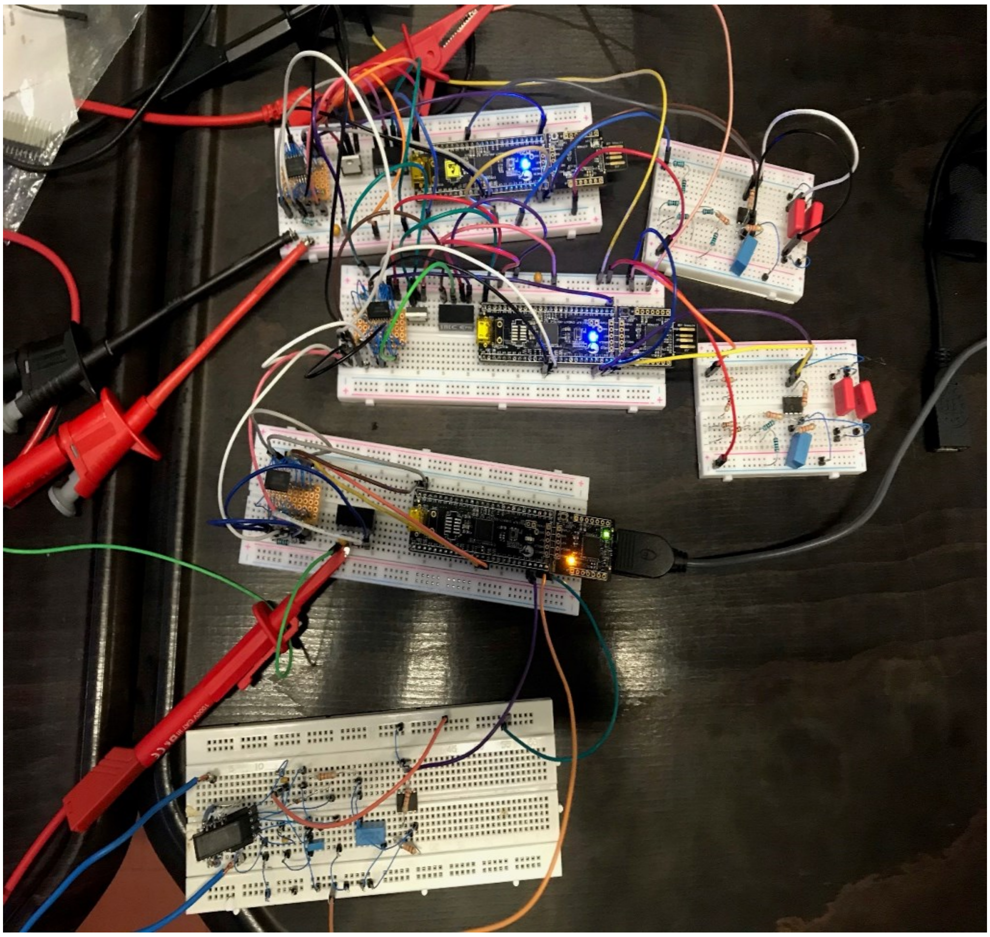


Figura 10.2.1: setup hardware dei componenti per il test

I circuiti in alto in figura 10.2.1, associati ai due kit CY8CKIT-059 contraddistinti dal LED blu illuminato, costituiscono i nodi Slave del CAN bus mentre in basso è visibile il nodo Master con relativo circuito di acquisizione della corrente della batteria, connesso poi al PC mediante una prolunga USB tra KitProg della scheda e porta USB del computer. Il processo di funzionamento descritto all'inizio di questo sotto-capitolo si traduce a livello software nei seguenti script con cui programmare rispettivamente i due kit CY8CKIT-059 Slave e il kit Master.

Il primo codice riportato in seguito è quello relativo alla programmazione dei due kit CY8CKIT-059 nell'ambiente di lavoro PSoC Creator configurati per agire da nodo Slave 1 e 2 della rete di comunicazione: ad ogni ciclo della main routine trasmettono sul CAN bus, qualora leggano il CAN frame di richiesta dei dati inviato dal nodo Master, i valori di tensione acquisiti dalle batterie monitorate mediante i circuiti di acquisizione connessi ad essi.

```
* ======================================== */

#include <project.h>
#include <stdio.h>        // libreria standard input-output
/* ===================================== */
/* Funzione principale */
int main(void)
{
    CyGlobalIntEnable;

    int16 tensione;   // definisco 'tensione' come un intero salvato
                      // in 2 byte

    /*  Inizializzazione dei componenti ADC_SAR_Seq e CAN */

    CAN_Init();
    CAN_Start();

    ADC_SAR_Seq_1_Start();
    ADC_SAR_Seq_1_StartConvert();

    for(;;)
       {
        CAN_ReceiveMsg0();
        int8 richiesta;
        richiesta=CAN_RX_DATA_BYTE1(CAN_RX_MAILBOX_0);

        if (richiesta==1)
        {

if(ADC_SAR_Seq_1_IsEndConversion(ADC_SAR_Seq_1_SAR_RETURN_STATUS))
         {
            LED_Write(1);

            uint16 ris_tensione = ADC_SAR_Seq_1_GetResult16(0);

            tensione = ADC_SAR_Seq_1_CountsTo_mVolts(ris_tensione);

            uint8 tensione_MSB = tensione >> 8 & 0xFF;

                                                    di

            uint8 tensione_LSB = tensione  & 0xFF;
```

```
            CAN_TX_DATA_BYTE1(CAN_TX_MAILBOX_0) = tensione_MSB;
            CAN_TX_DATA_BYTE2(CAN_TX_MAILBOX_0) = tensione_LSB;
            CAN_SendMsg0();

            CyDelay(450); // msec che intercorrono tra l'invio di un
                          // messaggio e l'altro sul bus
            }
        }
        richiesta=0;
        LED_Write(0);
    }
}

/* ===================================== */
/* [] END OF FILE */
```

La routine espressa nel codice, dopo aver inizializzato il componente CAN e ADC_SAR_Seq del kit CY8CKIT-059, implementa un ciclo ‘for’ infinito che prevede, in seguito alla ricezione dei nodi Slave del messaggio (ID 0x001) trasmesso dal nodo Master sul CAN bus per richiedere i dati in loro possesso, di inviare mediante CAN frame il valore di tensione (int16) ottenuto dal proprio circuito di acquisizione e opportunamente suddiviso nei singoli byte. Si deve ricordare infatti che l’API ‘CAN_SendMsgX()’, necessaria per inviare il CAN frame associato al buffer di trasmissione della mailbox X, richiede che i dati da inviare mediante il componente CAN debbano essere scomposti nei singoli byte con cui sono salvati; mediante poi l’API ‘CAN_TX_DATA_BYTE_Y(CAN_TX_MAILBOX_X)’ è possibile impostare il byte numero Y della mailbox di trasmissione X con un valore int8 da inviare.

Oltre al codice appena riportato, relativo ad entrambi i nodi Slave, è necessario programmare opportunamente anche il kit CY8CKIT-059 che si intende far funzionare come Master della rete di comunicazione; si riporta nel seguito lo script sviluppato a tal proposito.

```
* ======================================== */

#include <project.h>
#include <stdio.h>        // libreria standard input-output

/* ======================================== */
/* Funzione principale */

int main(void)
{
```

```
    // Attivazione degli interrupt globali
    CyGlobalIntEnable;

    // Definizione carattere (intero) che serve per trasmettere i
    // valori di tensione e corrente al PC quando quest'ultimo
    // comunica sulla porta seriale di essere pronto a riceverli

    uint8 cc;

    /* Inizializzazione dei componenti CAN, UART e ADC_SAR_Seq  */

    CAN_Init();
    CAN_Start();
    ADC_SAR_Seq_1_Start();
    ADC_SAR_Seq_1_StartConvert();
    UART_1_Start();

    for(;;)
    {
        cc = UART_1_GetChar();

        if('t' == cc)
        {
         LED_Write(1);

         int8 richiesta=1;

         CAN_TX_DATA_BYTE1(CAN_TX_MAILBOX_0)=richiesta;
         CAN_SendMsg0();

         CAN_ReceiveMsg0();

         uint8 tensione1_MSB = CAN_RX_DATA_BYTE1(CAN_RX_MAILBOX_0);
                             // byte più significante di 'tensione1'
         uint8 tensione1_LSB = CAN_RX_DATA_BYTE2(CAN_RX_MAILBOX_0);
                             // byte meno significativo di 'tensione1'
            uint16 tensione1 = tensione1_MSB*256 + tensione1_LSB;
                             // valore di tensione salvato in due byte


        CAN_ReceiveMsg1();

         uint8 tensione2_MSB = CAN_RX_DATA_BYTE1(CAN_RX_MAILBOX_1);
                             // byte più significante di 'tensione2'
         uint8 tensione2_LSB = CAN_RX_DATA_BYTE2(CAN_RX_MAILBOX_1);
                             // byte meno significativo di 'tensione2'
            uint16 tensione2 = tensione2_MSB*256 + tensione2_LSB;
                             // valore di tensione salvato in due byte


if(ADC_SAR_Seq_1_IsEndConversion(ADC_SAR_Seq_1_SAR_RETURN_STATUS))
            {
             uint16 c=ADC_SAR_Seq_1_GetResult16(0);
             uint16 corrente = ADC_SAR_Seq_1_CountsTo_mVolts(c);

             char TransmitBuffer[20];
             UART_1_PutChar('x');
sprintf(TransmitBuffer, "\r%d %d %d",tensione1,tensione2,corrente);
        UART_1_PutString(TransmitBuffer);   // invio al PC il valore
```

```
                // di tensione ricevuto dal nodo
                LED_Write(0);
               }
           }
       }
}
/* [] END OF FILE */
```

La main routine appena riportata prevede, dopo aver inizializzato i componenti CAN, UART e ADC_SAR_Seq, di leggere un carattere dalla porta seriale; qualora il kit CY8CKIT-059 riceva il carattere ASCII ‘t’, allora riconosce la richiesta da parte del PC di trasmettergli i valori in possesso dei diversi nodi della rete. Il suddetto nodo allora in quanto Master del CAN bus invia in esso un CAN frame con ID 0x001 per indicare ai nodi Slave di trasmettere i propri valori di tensione di cui dispongono. Mediante opportune API, consultabili nel corrispettivo datasheet dei diversi componenti utilizzati ed approfondite nell’Appendice A a fine trattazione, il nodo Master riceve i frame CAN inviati dai due nodi Slave collegati al bus, ricostruisce il valore dei due segnali “tensione1” e “tensione2” salvandoli come int16 e li invia tramite porta seriale al PC assieme al valore di corrente acquisito dal canale 0 del componente ADC_SAR_Seq, proveniente dal segnale restituito dal relativo circuito di acquisizione.

Per completare il processo, come già anticipato in precedenza, è necessario sviluppare parallelamente uno script Matlab che in primo luogo indichi al nodo Master che il PC è pronto a ricevere i dati, in secondo luogo legga tali valori sulla porta seriale e infine li renda disponibili visivamente all’utente mediante due opportuni grafici aggiornati in real time. Si riporta qui in seguito il codice sviluppato a tal proposito.

```
% prova RTR

clc
clear all

s = serialport("COM4", 9600);

n=1;

tensione1=[1];
tensione2=[1];
corrente=[1];
```

```
while(1)

    writeline(s,'t');
    pronto = read(s,1,'char');
    if(pronto == 'x')
       {
        messaggio = read(s,15,'string');
        messaggio_numeri=str2num(messaggio)/1000;

      tensione1(n,1)=messaggio_numeri(1)*3*1.00540504+0.0319;
      tensione2(n,1)=messaggio_numeri(2)*3*1.0033023-0.0148;
 corrente(n,1)=5*(messaggio_numeri(3)-2.5)/(3.12-2.5)*0.7618-0.7076;

        figure(1)
        tiledlayout(2,2)

        % real time dei dati di tensione
        nexttile
        plot(tensione1,'b')
        hold on
        plot(tensione2,'r')
        hold on
        grid on
        title('Andamento delle tensioni dei due nodi')

        % andamento locale delle tensioni
        nexttile
        axis([n-3 n+1 10 15])
        plot(tensione1,'b')
        hold on
        axis([n-3 n+1 10 15])
        plot(tensione2,'r')
        hold on
        grid on
        title('Andamento locale delle tensioni')

        % real time del valore di corrente
        nexttile
        plot(corrente,'b')
        hold on
        grid on
        title('Andamento corrente')

        % andamento locale della corrente
        nexttile
        plot(corrente,'b')
        axis([n-3 n+1 0 3])
        hold on
        grid on
        title('Andamento recente della corrente')

        n=n+1;
        pause(0.5)

    end

end
```

Il codice appena riportato è simile a quello illustrato nel capitolo 10.1 relativo al precedente esperimento. Vale la pena tuttavia commentare come i valori di tensione e corrente acquisiti dal PC tramite la rete di comunicazione siano rielaborati secondo delle apposite formule prima di essere visualizzati graficamente. Le tensioni in ingresso ai kit CY8CKIT-059 operanti da Slave infatti sono ottenute applicando a quelle in uscita dai rispettivi canali dell'alimentatore in continua una divisione mediante un partitore di tensione; in particolare, come illustrato in precedenza, tali tensioni vengono divise per 3 affinché si ottengano dei valori consoni a quelli ammissibili in ingresso alle schede ($V_{MAX\_INPUT} < 5V$). Allo stesso modo la sonda LEM utilizzata per l'acquisizione della corrente ritorna in uscita un valore di tensione proporzionale alla corrente in ingresso, come visto nel sotto-capitolo 8.1. I circuiti di acquisizione delle tensioni e della corrente però introducono anche un guadagno e un offset nella trasduzione dei relativi segnali che li attraversano; tale modifica dei valori reali di tensione e corrente, determinata principalmente dalla realizzazione fisica di tali circuiti, viene evidenziata nel seguito assieme ai codici sviluppati per il calcolo degli opportuni coefficienti con cui rielaborare i dati in input a Matlab al fine di correggere tali offset e guadagni.

Lo script Matlab sviluppato per la correzione dei valori di tensione acquisiti dai due canali dell'alimentatore in continua è dunque il seguente.

```
clc
% tensioni nodo 1 prima della correzione

MATLAB_tensione1=[9.9655,10.4988,10.9383,11.4473,11.9841,12.4879,
                  13.0048,13.4898,13.9681,14.2240];
ALIMENTAZIONE_tensione1=[10.05,10.59,11.04,11.53,12.07,12.58,13.11,
                         13.59,14.07,14.33];
x=1:1:10;

correlazione_nodo1=polyfit(ALIMENTAZIONE_tensione1,MATLAB_tensione1,
                           1);

% tensioni nodo 2 prima della correzione
MATLAB_tensione2=[10.0580,10.6007,11.0420,11.5408,12.0738,12.5856,
                  13.1076, 13.5844,14.0639,14.3200];
ALIMENTAZIONE_tensione2=[10.09,10.62,11.07,11.56,12.10,12.61,13.13,
                         13.61,14.10,14.36];

correlazione_nodo2=polyfit(ALIMENTAZIONE_tensione2,MATLAB_tensione2,
                           1);
```

```
scarto_tensione1=MATLAB_tensione1-ALIMENTAZIONE_tensione1;
scarto_tensione2=MATLAB_tensione2-ALIMENTAZIONE_tensione2;

% calcolo offset e guadagno nodo1
gain_tensione1=1/correlazione_nodo1(1)

offset_tensione1=-mean(((MATLAB_tensione1.*gain_tensione1)-
                         ALIMENTAZIONE_tensione1))

% calcolo offset e guadagno nodo2
gain_tensione2=1/correlazione_nodo2(1)

offset_tensione2=-mean(((MATLAB_tensione2.*gain_tensione2)-
                         ALIMENTAZIONE_tensione2))

% plot dati prima della correzione
figure(1)
tiledlayout(2,2)

nexttile
axis([1 10 10 15])
plot(ALIMENTAZIONE_tensione1,'b')
hold on
axis([1 10 10 15])
plot(MATLAB_tensione1,'r')
hold on
grid on
legend('dati reali','dati acquisiti')
title('Nodo 1 prima della correzione')

nexttile
axis([1 10 10 15])
plot(ALIMENTAZIONE_tensione2,'b')
hold on
axis([1 10 10 15])
plot(MATLAB_tensione2,'r')
hold on
grid on
legend('dati reali','dati acquisiti')
title('Nodo 2 prima della correzione')

nexttile
plot(scarto_tensione1,'k')
axis([1 10 -0.11 0])
hold on
grid on
legend('differenza tra dati grezzi')
title('Scarto dati nodo 1 prima della correzione')

nexttile
plot(scarto_tensione2,'k')
axis([1 10 -0.11 0])
hold on
grid on
legend('differenza tra dati grezzi')
title('Scarto dati nodo 2 prima della correzione')

figure(2)
tiledlayout(1,2)
```

```
nexttile
plot(ALIMENTAZIONE_tensione1,MATLAB_tensione1,'b+:')
axis([9 15 9 15])
hold on
plot(ALIMENTAZIONE_tensione1,ALIMENTAZIONE_tensione1,'r+:')
axis([9 15 9 15])
hold on
grid on
legend('matlab-alimentazione','alimentazione-alimentazione')
title('Offset e guadagno tensione nodo 1')

nexttile
plot(ALIMENTAZIONE_tensione2,MATLAB_tensione2,'b+:')
axis([9 15 9 15])
hold on
plot(ALIMENTAZIONE_tensione2,ALIMENTAZIONE_tensione2,'r+:')
axis([9 15 9 15])
hold on
grid on
legend('matlab-alimentazione','alimentazione-alimentazione')
title('Offset e guadagno tensione nodo 2')


% tensioni nodo 1 dopo la correzione

MATLAB_tensione1_finale=[10.0865,10.5157,10.9876,11.5077,12.0064,
                         12.5016,12.9769,13.4949,13.9405,14.3850];
ALIMENTAZIONE_tensione1_finale=[10.08,10.51,10.99,11.51,12.01,12.50,
                                12.98, 13.49,13.94,14.39];
correlazione_finale_nodo1=polyfit(ALIMENTAZIONE_tensione1_finale,
                                  MATLAB_tensione1_finale,1);

% tensioni nodo 2 dopo la correzione
MATLAB_tensione2_finale=[10.1063,10.5468,11.0227,11.5281,12.0233,
                         12.5214,13.0023,13.5104,13.9586,14.4076];
ALIMENTAZIONE_tensione2_finale=[10.11,10.54,11.02,11.53,12.03,12.52,
                                13.00,13.51,13.96,14.41];
correlazione_finale_nodo2=polyfit(ALIMENTAZIONE_tensione2_finale,
                                  MATLAB_tensione2_finale,1);

scarto_tensione1_finale=MATLAB_tensione1_finale-
                        ALIMENTAZIONE_tensione1_finale;
scarto_tensione2_finale=MATLAB_tensione2_finale-
                        ALIMENTAZIONE_tensione2_finale;

% calcolo offset e guadagno finali nodo1
gain_tensione1_finale=1/correlazione_finale_nodo1(1);

offset_tensione1_finale=-mean(((MATLAB_tensione1_finale.*
   gain_tensione1_finale)-ALIMENTAZIONE_tensione1_finale));

% calcolo offset e guadagno finali nodo2
gain_tensione2_finale=1/correlazione_finale_nodo2(1);

offset_tensione2_finale=-mean(((MATLAB_tensione2_finale.*
   gain_tensione2_finale)-ALIMENTAZIONE_tensione2_finale));

% plot dati dopo la correzione
```

```
figure(3)
tiledlayout(2,2)

nexttile
axis([1 10 10 15])
plot(ALIMENTAZIONE_tensione1_finale,'b')
hold on
axis([1 10 10 15])
plot(MATLAB_tensione1_finale,'r')
hold on
grid on
legend('dati reali','dati acquisiti corretti')
title('Nodo 1 dopo la correzione')

nexttile
axis([1 10 10 15])
plot(ALIMENTAZIONE_tensione2_finale,'b')
hold on
axis([1 10 10 15])
plot(MATLAB_tensione2_finale,'r')
hold on
grid on
legend('dati reali','dati acquisiti corretti')
title('Nodo 2 dopo la correzione')

nexttile
plot(scarto_tensione1_finale,'k')
axis([1 10 -0.01 0.01])
hold on
grid on
legend('differenza tra dati corretti')
title('Scarto dati nodo 1 dopo la correzione')

nexttile
plot(scarto_tensione2_finale,'k')
axis([1 10 -0.01 0.01])
hold on
grid on
legend('differenza tra dati corretti')
title('Scarto dati nodo 2 dopo la correzione')

figure(4)
tiledlayout(1,2)

nexttile
plot(ALIMENTAZIONE_tensione1_finale,MATLAB_tensione1_finale,'b+:')
axis([9 15 9 15])
hold on
plot(ALIMENTAZIONE_tensione1_finale,ALIMENTAZIONE_tensione1_finale,'
r+:')
axis([9 15 9 15])
hold on
grid on
legend('matlab-alimentazione','alimentazione-alimentazione')
title('Offset e guadagno finali tensione nodo 1')

nexttile
plot(ALIMENTAZIONE_tensione2_finale,MATLAB_tensione2_finale,'b+:')
axis([9 15 9 15])
```

```
hold on
plot(ALIMENTAZIONE_tensione2_finale,ALIMENTAZIONE_tensione2_finale,'
r+:')
axis([9 15 9 15])
hold on
grid on
legend('matlab-alimentazione','alimentazione-alimentazione')
title('Offset e guadagno finali tensione nodo 2')
```

Per operare le opportune correzioni di guadagno e offset dei valori di tensione acquisiti dai due nodi della rete si effettuano diverse misurazioni mediante voltmetro delle tensioni in uscita dai canali dell'alimentatore in continua (vettori 'ALIMENTAZIONE_tensione1' e 'ALIMENTAZIONE_tensione2' nello script); per ognuno di questi valori si esegue il programma Matlab riportato precedentemente per la richiesta e la raccolta dei valori di tensione acquisiti dalla rete di comunicazione tramite porta seriale e se ne esegue una media (vettori 'MATLAB_tensione1' e 'MATLAB_tensione2' nello script) su un centinaio di campioni, così da diminuire l'influenza delle fluttuazioni della tensione in uscita dai canali dell'alimentatore. La discordanza tra i valori riportati da Matlab e quelli misurati per ogni nodo della rete, visibile in figura 10.2.2, rappresenta l'alterazione dei segnali dovuta alla circuiteria hardware costituente la rete di acquisizione stessa; le diverse saldature operate, oltre che i collegamenti realizzati tra i diversi componenti, vanno infatti ad influenzare le tensioni acquisite dal kit CY8CKIT-059. Si può notare come il nodo 1 presenti un'alterazione dei dati acquisiti rispetto a quelli misurati molto più marcata del nodo 2; ciò può essere dovuto alla realizzazione fisica dei rispettivi circuiti di acquisizione, oltre che alla fluttuazione intrinseca delle tensioni proveniente dai rispettivi canali dell'alimentatore in continua. Il valore massimo di tale scarto infatti per il nodo 1 della rete di comunicazione è circa pari a 0.11V, mentre quello del nodo 2 non supera gli 0.04V come si può notare nei due grafici in basso in figura 10.2.2.

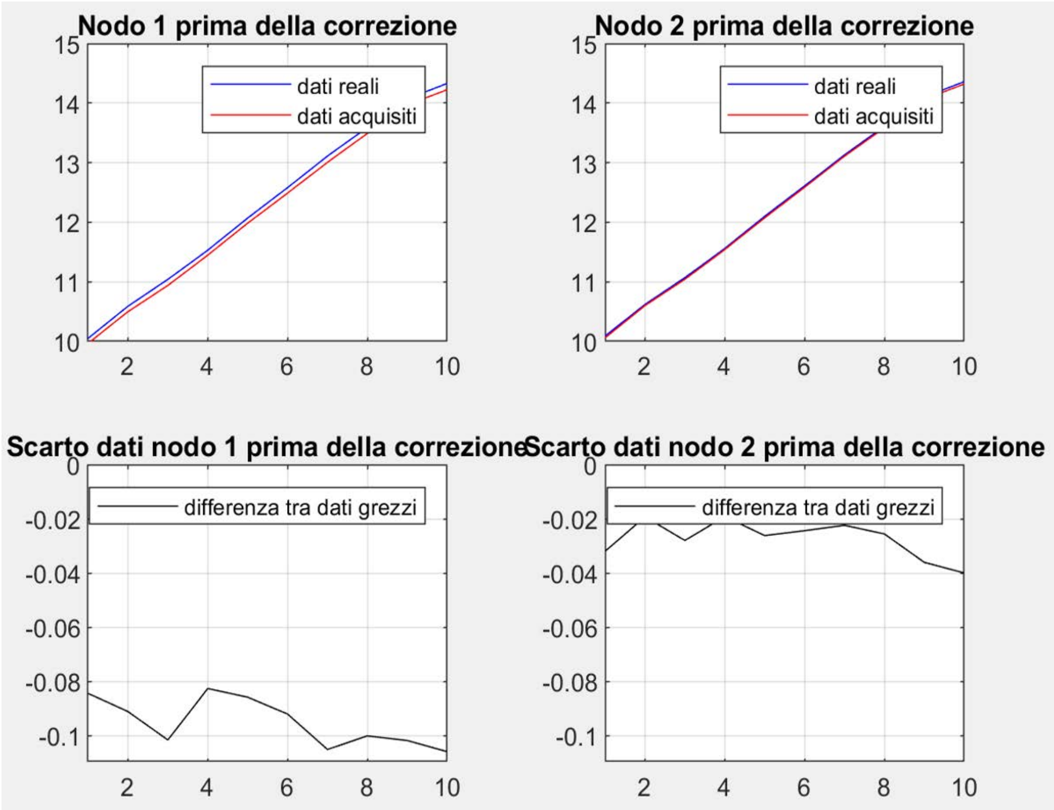


Figura 10.2.2: discordanza tra i valori di tensione misurati e quelli acquisiti

Oltre ad un confronto in valore assoluto dei valori misurati ed acquisiti per ognuno dei due nodi della rete, evidenziato nella figura 10.2.2, è significativo ricavare anche una correlazione tra tali dati mettendoli in relazione l'uno all'altro come si vede in figura 10.2.3. I due grafici infatti mostrano i valori di tensione acquisiti dai due nodi (ordinate) rapportati ai relativi valori misurati tramite voltmetro (ascisse); nel caso ideale , ovvero in assenza di perturbazioni dei segnali da parte dei circuiti fisici di acquisizione, la retta così definita dovrebbe coincidere con la bisettrice (correlazione tra valori misurati e loro stessi), mentre nel caso reale l'alterazione dei segnali ne provoca uno scostamento, più o meno marcato in funzione dell'entità dell'offset e del guadagno presenti.

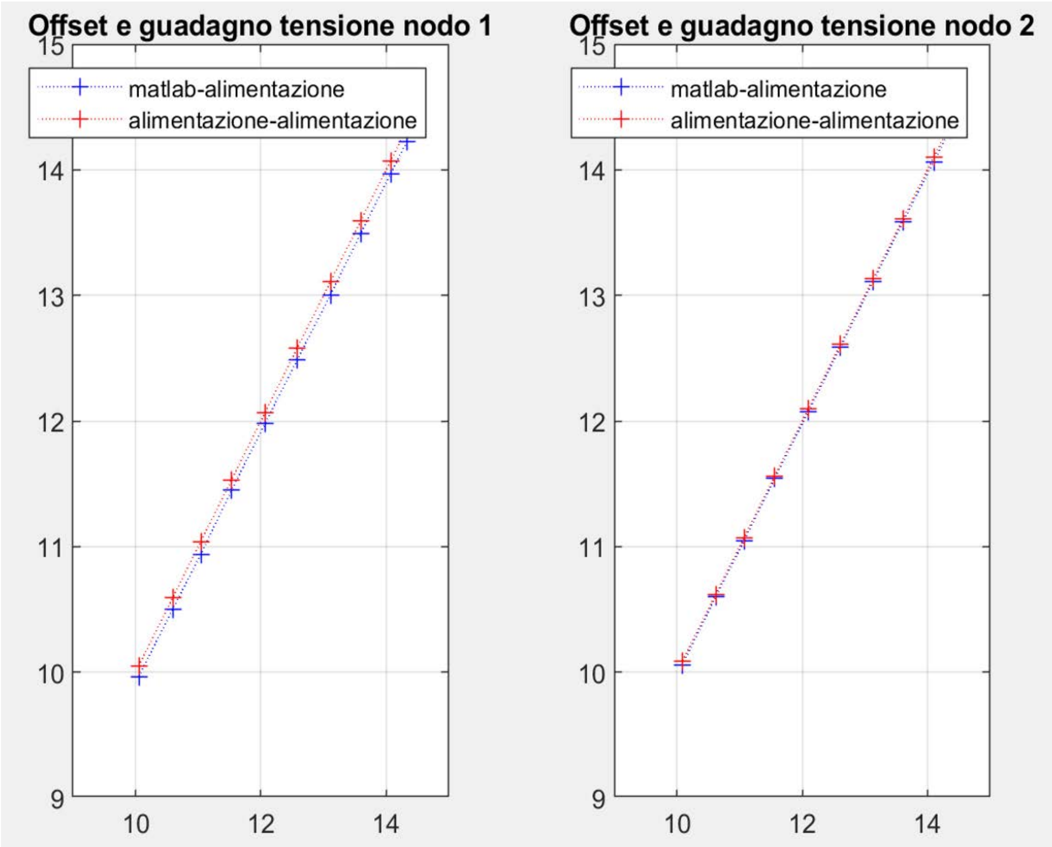


Figura 10.2.3: correlazione tra i valori di tensione acquisiti e quelli misurati per ogni nodo

Si procede allora a calcolare i coefficienti con cui correggere i dati di tensione acquisiti. In particolare, per eliminare il guadagno introdotto dai circuiti fisici si devono moltiplicare i valori di tensione acquisiti dalla rete di comunicazione per l'inverso del coefficiente angolare della retta ottenuta dalla correlazione tra dati misurati e dati acquisiti per ogni canale, calcolabile mediante la funzione 'polyfit' di Matlab.

```
correlazione_nodo=polyfit(ALIMENTAZIONE_tensione,MATLAB_tensione,1);
gain_tensione=1/correlazione_nodo(1); % inverso del coeff. angolare
```

La funzione 'P=polyfit(X,Y,N)' infatti restituisce i coefficienti di un polinomio P(X) di grado N che si adattano meglio ai dati contenuti nel vettore Y, in particolare minimizzandone lo scarto quadratico medio. L'output P è un vettore riga di lunghezza N+1 contenente i coefficienti polinomiali ordinati in potenze discendenti 'P(1)*X^N + P(2)*X^(N-1) +...+ P(N)*X + P(N+1)'. Nel caso specifico del progetto in esame l'interpolazione scelta è quella lineare e di conseguenza il grado N del polinomio scelto è pari a 1. A questo punto per correggere l'offset si deve applicare ai dati acquisiti l'opposto della media della differenza tra i valori acquisiti moltiplicati per il guadagno appena calcolato e i valori misurati.

```
offset_tensione=-mean(((MATLAB_tensione.*gain_tensione)-
                        ALIMENTAZIONE_tensione));
```

Una volta effettuato questo procedimento di correzione di guadagno e offset per ognuno dei due nodi della rete e applicati i coefficienti ricavati allo script Matlab di acquisizione dei dati dalla porta seriale, per verificare l'efficacia della correzione effettuata si procede ad ripetere nuovamente l'intero procedimento appena descritto, raccogliendo nuove misurazioni tramite voltmetro e parallelamente nuove acquisizioni mediante la rete di comunicazione realizzata. I dati corretti ottenuti da ognuno dei nodi Slave del CAN bus, inviati al nodo Master e trasmessi poi tramite porta seriale al PC sono rappresentati in figura 10.2.4.

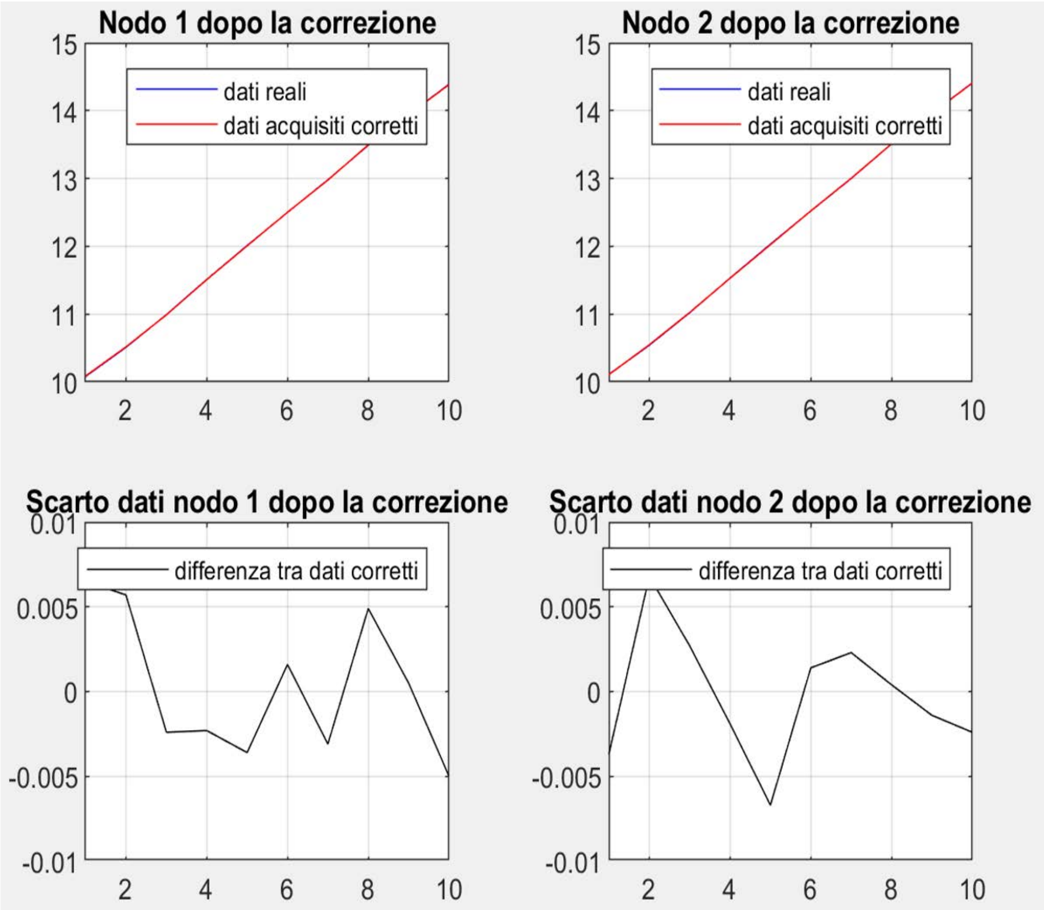


Figura 10.2.4: discordanza tra i valori di tensione misurati e quelli acquisiti dopo la correzione dell’offset e del guadagno introdotti dai circuiti di acquisizione

Si può notare visivamente dai due grafici in alto in figura 10.2.4 come, una volta corretti offset e guadagno dei due circuiti di acquisizione, i valori di tensione acquisiti corrispondano quasi perfettamente con quelli misurati tramite voltmetro. Lo scarto tra tali dati, comunque ancora presente, è visibile per ognuno dei nodi nelle due figure in basso; i picchi massimi che assume sono in entrambi i casi compresi entro gli 0.006V in valore assoluto e quindi trascurabili dal punto di vista numerico. Inoltre, il loro andamento è centrato sugli 0V, e quindi l’offset presente nei dati acquisiti nel tempo è pressoché nullo e la retta che mette in relazione i dati acquisiti rispetto a quelli misurati coincide di fatto con la bisettrice, come si può vedere in figura 10.2.5.

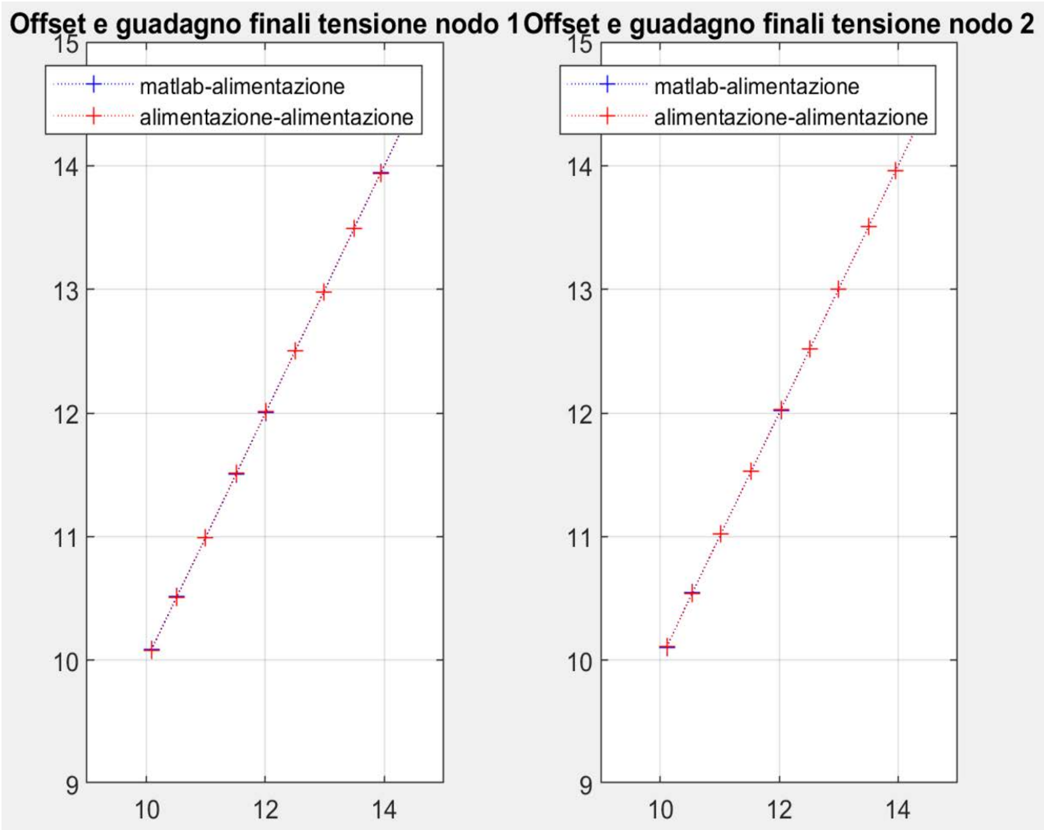


Figura 10.2.5: correlazione tra i valori di tensione acquisiti e quelli misurati per ogni nodo dopo la correzione

Un altro paragone che si effettua per validare la bontà della correzione effettuata è quello tra la figura 10.2.2 e 10.2.4; l'evidente riduzione sia dello scostamento tra l'andamento dei valori di tensione misurati ed acquisiti, che dello scarto tra coppie di dati, testimonia che dopo la correzione l'acquisizione dei dati da parte del sistema distribuito è affidabile e rispecchia con precisione il valore reale delle tensioni in uscita dai canali dell'alimentatore in continua. Il sistema di acquisizione delle tensioni è quindi pronto per essere applicato al monitoraggio distribuito di diverse batterie collegate tra loro.

Così come accade per le tensioni, anche la circuiteria hardware relativa all'acquisizione della corrente introduce un guadagno e un offset sui dati rilevati da Matlab, scostandoli di fatto da quelli reali misurati tramite amperometro. Lo script realizzato per calcolare gli opportuni coefficienti con cui correggere i dati acquisiti al fine di farli coincidere con quelli misurati è riportato qui nel seguito.

```
clc

% valori prima della correzione 1

MATLAB_corrente=[1.7004,2.3360,3.0250,3.6986,4.3992];
ALIMENTAZIONE_corrente=[0.51,1.01,1.53,2.04,2.57];
x=0:1:4;
corrente=polyfit(x,MATLAB_corrente,1);
ALIMENTAZIONE=polyfit(x,ALIMENTAZIONE_corrente,1);

correlazione=polyfit(ALIMENTAZIONE_corrente,MATLAB_corrente,1);
gain_corrente=1/correlazione(1);

offset_corrente=-mean(((MATLAB_corrente.*gain_corrente)-
                        ALIMENTAZIONE_corrente));

scarto=MATLAB_corrente-ALIMENTAZIONE_corrente;

figure(1)
tiledlayout(2,1)

nexttile
axis([1 5 0 4.5])
plot(MATLAB_corrente,'b')
hold on
axis([1 5 0 4.5])
plot(ALIMENTAZIONE_corrente,'r')
hold on
grid on
legend('dati acquisiti','dati reali')
title('Acquisizione corrente')

nexttile
plot(scarto,'k')
axis([1 5 1 2])
hold on
grid on
legend('differenza tra dati')
title('Scarto dati corrente')

figure(2)
plot(ALIMENTAZIONE_corrente,MATLAB_corrente,'b+:')
axis([0 5 0 5])
hold on
plot(ALIMENTAZIONE_corrente,ALIMENTAZIONE_corrente,'r+:')
axis([0 5 0 5])
hold on
grid on
legend('matlab-alimentazione','alimentazione-alimentazione')
title('Offset e guadagno')

%%%%%%%%%%%%%%%%%%%%%%%%%%%
% valori dopo la correzione

corrente_finale=[0.5083,1.0078,1.5345,1.9938,2.5046];
ALIMENTAZIONE_finale=[0.51,1.01,1.53,1.99,2.51];
x=0:1:4;
corrente_corretta=polyfit(x,corrente_finale,1);
ALIMENTAZIONE_standard=polyfit(x,ALIMENTAZIONE_finale,1);
```

```
scarto_finale=corrente_finale-ALIMENTAZIONE_finale;

correlazione_finale=polyfit(ALIMENTAZIONE_finale,corrente_finale,1);
gain_corrente_finale=1/correlazione_finale(1);
offset_corrente_finale=mean(((corrente_finale.*gain_corrente_finale)
                              -ALIMENTAZIONE_finale));

figure(3)
tiledlayout(2,1)
nexttile
axis([1 5 0 4.5])
plot(corrente_finale,'b')
hold on
axis([1 5 0 4.5])
plot(ALIMENTAZIONE_finale,'r')
hold on
grid on
legend('dati acquisiti corretti','dati reali')
title('Acquisizione corrente corretta')

nexttile
plot(scarto_finale,'k')
axis([1 5 -0.01 0.01])
hold on
grid on
legend('differenza tra dati corretti ')
title('Scarto dati corretti corrente')

figure(4)
plot(ALIMENTAZIONE_finale,corrente_finale,'b+:')
axis([0 5 0 5])
hold on
plot(ALIMENTAZIONE_finale,ALIMENTAZIONE_finale,'r+:')
axis([0 3 0 3])
hold on
grid on
legend('matlab-alimentazione','alimentazione-alimentazione')
title('Offset e guadagno finali')
```

Il procedimento implementato è concettualmente identico a quello realizzato per la correzione dei valori di tensione acquisiti; l'unica differenza sostanziale è che, mentre per le tensioni si devono correggere guadagno e offset introdotti dai relativi circuiti di acquisizione di entrambi i nodi della rete, la corrente da misurare e acquisire è unica dato che nella prova finale da realizzare si deve eseguire il monitoraggio di due batterie poste in serie tra loro e quindi attraversate dalla stessa corrente.

La discordanza presente tra i valori di corrente reali misurati tramite amperometro e quelli acquisiti è visibile in figura 10.2.6 e 10.2.7.

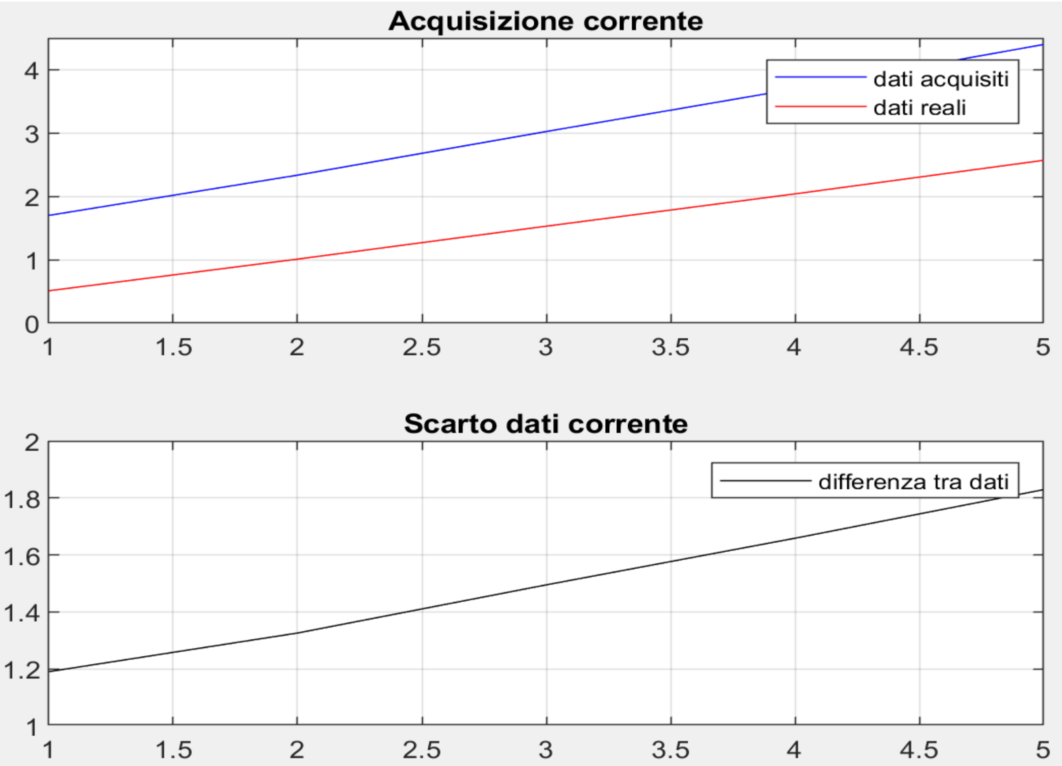


Figura 10.2.6: discordanza tra i valori di corrente misurati e quelli acquisiti

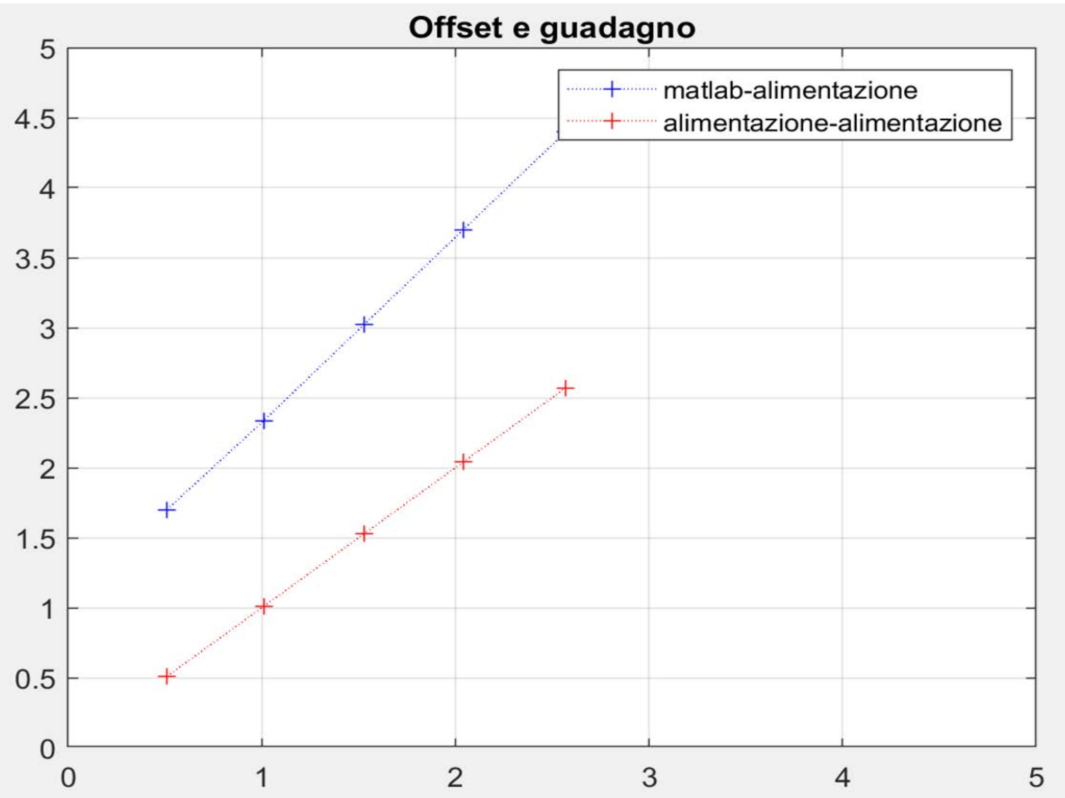


Figura 10.2.7: correlazione tra i valori di corrente acquisiti e quelli misurati

Una volta eseguito il calcolo dei coefficienti con cui correggere offset e guadagno introdotti dalla circuiteria hardware come già illustrato in precedenza per le tensioni, applicandoli allo script Matlab di acquisizione dei dati dalla porta seriale si ottengono gli andamenti corretti dei dati acquisiti dalla rete visibili in figura 10.2.8 e 10.2.9.

Anche in questo caso è evidente il risultato della correzione effettuata: l'andamento dei dati acquisiti coincide quasi perfettamente con quello dei dati misurati tramite amperometro come si può vedere in figura 10.2.8. Lo scarto tra dati misurati e acquisiti in valore assoluto è compreso entro gli 0.005A, indice di un'elevata precisione nell'acquisizione dei dati.

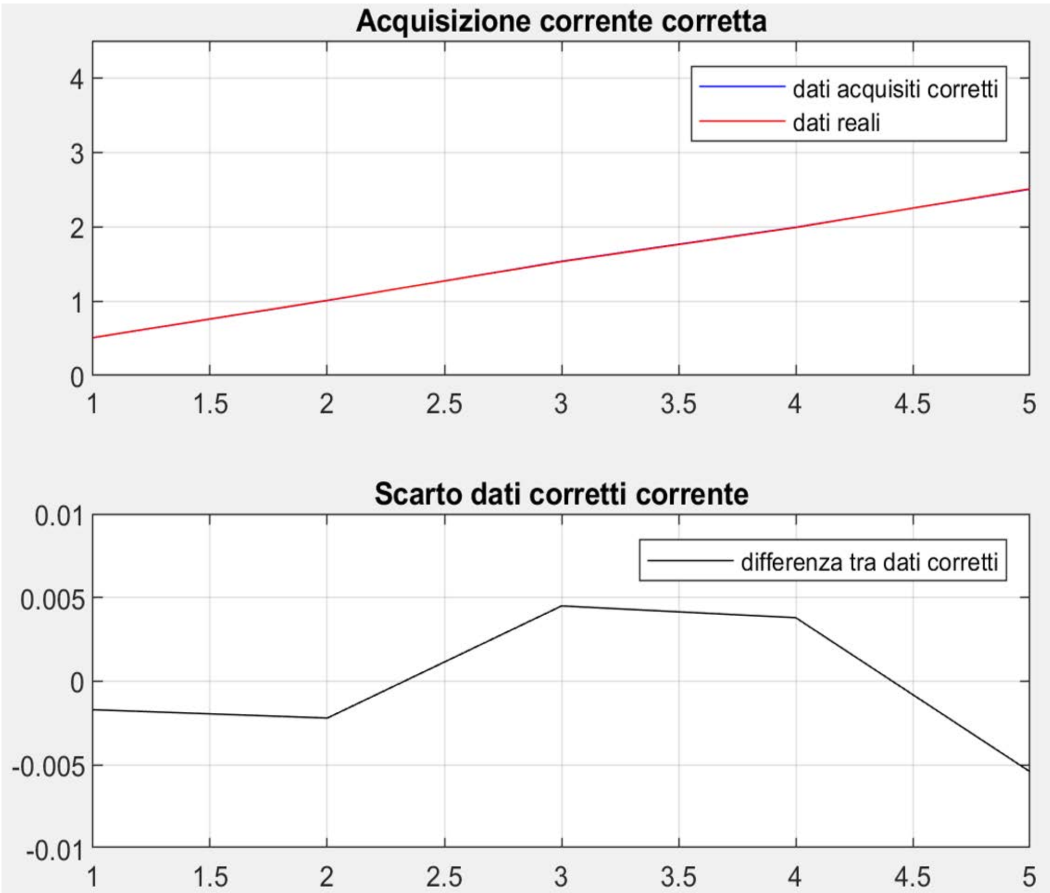


Figura 10.2.8: discordanza tra i valori di corrente misurati e quelli acquisiti dopo la correzione dell'offset e del guadagno introdotti dal circuito di acquisizione

Parallelamente in figura 10.2.9 si mette in evidenza la retta indicante la correlazione tra i dati di corrente acquisiti e misurati, confrontata con la bisettrice corrispondente alla correlazione tra dati misurati tramite amperometro e loro stessi; le due rette sono indistinguibili in quanto coincidenti, a testimonianza della validità dei coefficienti calcolati per la correzione dell'offset e guadagno presenti nell'acquisizione iniziale dei valori di corrente.

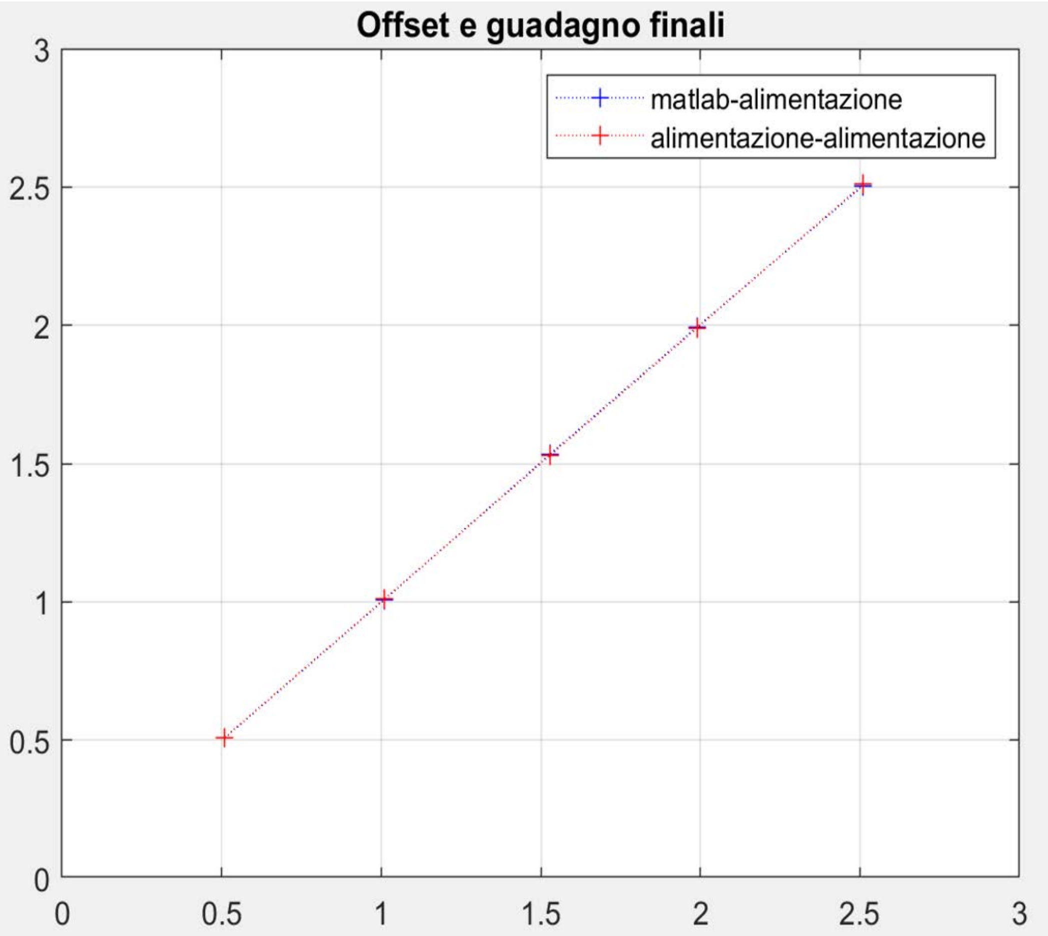


Figura 10.2.9: correlazione tra i valori di corrente acquisiti e quelli misurati dopo la correzione

Una volta corretti guadagno e offset dei circuiti di acquisizione delle tensioni e della corrente, si procede ad effettuare il test vero e proprio, andando a visualizzare in real time sia la corrente che le tensioni acquisite mediante la rete di comunicazione implementata. In particolare, i due canali regolabili dell'alimentatore in continua comandati in tensione permettono di cambiare la tensione stessa in uscita ai loro capi mentre l'utilizzo del reostato variabile, in combinazione con un terzo canale dell'alimentatore che mette a disposizione 5V costanti (con corrente uscente massima pari a 3A), consente di simulare la variazione di corrente da acquisire entro il range messo a disposizione dallo strumento di potenza. In figura 10.2.10 si possono vedere i grafici ottenuti dallo script Matlab in relazione al test eseguito.

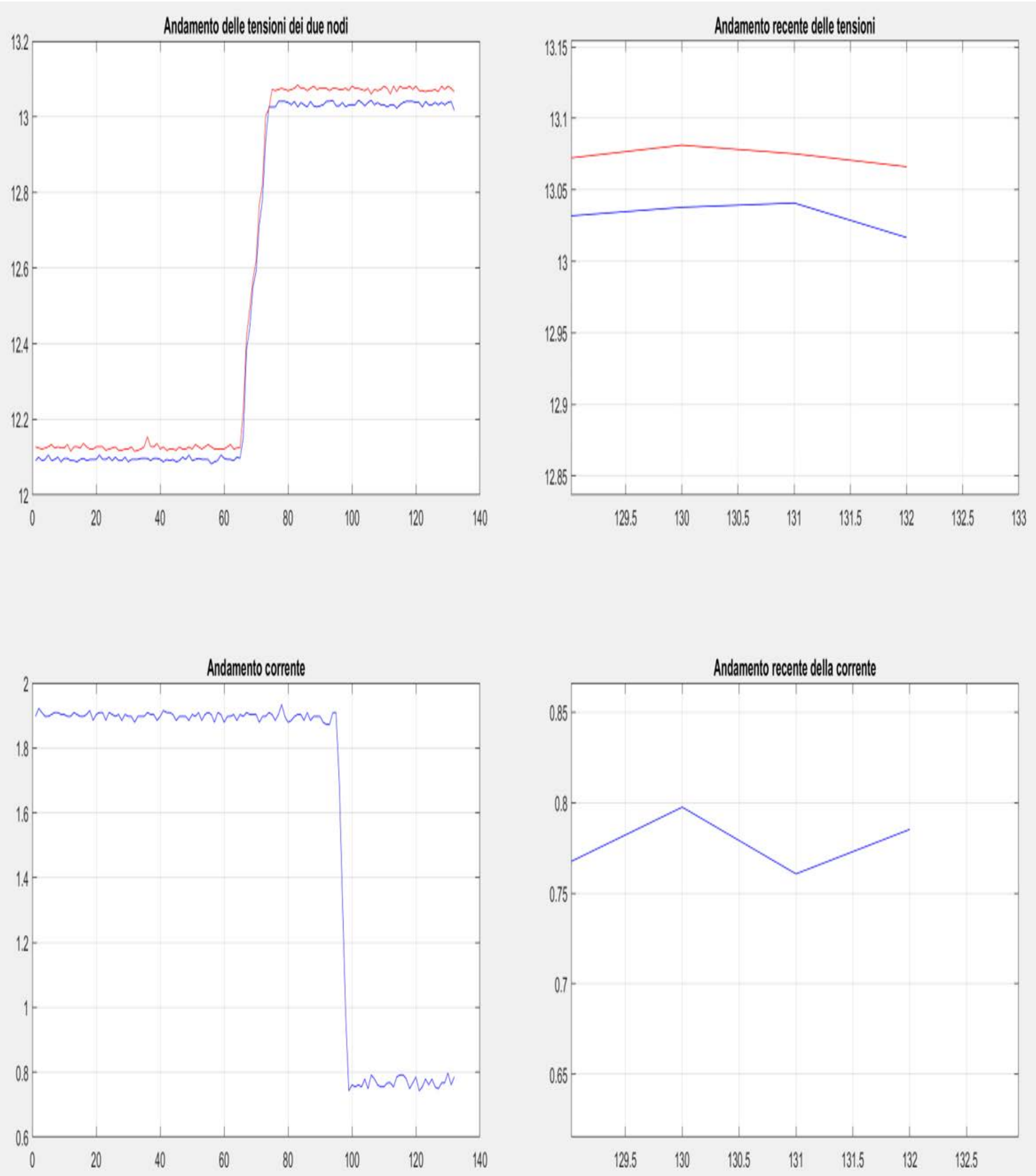


Figura 10.2.10: andamento delle tensioni e della corrente durante il test

La porta seriale è programmata nello script Matlab precedentemente illustrato in questo capitolo per richiedere ed acquisire i dati dalla rete di comunicazione ogni 0.5 secondi; il test è effettuato raccogliendo in totale 132 campioni nell'arco corrispettivo di un minuto e 6 secondi. Il numero di campioni e l'arco temporale scelti non sono comunque di particolare rilevanza; l'obiettivo dell'esperimento è monitorare l'andamento dei parametri dei vari canali dell'alimentatore che permettono all'utente di variare casualmente dapprima la tensione da 12.1V a 13.05V circa, visibile nei due grafici in alto in figura 10.2.10, e successivamente la corrente da 1.9A a 0.775A (aumentando casualmente la resistenza tramite il reostato variabile); tali variazioni sono ben visibili nei due grafici a sinistra di figura 10.2.10. Le variazioni di tensione e corrente non devono essere perfettamente a gradino; la pendenza della variazione stessa infatti dipende dalle azioni manuali dell'utente, la cui celerità confrontata con il periodo di acquisizione dei valori dalla porta seriale determina l'inclinazione della retta visibile nei grafici delle tensioni e della corrente. Maggiore è il tempo impiegato per la variazione manuale effettuata dallo sperimentatore, sia essa tramite manopole per regolare la tensione di uscita dell'alimentatore in continua oppure tramite reostato variabile, minore sarà la pendenza della variazione stessa restituita sui grafici.

Si può notare sempre in figura 10.2.10 come l'intero sistema di acquisizione risponda perfettamente in tempo reale alle variazioni delle grandezze monitorate, garantendo sempre accuratezza nei valori dei dati acquisiti e soddisfacendo di fatto le specifiche per cui è progettato e realizzato. A questo punto il passo conclusivo è applicare il sistema di acquisizione ad un sistema composto da due batterie al piombo acido poste in serie tra loro, per effettuare un monitoraggio distribuito della loro carica e scarica.

10.3. Applicazione del sistema di monitoraggio distribuito a quattro batterie poste in serie

Una volta effettuati tutti i test preliminari riportati nei sotto-capitoli 10.1 e 10.2 e verificato il corretto funzionamento del sistema di monitoraggio, l'esperimento finale condotto è l'applicazione del sistema a quattro batterie al piombo acido poste in serie tra loro al fine di monitorarne la tensione e la corrente durante le fasi di carica e scarica. In particolare, sono svolti e illustrati nel seguito gli esperimenti relativi ad entrambi i processi appena nominati.

Il primo test è realizzato monitorando il processo di carica delle batterie e necessita, oltre alla configurazione della rete di comunicazione già illustrata nel precedente sotto-capitolo, di un caricabatterie; si è scelto di utilizzare il dispositivo SM82A 48V di Murphy disponibile in laboratorio, caratterizzato da una tensione continua in uscita di 48V e corrente massima di 1.5A [15]. Una volta eseguito lo script Matlab per l'acquisizione dei valori di tensione e corrente tramite porta seriale, i grafici restituiti dal programma sono visibili in figura 10.3.1.

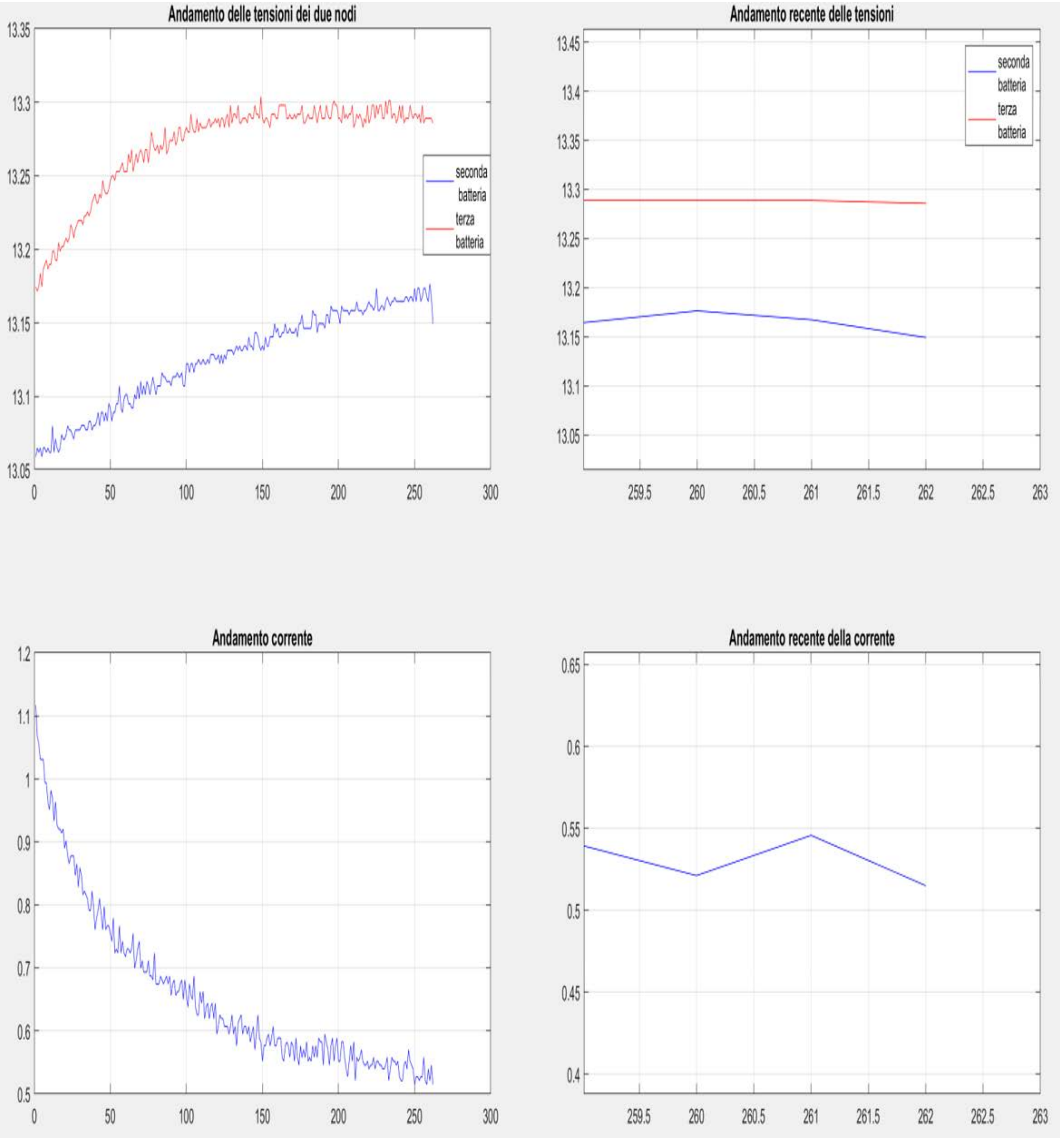


Figura 10.3.1: andamento di tensione e corrente della seconda e terza batteria della serie durante la fase di carica

In particolare, si sono monitorate la seconda e la terza batteria della serie complessiva; è evidente nel grafico in basso a sinistra di figura 10.3.1 come la corrente di carica, inizialmente di valore attorno ad 1.1A, decresca esponenzialmente nel tempo all'aumentare della tensione delle batterie monitorate. Tramite voltmetro si sono misurate le tensioni delle batterie della serie e si è notato, visibile anche nel grafico in alto a sinistra della medesima figura, uno squilibrio evidenziato in figura 10.3.2 tra le tensioni delle batterie monitorate.

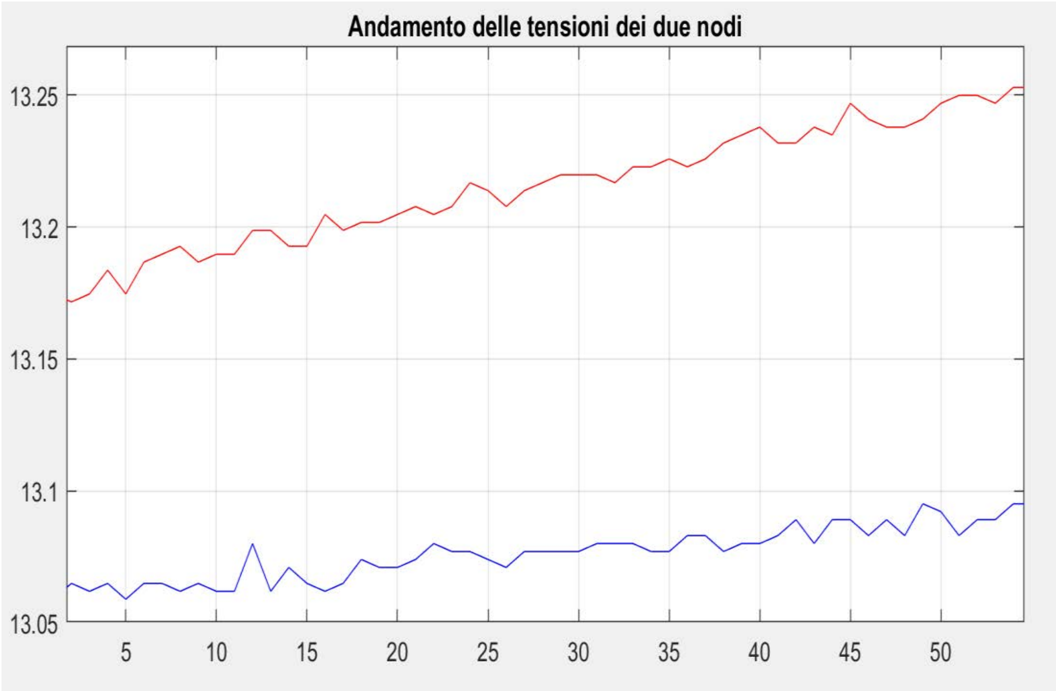


Figura 10.3.2: squilibrio fra le tensioni delle due batterie monitorate

Ciò è dovuto probabilmente al fatto che in parallelo ad ogni batteria monitorata vi è il circuito di acquisizione della tensione con relativa rete di comunicazione dei dati sul CAN bus; l'intero circuito costituisce un carico, seppur minimo, per l'accumulatore monitorato che lo alimenta e ne va ad influenzare quindi la tensione in fase di carica. A supporto di questa considerazione le misure con il voltmetro effettuate durante l'esperimento evidenziano come la tensione delle due batterie non monitorate risulti maggiore rispetto alla tensione degli accumulatori monitorati. A riprova del fenomeno appena descritto si è effettuato nuovamente il test di carica monitorando in particolare la seconda e la quarta batteria della serie, ottenendo l'andamento rappresentato in figura 10.3.3.

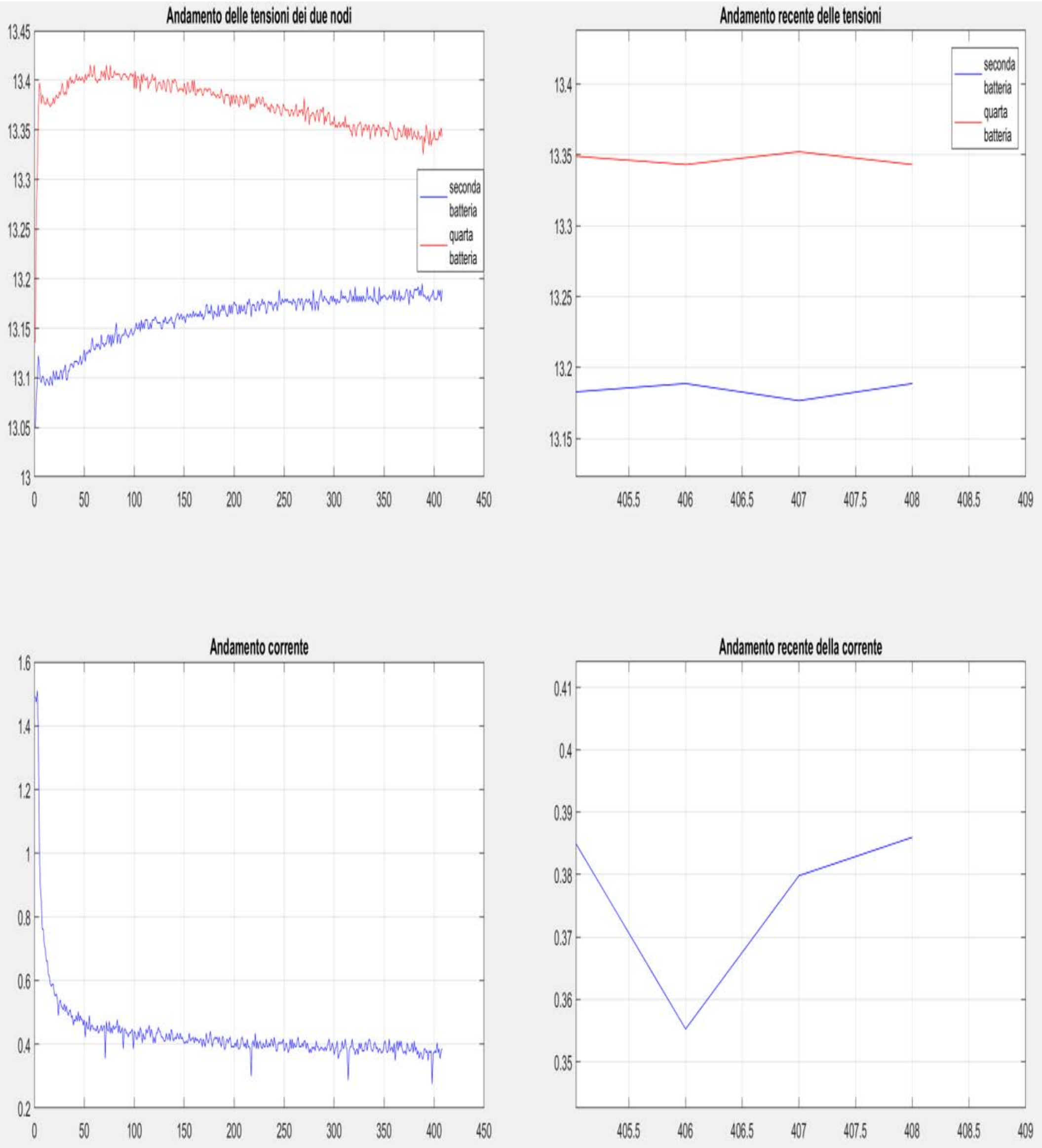


Figura 10.3.3: andamento di tensione e corrente della seconda e quarta batteria della serie durante la fase di carica

Si può notare dal grafico in alto sinistra di figura 10.3.3 come la tensione della quarta batteria nel precedente test non monitorata sia sensibilmente maggiore di quella della seconda batteria, monitorata anche nel precedente test, ciò a conferma della lieve influenza del circuito di acquisizione e comunicazione della tensione sulla carica degli accumulatori monitorati. Inoltre, è visibile sempre nel medesimo grafico come la tensione della quarta batteria (andamento rosso in figura 10.3.3) inizialmente continui a crescere per poi cominciare a diminuire; questo può essere dovuto al fatto che l'accumulatore ha raggiunto la fase di fine carica, come si può constatare dalla piccola corrente di carica fornita dal caricabatterie, e quindi perda una tensione minima rispetto a quella raggiunta in precedenza in piena fase di carica. In figura 10.3.4 si può invece notare come la corrente di carica decresca molto rapidamente nella fase iniziale dell'acquisizione: tale diminuzione marcata è dovuta molto probabilmente alla raggiunta carica delle batterie della serie (ipotesi confermata dalla diminuzione della tensione della quarta batteria).

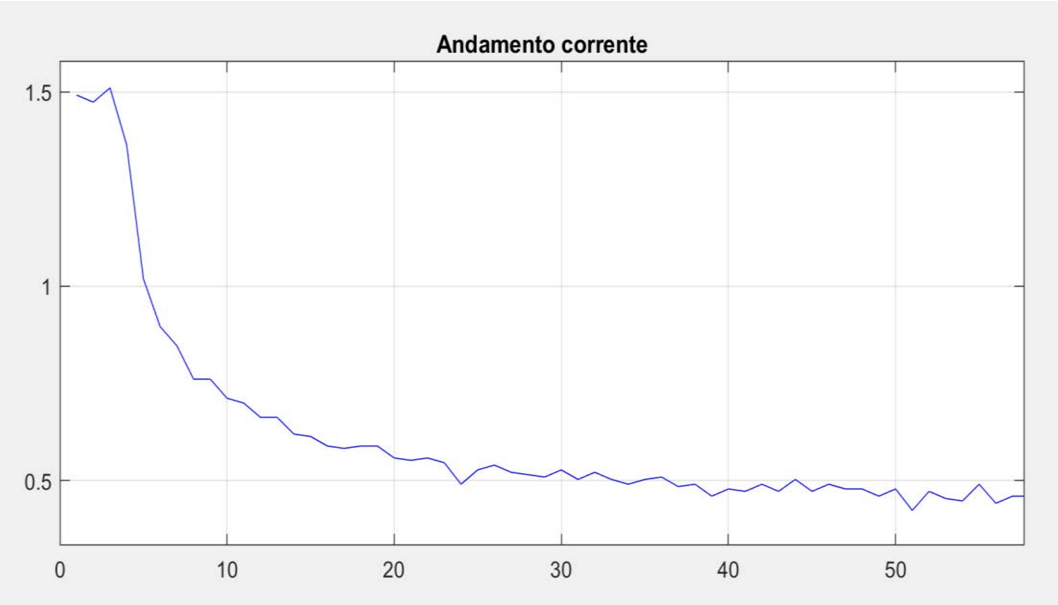


Figura 10.3.4: diminuzione marcata della corrente di carica

Le piccole fluttuazioni sulle grandezze monitorate visibili nei grafici sono dovute probabilmente alla corrente fornita dal caricabatterie; piccole variazioni istantanee internamente al dispositivo infatti vengono rilevate ed acquisite dal sistema di acquisizione e comunicazione, e poi restituite visivamente da Matlab. A tal proposito potrebbero avere una certa influenza anche i circuiti e i collegamenti fisici realizzati dall'utente, sensibili ad eventuali spostamenti spontanei delle connessioni.

Il secondo test realizzato riguarda invece la scarica delle batterie e dunque si sostituisce il caricabatterie con un reostato variabile, regolato ad inizio esperimento per costituire una resistenza di valore 40Ω così da limitare la corrente di scarica. I risultati ottenuti sono visibili in figura 10.3.5; si può vedere come nella fase iniziale del processo di scarica la tensione delle due batterie monitorate diminuisca marcatamente per poi assestarsi su un valore costante. Nella realtà la tensione degli accumulatori continua a diminuire però con derivata molto minore, quasi impercettibile nel breve periodo. Parallelamente la corrente di scarica nella prima metà del test (primi 500 campioni) rimane pressoché invariata intorno ai -0.6A; il segno negativo è dovuto all'inversione del verso della corrente in entrata alla sonda LEM rispetto al test di carica delle batterie effettuato precedentemente. Per una maggiore chiarezza visiva in figura 10.3.6 è rappresentata la fase iniziale della scarica, dove la resistenza di carico del reostato viene mantenuta costante pari a 40Ω.

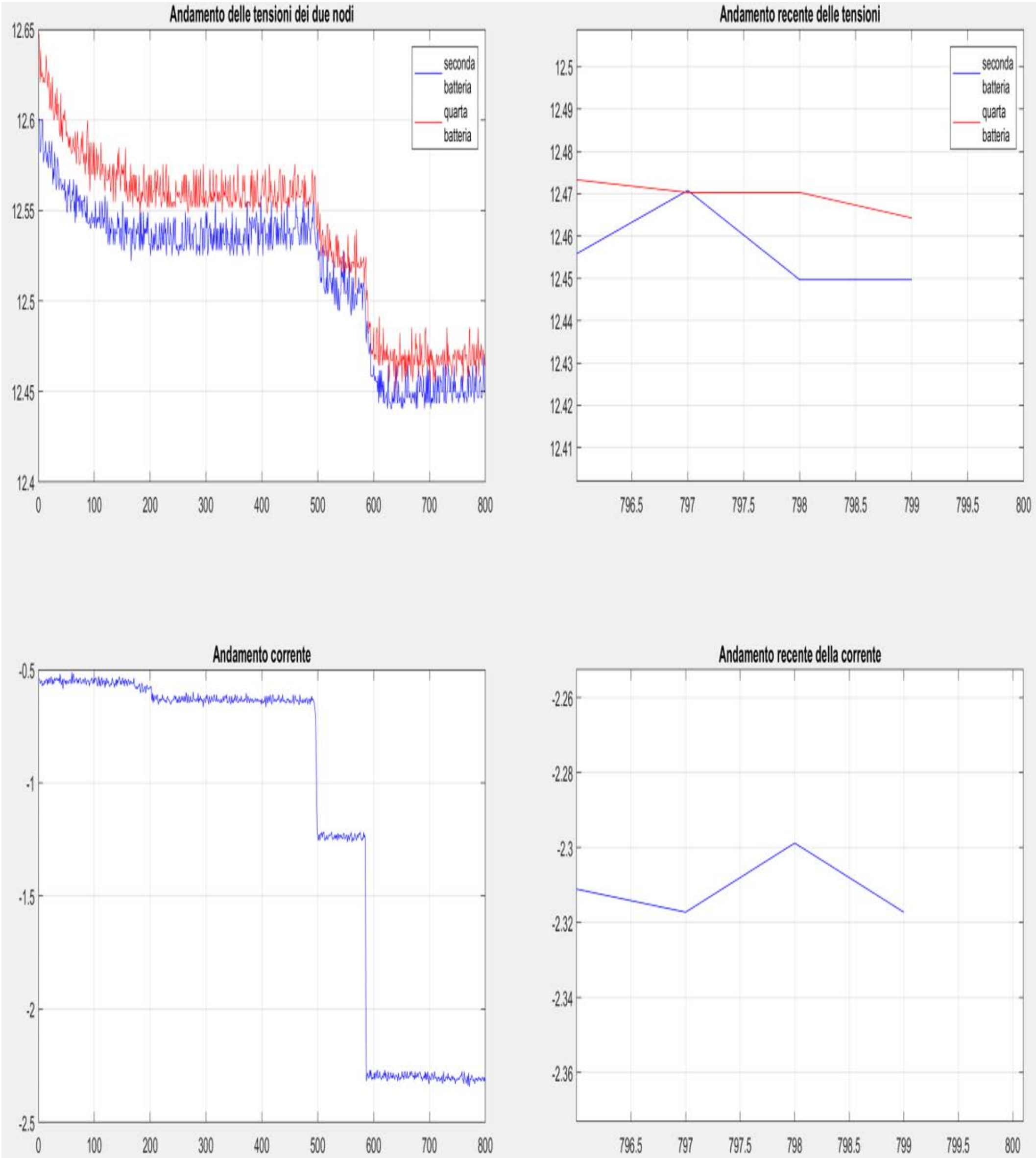


Figura 10.3.5: andamento di tensione e corrente della seconda e quarta batteria della serie durante la fase di scarica su resistenza di carico variabile

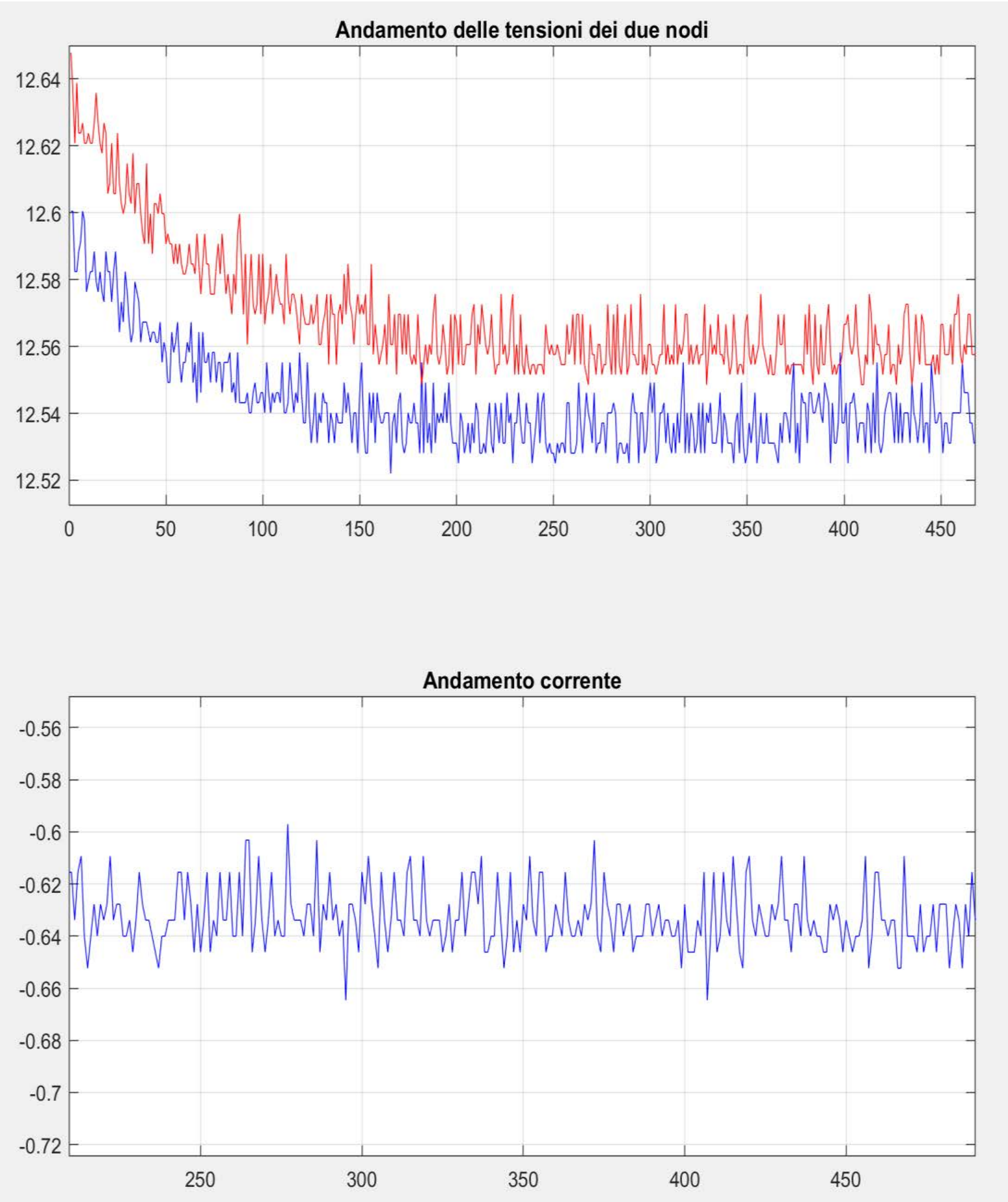


Figura 10.3.6: andamento di corrente e tensioni delle batterie nella fase iniziale della scarica

Per ulteriore scrupolo la resistenza del reostato viene modificata in due momenti durante il test: il primo all'incirca dopo aver acquisito 500 campioni e il secondo poco prima dei 600. Si possono notare più nel dettaglio i due momenti di variazione della resistenza di carico in figura 10.3.7 cui corrisponde una leggera diminuzione della tensione della batteria e un più marcato aumento in valore assoluto della corrente di scarica, in accordo con le aspettative teoriche.

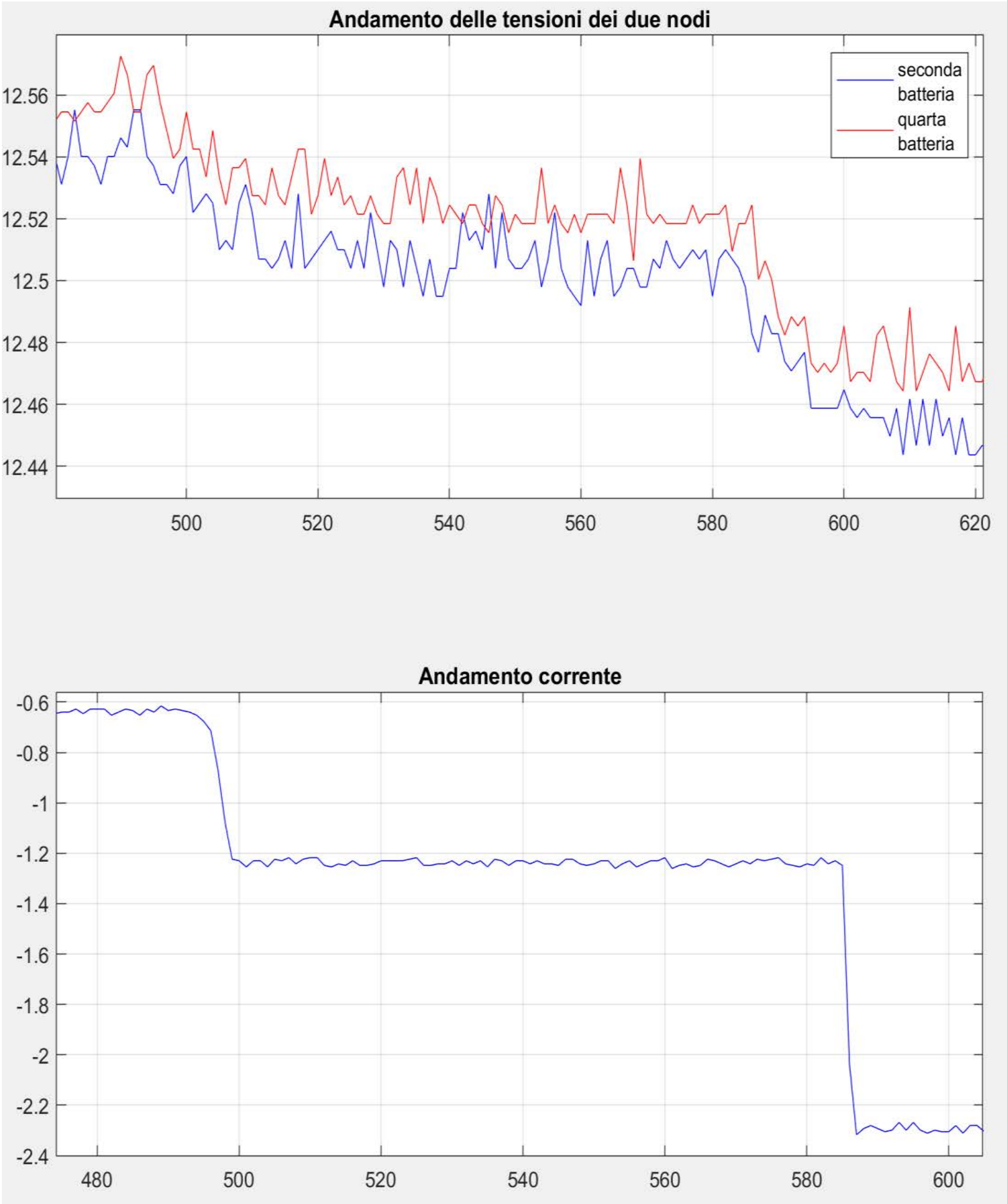


Figura 10.3.7: andamento di corrente e tensioni delle batterie in corrispondenza di due aumenti della resistenza di carico

Come già spiegato in precedenza, le piccole fluttuazioni che contraddistinguono gli andamenti acquisiti di tensione e corrente dipendono sostanzialmente dalla circuiteria hardware; molti collegamenti effettuati per condurre il test sono infatti sensibili a piccoli spostamenti spontanei e ciò va ad inficiare le misure. È da tenere in considerazione tuttavia che tali perturbazioni sono talmente piccole (~1‰) da non condizionare in alcun modo le misure stesse.

L'esito positivo di questo ultimo esperimento conferma quanto già anticipato dai due test precedenti illustrati nei capitoli 10.1 e 10.2: la rete di comunicazione di un sistema di monitoraggio distribuito della carica e scarica di una batteria progettata e realizzata in questa trattazione è perfettamente applicabile e funzionante, garantendo un'acquisizione corretta e in real time, nonché elevata precisione, dei valori di tensione e corrente delle batterie monitorate.

11. Sistema di monitoraggio distribuito basato su una rete di comunicazione radio

La rete di comunicazione realizzata ed illustrata fino a questo punto della trattazione, seppur funzionante e facilmente modificabile in quanto modulare, basa il suo funzionamento sul CAN bus e presenta dunque degli svantaggi nell'ottica di un suo utilizzo a bordo di un veicolo elettrico dove la gestione dello spazio e dei collegamenti elettrici è di fondamentale importanza. Ogni nodo Slave della rete di comunicazione basata sul CAN bus infatti è costituito da diversi componenti elettronici e di conseguenza sorge innanzitutto un problema di cablaggio; il numero di cavi da utilizzare per implementare un nodo della rete infatti è elevato e ciò aumenta l'occupazione in termini di spazio del circuito complessivo. Inoltre, la comunicazione dei dati fra i diversi nodi della rete CAN bus richiede necessariamente il collegamento diretto dei nodi mediante un mezzo fisico comune. Pensando quindi all'utilizzo del sistema sviluppato su un pacco batterie a bordo di un veicolo elettrico è evidente come le considerazioni appena fatte non semplifichino né l'installazione del sistema stesso né la sua gestione e manutenzione ed evidenziano un'occupazione in termini di volume non indifferente.

Per ovviare a questi inconvenienti si è pensato di semplificare notevolmente la struttura del sistema di monitoraggio distribuito sostituendo la rete di comunicazione basata sul CAN bus con un sistema di comunicazione radio. In questo modo infatti viene ridotto il numero di componenti elettronici necessari, diminuendo fortemente la complessità del cablaggio da realizzare e il volume necessario, e non è più necessario collegare direttamente tra loro i diversi nodi della rete poiché il mezzo fisico di trasmissione è l'aria stessa. A tal proposito si è deciso di impiegare il componente nRF24L01 di Nordic Semiconductor, programmabile con un microcontrollore mediante SPI (Serial Peripheral Interface) ovvero un protocollo di comunicazione seriale sincrona non approfondito in questa trattazione; si procede ad introdurre ed illustrare più nel dettaglio il ricetrasmettitore.

11.1. Componente nRF24L01 di Nordic Semiconductor

Il componente nRF24L01 di Nordic Semicoductor è un ricetrasmettitore a singolo chip operante a 2.4GHz progettato per applicazioni wireless a bassissima potenza e per il funzionamento nella banda di frequenza mondiale ISM a 2.400-2.4835GHz; se ne può vedere una fotografia in figura 11.1.1. [16].

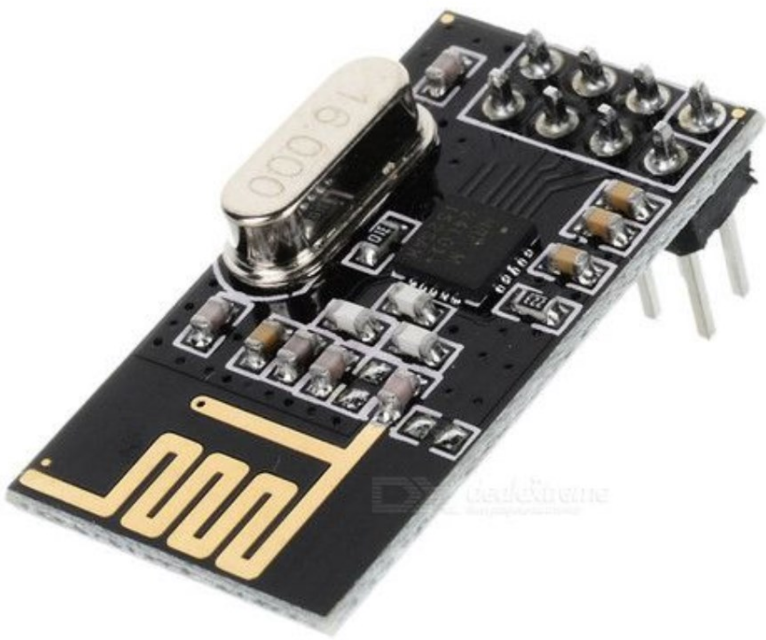


Figura 11.1.1: componente nRF24L01 di Nordic Semiconductor

Per implementare un sistema radio basato su questo componente si necessita di un microcontrollore e pochi componenti elettronici passivi, riducendo notevolmente la complessità del circuito hardware da realizzare. Come accennato precedentemente, il chip nRF24L01 viene configurato e gestito attraverso un'interfaccia seriale periferica (SPI) che permette di accedere alla mappa dei registri del componente, la quale contiene tutti i registri di configurazione ed è accessibile in tutte le modalità di funzionamento del chip. Il ricetrasmettitore incorpora il protocollo Enhanced ShockBurst™ in banda base basato sulla comunicazione a pacchetti: i registri FIFO (First In First Out) interni al componente nRF24L01 assicurano un flusso di dati fluido tra il front end della radio e il microcontrollore del sistema. Il front end stesso della radio utilizza la modulazione GFSK (Gaussian Frequency Shift Keying) e i parametri che lo caratterizzano sono configurabili dall'utente; tra i principali si può personalizzare il canale di frequenza della comunicazione, la potenza di uscita in trasmissione e la velocità dei dati dell'aria, configurabile fino a 2Mbps. Il diagramma a blocchi del componente è riportato in figura 11.1.2.

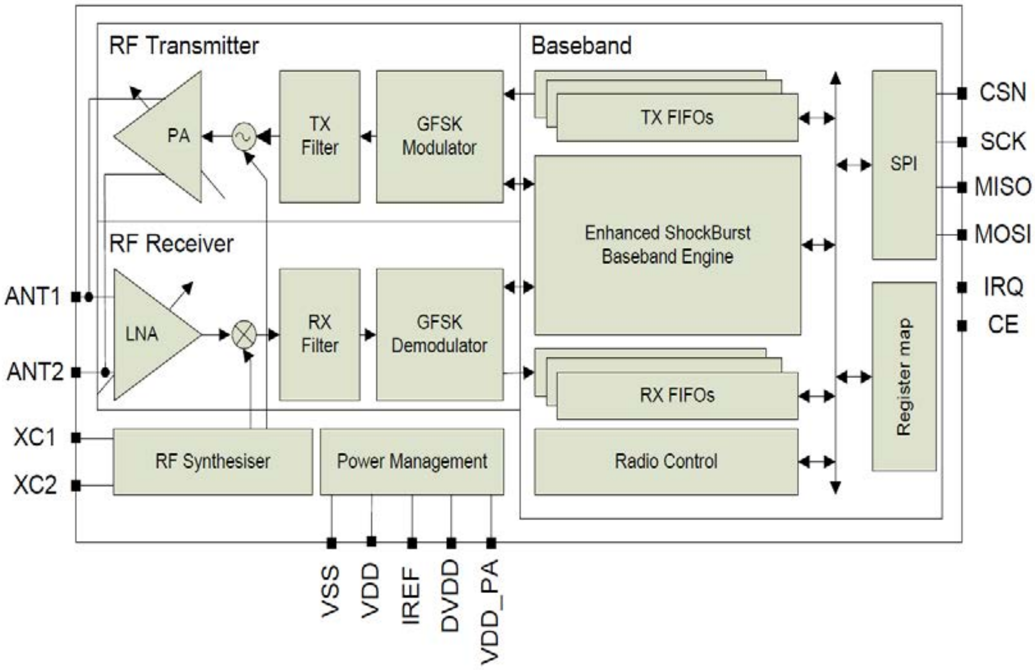


Figura 11.1.2: diagramma a blocchi funzionali del componente

Si può notare in figura 11.1.2 come i pin che il chip nRF24L01 mette a disposizione siano di diverso tipo: in particolare, i pin CSN, SCK, MISO e MOSI devono essere collegati al microcontrollore tramite SPI mentre i pin V_{DD}, V_{SS}, IRQ e CE vanno opportunamente collegati a seconda della funzione che svolgono. Tra i pin SPI quello CSN costituisce il Chip Select che va connesso al pin digitale SS (Slave Select) del componente SPI Master del microcontrollore; comandando il pin SS e quindi il pin CSN si può abilitare la comunicazione SPI tra il ricetrasmettitore e il MC (MicroController). Il pin SCK trasmette il clock che implementa e regola la comunicazione SPI; il pin MOSI (Master Output Slave Input) trasmette i dati inviati dal componente SPI Master del MC in input al ricetrasmettitore mentre il pin digitale MISO (Master Input Slave Output) gestisce la comunicazione nel senso contrario, ovvero trasmette dati dal chip al MC. Gli altri pin d’interesse del componente nRF24L01 sono quello della tensione d’alimentazione V_{DD} (compresa tra 1.9V e 3.6V), quello della connessione di terra V_{SS}, il pin CE (Chip Enable) per attivare le modalità di ricezione o trasmissione del ricetrasmettitore ed infine il pin IRQ (Interrupt Request) che è un pin mascherabile di interrupt attivo basso.

Il chip nRF24L01 può operare in diverse modalità, rappresentate schematicamente in figura 11.1.3: in particolare, il suo stato non è definito fino a quando sul pin V_{DD} non ci siano 1,9V o più. Quando ciò accade il componente entra nello stato di reset 'Power On' che permane fino a quando non entra in modalità 'Power Down'. Anche quando il chip nRF24L01 entra in quest'ultima modalità, il microcontrollore può comunque configurare e controllare il chip attraverso l'SPI e il pin CE di abilitazione del chip stesso. Le quattro modalità funzionali in cui il componente può accedere sono 'Power Down', 'Standby', 'RX' e 'TX'; differiscono tra di loro a seconda dei particolari settaggi dei registri del componente che si possono impostare mediante l'SPI. Non è tuttavia d'interesse in questa trattazione approfondire specificatamente ogni singola modalità o le diverse transizioni di stato che si possono operare; ai fini del progetto in esame è sufficiente comprendere come poter programmare e controllare il chip nRF24L01 tramite il kit CY8CKIT-059 affinché svolga i compiti richiesti.

In figura 11.1.3 si possono notare delle indicazioni per quanto concerne la tempistica che il chip nRF24L01deve osservare nel passaggio tra stati di funzionamento, che costituisce di fatto una delle accortezze fondamentali da prestare in fase di configurazione per farlo lavorare correttamente. Nella parte alta per esempio si può vedere come, qualora il chip venga alimentato sul pin V_{DD} ad una tensione adatta, impieghi 10.3ms per passare da uno stato indefinito a quello di 'Power Down'. Si deve prestare particolare attenzione a questi intervalli temporali che contraddistinguono il funzionamento del chip nRF24L01 soprattutto in fase di programmazione del microcontrollore, che costituisce assieme al protocollo SPI il mezzo per configurare il ricetrasmettitore; ulteriori considerazioni per quanto riguarda tale argomento vengono riprese nel seguito della trattazione.

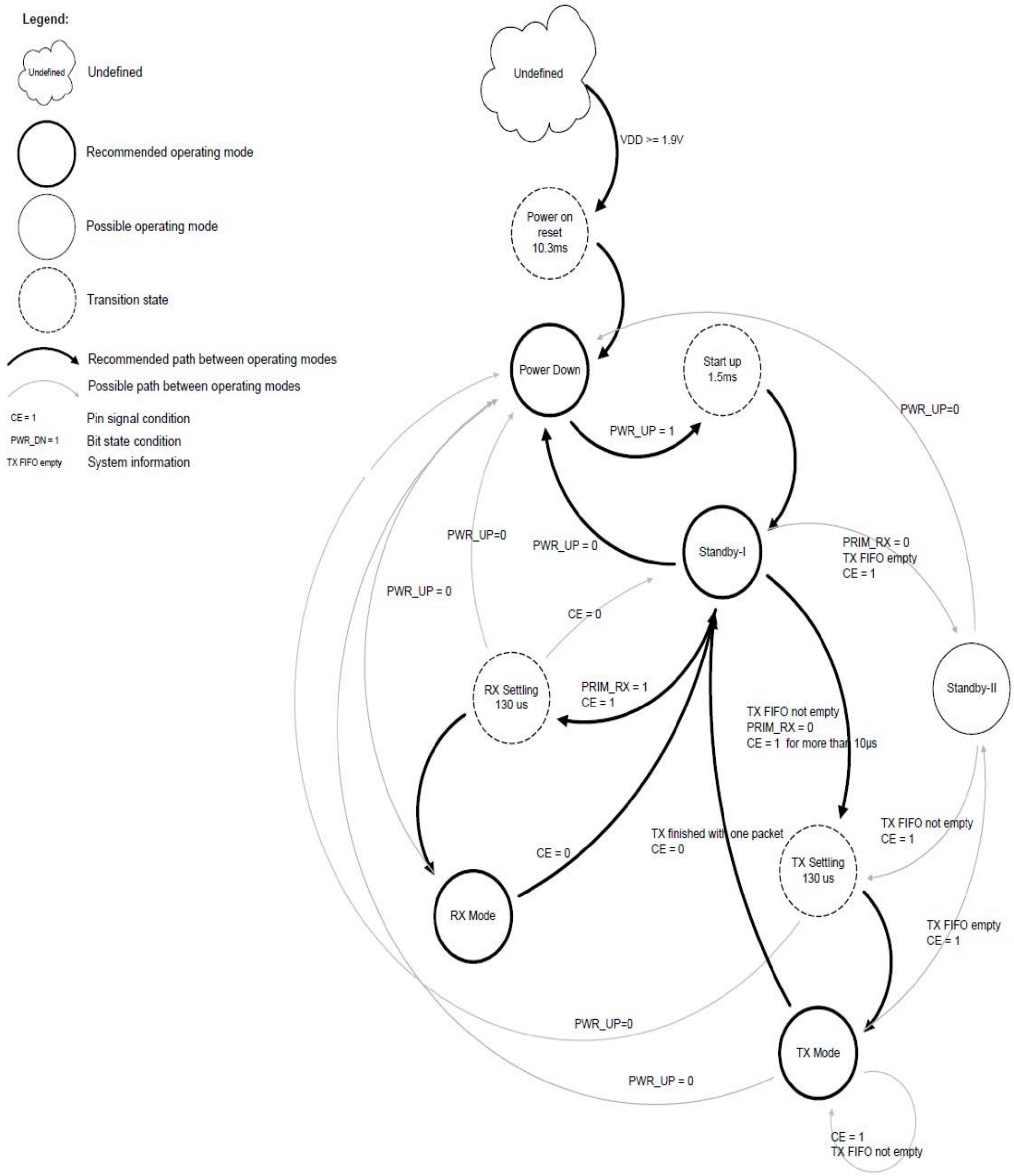


Figura 11.1.3: diagramma degli stati del componente e delle loro transizioni

Come introdotto in precedenza, il nRF24L01 implementa il protocollo Enhanced ShockBurst™ che di fatto costituisce un data link layer basato sui pacchetti: garantisce un assemblaggio automatico della struttura e temporizzazione di ogni pacchetto dati e fornisce la possibilità di utilizzare le funzioni di AutoAcknowledgment (AA) e ritrasmissione automatica. Durante la trasmissione, Shockburst™ assembla e sincronizza i bit nel pacchetto dati nel chip per la trasmissione; durante la ricezione invece ricerca costantemente un indirizzo valido nel segnale demodulato e quando lo trova elabora il resto del pacchetto e lo convalida controllando il CRC (Cyclic Redundancy Check). Se il pacchetto è

valido il carico utile è spostato nel RX FIFO e pronto per essere letto dal microcontrollore mediante SPI.
Un aspetto fondamentale da studiare ed approfondire è l'interfaccia dati e di controllo che permette di accedere a tutte le funzionalità di nRF24L01 ed è costituita di fatto dai seguenti sei segnali digitali con altrettanti pin dedicati:

- IRQ, attivo basso e controllato da tre sorgenti di interrupt mascherabili
- CE, attivo alto e utilizzato per attivare il chip in modalità RX o TX
- CSN, SCK, MOSI, MISO (segnali SPI)

L'attivazione della mappa dei registri o dei data FIFO avviene mediante specifici comandi espressi con una parola lunga un byte inviati dal microcontrollore al chip nRF24L01 tramite SPI: ogni comando, inviato fisicamente al ricetrasmettitore tramite il pin MOSI, deve essere preceduto da una transizione alto-basso sul pin CSN affinché venga riconosciuto in quanto tale. Contemporaneamente alla trasmissione del comando il MC riceve il registro 'STATUS' sul pin digitale di input MISO, cosicché il microcontrollore può controllare costantemente lo stato del chip. In generale ogni parola di comando, costituita da un byte specifico a seconda della funzione che il comando svolge, deve essere seguita da un determinato numero di data bytes da trasmettere tramite SPI, ordinati a partire dal byte meno significante (LSByte, Less Significant Byte).

Si riporta nella tabella 11.1.1 alcuni tra i comandi utilizzati nel progetto in esame, assieme al loro corrispettivo binario e alle funzioni specifiche che svolgono.

Tabella 11.1.1: tabella dei comandi e delle funzioni associate

Command name	**Command name (binary)**	**Data bytes**	**Operation**
R_REGISTER	000A AAAA	Da 1 a 5, il primo è il LSByte	Comando di lettura dei registri, i 5 bit AAAAA costituiscono l'indirizzo specifico della mappa dei registri su cui applicare il comando

W_REGISTER	001A AAAA	Da 1 a 5, il primo è il LSByte	Comando di scrittura dei registri, i 5 bit AAAAA costituiscono l'indirizzo specifico della mappa dei registri su cui applicare il comando
R_RX_PAYLOAD	0110 0001	Da 1 a 32, il primo è il LSByte	Comando di lettura del carico utile nel registro RX FIFO; il carico utile viene cancellato dopo la lettura
W_TX_PAYLOAD	1010 0000	Da 1 a 32, il primo è il LSByte	Comando di scrittura del carico utile nel registro TX FIFO; l'operazione di scrittura inizia dal byte 0
FLUSH_TX	1110 0001	0	Pulizia del TX FIFO, utilizzato in modalità TX
FLUSH_RX	1110 0010	0	Pulizia del RX FIFO, utilizzato in modalità RX
ACTIVATE	0101 0000	1 (0x73)	Questo comando di scrittura seguito dal byte 0x73 attiva diverse funzionalità, tra cui W_TX_PAYLOAD_NOACK
W_TX_PAYLOAD_NOACK	1011 0000	Da 1 a 32, il primo è il LSByte	Comando per disabilitare l'AutoAcknowledgment nel pacchetto specifico; utilizzato solo in modalità TX

Come accennato in precedenza, è fondamentale tenere conto del preciso timing richiesto sia dal funzionamento del componente nRF24L01 che dalla comunicazione SPI al fine di inviare correttamente un comando dal microcontrollore al chip stesso; si riporta in figura 11.1.4 una rappresentazione temporale dell'operazione di lettura del microcontrollore tramite SPI.

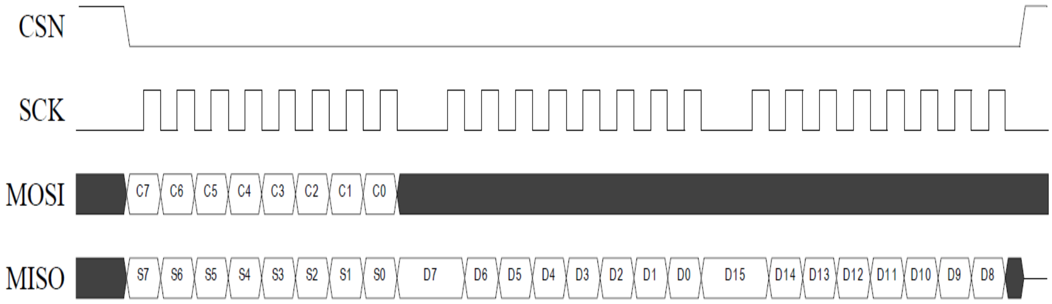


Figura 11.1.4: operazione di lettura tramite SPI

Si può notare come sul pin MOSI avvenga la trasmissione di 1 byte di dati relativi al comando specifico identificato dalla lettera maiuscola ‘C’; contemporaneamente sul pin MISO avviene la ricezione sia del byte relativo al registro ‘STATUS’ che dei byte di dati che si intende leggere. L’inizio dell’operazione è determinato da un fronte alto-basso sul pin CSN del chip controllato tramite il pin SS del componente SPI Master del microcontrollore; appena terminata la transizione parte la trasmissione del clock che scandisce e regola la comunicazione seriale. Si deve sempre tenere in considerazione infatti che il protocollo SPI implementa una comunicazione seriale sincrona, caratterizzata dalla necessità di condivisione del segnale di clock tra i dispositivi comunicanti. Il pin CSN (e di conseguenza il pin SS) deve essere mantenuto basso per tutto il tempo necessario al completamento dell’operazione. È fondamentale evidenziare inoltre che il tempo che intercorre tra l’inizio della transizione sul pin CSN e la trasmissione del clock sul pin SCK è pari a 10µs; questa è un’accortezza fondamentale a cui prestare molta attenzione in fase di sviluppo del software con cui programmare il microcontrollore. Qualora non vengano rispettate queste tempistiche, non si potrebbe implementare una comunicazione SPI funzionante e di conseguenza configurare e controllare correttamente il chip nRF24L01.

Analogamente in figura 11.1.5 si riporta una rappresentazione temporale dell’operazione di scrittura tramite SPI.

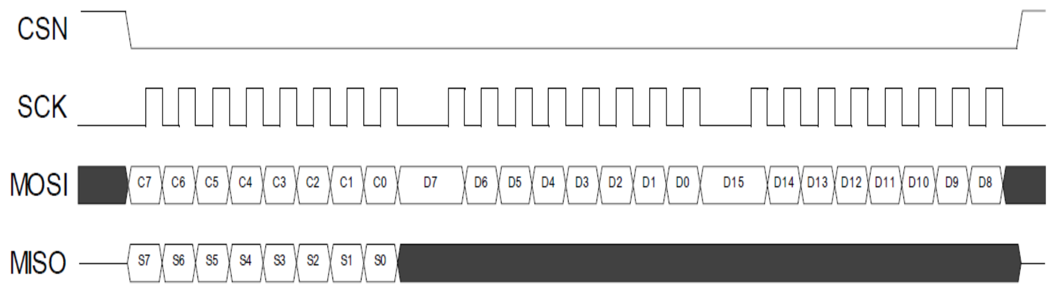


Figura 11.1.5: operazione di scrittura tramite SPI

L'operazione dal punto di vista del comportamento dei pin CSN e SCK è uguale a quella di lettura; in questa situazione tuttavia sul pin MOSI viene trasmesso il byte associato alla parola di comando da eseguire seguito dai data bytes necessari per completare il comando. Contemporaneamente sul pin MISO viene trasmesso dal chip nRF24L01 al microcontrollore il byte relativo allo stato del registro 'STATUS'.

È di fondamentale importanza al fine di configurare e controllare correttamente il chip nRF24L01 comprendere il principio di funzionamento del pin IRQ; tale pin costituisce un segnale di interrupt attivo basso comandato dal chip stesso. Viene attivato quando i bit TX_DS, RX_DR e/o MAX_RT del registro 'STATUS' sono portati allo stato alto dalla macchina di stato interna al componente. Nella fase di configurazione del chip infatti, mediante un apposito comando di scrittura del registro 'STATUS', si può impostare il ricetrasmettitore affinché il pin IRQ si attivi qualora avvenga un determinato evento, tra cui l'invio di un pacchetto (TX_DS, TX_DataSent), la ricezione di un pacchetto (RX_DR, RX_DataReceived) o l'aver raggiunto il limite massimo di ritrasmissioni possibili di un messaggio (MAX_RT, MAX_ReTransmission). Si capisce immediatamente quanto questa funzionalità possa essere d'aiuto al fine di monitorare lo stato del chip e la corretta esecuzione della routine che si vuole implementare. Per resettare lo stato del pin IRQ e quindi riportarlo allo stato alto il microcontrollore deve impostare ad 1 il bit del registro 'STATUS' corrispondente alla sorgente di interrupt mediante apposito comando SPI.

Una volta definiti i comandi utili, si prosegue illustrando più nello specifico la mappa dei registri di nRF24L01 che permette di configurarlo e controllarlo tramite SPI utilizzando gli opportuni comandi di lettura e scrittura precedentemente descritti. Dato che i registri sono numerosi e sono comunque consultabili nel datasheet del componente, si riporta a titolo esemplificativo in figura 11.1.6 solamente qualcuno fra quelli utilizzati nella presente trattazione con relativi indirizzi e significato dei singoli bit che li compongono. Si può notare come ogni registro sia associato ad uno specifico indirizzo esadecimale nella mappa dei registri e come ogni bit o gruppo di bit che lo definisce debba essere configurato appropriatamente a seconda delle funzioni che il componente nRF24L01 deve svolgere.

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
00	CONFIG				Configuration Register
	Reserved	7	0	R/W	Only '0' allowed
	MASK_RX_DR	6	0	R/W	Mask interrupt caused by RX_DR 1: Interrupt not reflected on the IRQ pin 0: Reflect RX_DR as active low interrupt on the IRQ pin
	MASK_TX_DS	5	0	R/W	Mask interrupt caused by TX_DS 1: Interrupt not reflected on the IRQ pin 0: Reflect TX_DS as active low interrupt on the IRQ pin
	MASK_MAX_RT	4	0	R/W	Mask interrupt caused by MAX_RT 1: Interrupt not reflected on the IRQ pin 0: Reflect MAX_RT as active low interrupt on the IRQ pin
	EN_CRC	3	1	R/W	Enable CRC. Forced high if one of the bits in the EN_AA is high
	CRCO	2	0	R/W	CRC encoding scheme '0' - 1 byte '1' – 2 bytes
	PWR_UP	1	0	R/W	1: POWER UP, 0:POWER DOWN
	PRIM_RX	0	0	R/W	RX/TX control 1: PRX, 0: PTX
07	STATUS				Status Register (In parallel to the SPI command word applied on the MOSI pin, the STATUS register is shifted serially out on the MISO pin)
	Reserved	7	0	R/W	Only '0' allowed
	RX_DR	6	0	R/W	Data Ready RX FIFO interrupt. Asserted when new data arrives RX FIFO[b]. Write 1 to clear bit.
	TX_DS	5	0	R/W	Data Sent TX FIFO interrupt. Asserted when packet transmitted on TX. If AUTO_ACK is activated, this bit is set high only when ACK is received. Write 1 to clear bit.
	MAX_RT	4	0	R/W	Maximum number of TX retransmits interrupt Write 1 to clear bit. If MAX_RT is asserted it must be cleared to enable further communication.
	RX_P_NO	3:1	111	R	Data pipe number for the payload available for reading from RX_FIFO 000-101: Data Pipe Number 110: Not Used 111: RX FIFO Empty
	TX_FULL	0	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.

Figura 11.1.6: mappa dei registri 'CONFIG' e 'STATUS' del nRF24L01

Facendo riferimento al registro ad un byte di configurazione 'CONFIG', visibile nella parte alta di figura 11.1.6, si può vedere come i bit dal 6 al 4 (ordinati a partire dal MSBit al LSBit) siano di tipologia R/W, ovvero accessibili sia in lettura (Read) che scrittura (Write), e costituiscano di fatto delle sorgenti di interrupt mascherabili riflessi sul pin IRQ attivo basso impostabili a piacere dall'utente. In particolare, impostando a 0 il pin 6, 5 e 4 si fa in modo che il pin IRQ rifletta rispettivamente l'interrupt relativo alla ricezione di dati (RX_DR), alla trasmissione di dati (TX_DS) o al raggiungimento del massimo numero di ritrasmissioni di un pacchetto (MAX_RT). Viceversa, impostando uno di questi bit al valore 1 se ne disattiva il legame tra l'evento di interrupt e il pin IRQ. I bit 3 e 2 invece sono configurabili per attivare o meno il metodo di verifica dei pacchetti CRC, decidendone in caso di utilizzo la lunghezza in byte dello schema di codifica. Il bit 1 invece può essere configurato per gestire lo stato di funzionamento del chip nRF24L01: se su tale bit viene scritto il valore 1 allora il componente si accende ('POWER UP') mentre se impostato a 0 si spegne ('POWER DOWN'). Infine, il bit 0 (PRIM_RX) definisce la modalità operativa del componente, controllando il suo funzionamento come ricevitore (valore 1) o trasmettitore (valore 0). È altresì interessante notare come venga fornita sulla tabella della mappa dei registri, nella colonna 'Reset Value', l'indicazione relativa al valore che assumono i bit di ogni singolo registro qualora il componente venga avviato senza configurarle manualmente tale registro. Si deve tenere a mente quindi in fase di configurazione del chip nRF24L01 di controllare non solo la corretta scrittura dei registri d'interesse ma anche di verificare che quelli a cui non si accede manualmente assumano dei valori di reset e quindi attivino delle funzionalità specifiche consone allo scopo progettuale.

Per comprendere meglio il processo di invio di un comando associato ad uno specifico registro tramite SPI si procede a riportare un ritaglio di codice utile per configurare il registro 'CONFIG'.

```
/*  ======================================== */
CyDelayUs(10);    // aspetto 10 microsecondi
SS_Write(1);      // imposto al valore 1 il pin CSN (SS)
CyDelayUs(10);    // aspetto 10 microsecondi
```

```
// Settaggio in scrittura del registro CONFIG
spiData[0] = W_REGISTER | CONFIG;      // 0010 0000
spiData[1] = nRF_CONFIG_RX_INIT;       // 0011 1011
SS_Write(0);     // imposto al valore 0 il pin CSN (SS)
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
```

Nelle prime tre righe dello script si imposta al valore 1 il pin SS (CSN); si posizionano poi byte per byte all'interno di un vettore prima il comando da eseguire e poi gli eventuali data bytes seguenti il comando stesso da inviare tramite SPI. Nel caso particolare appena riportato l'intenzione dell'utente è quella di configurare in scrittura il registro 'CONFIG'; facendo riferimento alla tabella 11.1.1 si può vedere come il comando specifico di scrittura 'W_REGISTER' sia espresso tramite il byte 0b001AAAAA dove l'espressione binaria di 5 bit AAAAAA deve essere sostituita con l'indirizzo dello specifico registro cui accedere. Il registro 'CONFIG' come si può vedere in figura 11.1.6 è contraddistinto dall'indirizzo 0x00 e dunque si può facilmente tradurre il comando di scrittura su tale registro sfruttando l'operatore OR logico ('|') tra 0b00100000 ('W_REGISTER') e 0b00000000 ('indirizzo del registro CONFIG'). Dato che il registro 'CONFIG' è composto da un singolo byte, oltre al comando si deve inviare tramite SPI al chip nRF24L01 un secondo byte contenente i valori bit per bit con cui si desidera impostare il registro stesso: nel caso particolare appena riportato tale valore binario 'nRF_CONFIG_RX_INIT' (0b00111011) associato al comando di scrittura del registro 'CONFIG' fa sì che l'interrupt relativo alla ricezione di un pacchetto dati (RX_DR) si rifletta sul pin IRQ attivo basso, si abilita una verifica CRC ad un byte e si accende il ricetrasmettitore in modalità PRX (ricezione). Va notato che prima di inviare questo specifico comando tramite la funzione 'SPIM_PutArray' si deve abbassare il pin SS (CSN) ottenendo su di esso un fronte alto-basso ed aspettare 10 microsecondi, in accordo con le indicazioni temporali necessarie per eseguire correttamente un'operazione di scrittura tramite SPI. La procedura appena descritta deve essere seguita per eseguire un comando qualsiasi operante su un generico registro, sostituendo opportunamente i byte inviati a seconda delle intenzioni dell'utente.

Per la presente trattazione è interessante approfondire anche la funzionalità Multiceiver che il chip nRF24L01 mette a disposizione, vista la natura modulare del sistema di monitoraggio ideato; se ne può vedere una rappresentazione schematica in figura 11.1.7.

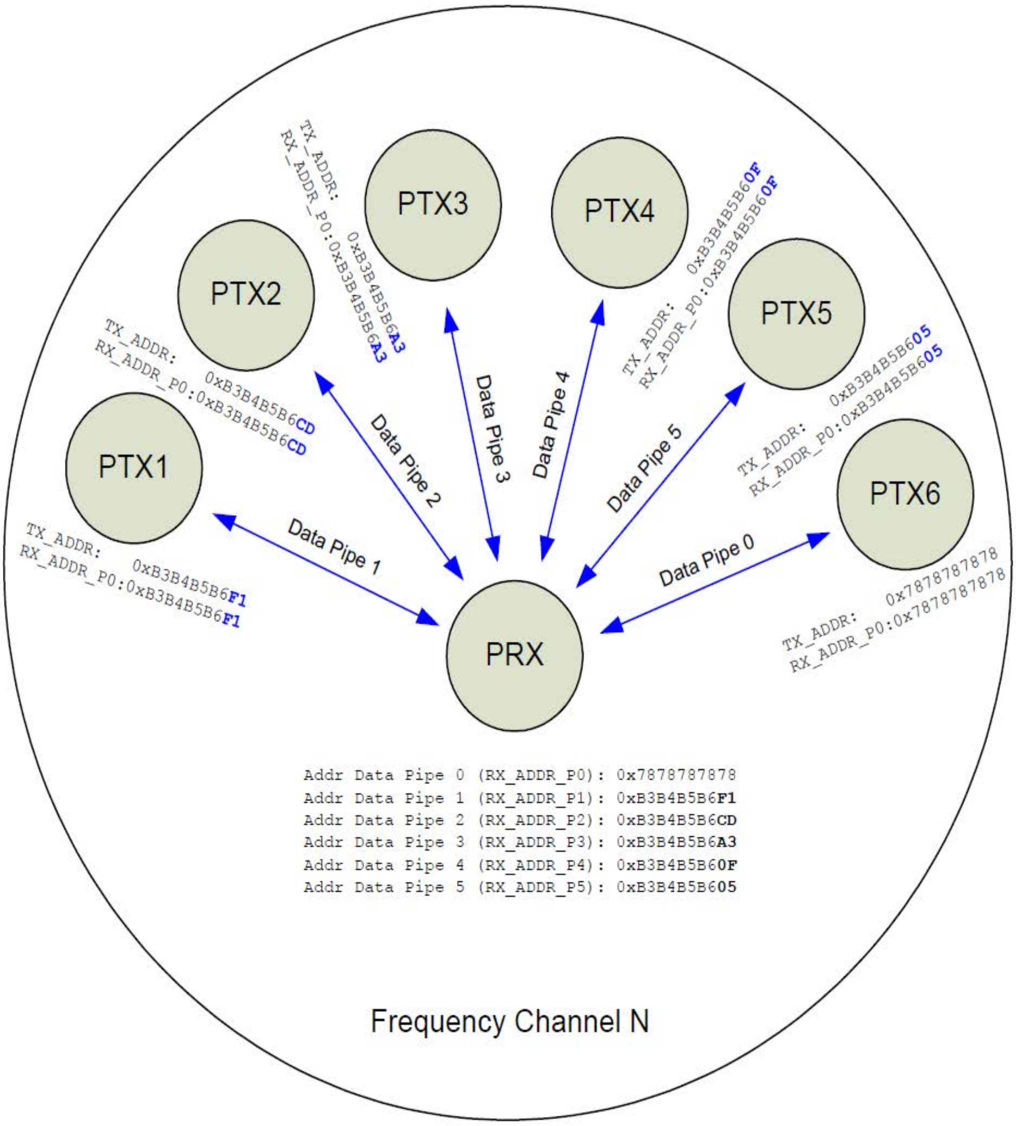


Figura 11.1.7: esempio di indirizzamento dei data pipe in modalità multiceiver

In modalità ricezione RX il chip dispone infatti di 6 canali logici di dati in parallelo sulla stessa frequenza RF distinti da indirizzi univoci; il chip nRF24L01 configurato come PRX (ricevitore primario) può ricevere quindi dati indirizzati a sei diversi data pipe sfruttando un unico canale di frequenza, fornendo all’utente la possibilità di configurare ogni data pipe per svolgere routine differenti le une dalle altre. I diversi canali dati devono però condividere lo stesso protocollo di verifica CRC, la stessa lunghezza in byte degli indirizzi di ricezione, lo stesso canale di frequenza e la stessa velocità di trasmissione dei dati in aria. È importante sottolineare come un canale dati non possa comunque ricevere dei messaggi fintanto che non sia ricevuto un pacchetto completo da un altro data pipe che abbia riconosciuto il proprio indirizzo di ricezione; qualora dunque diversi chip configurati come PTX (trasmettitori) stiano inviando simultaneamente dei pacchetti allo stesso dispositivo ricevente è necessario sfruttare la funzione di ritrasmissione automatica di un pacchetto che il chip nRF24L01 mette a disposizione per garantire di non perdere alcun pacchetto in ricezione.

Illustrato nel dettaglio il funzionamento del ricetrasmettitore nRF24L01 di Nordic Semiconductor, si procede ad illustrarne nello specifico l'utilizzo allo scopo di implementare una rete di comunicazione radio di un sistema di monitoraggio distribuito della carica e scarica di una batteria.

11.2. Progettazione e realizzazione del circuito hardware

Il circuito fisico da realizzare può essere visto come una modifica di quello precedentemente illustrato nel capitolo 8, tenendo presente che si sono sostituiti i ricetrasmettitori CAN bus isolati ISO1050DWR con diversi chip nRF24L01 di Nordic Semiconductor. Rimuovendo il mezzo fisico di comunicazione (CAN bus) non è più necessario nemmeno l'utilizzo dei convertitori DC-DC SPA01A-05 regolati a 5V, riducendo notevolmente la complessità del circuito e del cablaggio da realizzare. Permane invece la necessità di utilizzare i regolatori di tensione LM7805CT/NOPB per alimentare il kit CY8CKIT-059 tramite la batteria monitorata e si deve inoltre introdurre il componente AP1117 TO220-3 di Diodes Incorporated, un regolatore di tensione con tensione d'uscita fissa e pari a 3.3V indispensabile per alimentare correttamente il chip nRF24L01 secondo specifiche [17]. Il componente AP1117 non viene approfondito nella presente trattazione; l'unico dettaglio fondamentale da riportare è la tensione in input V_{IN} accettata dal componente per garantirne un corretto funzionamento, compresa tra -0.3V e +18V.

Per qualsiasi altro approfondimento e dettaglio specifico si rimanda il lettore al datasheet del componente.

Ne consegue che il circuito fisico da realizzare per un nodo Slave della rete di comunicazione, incaricato di acquisire la tensione della batteria monitorata e inviarla tramite comunicazione radio al nodo Master, è quello schematicamente riportato in figura 11.2.1.

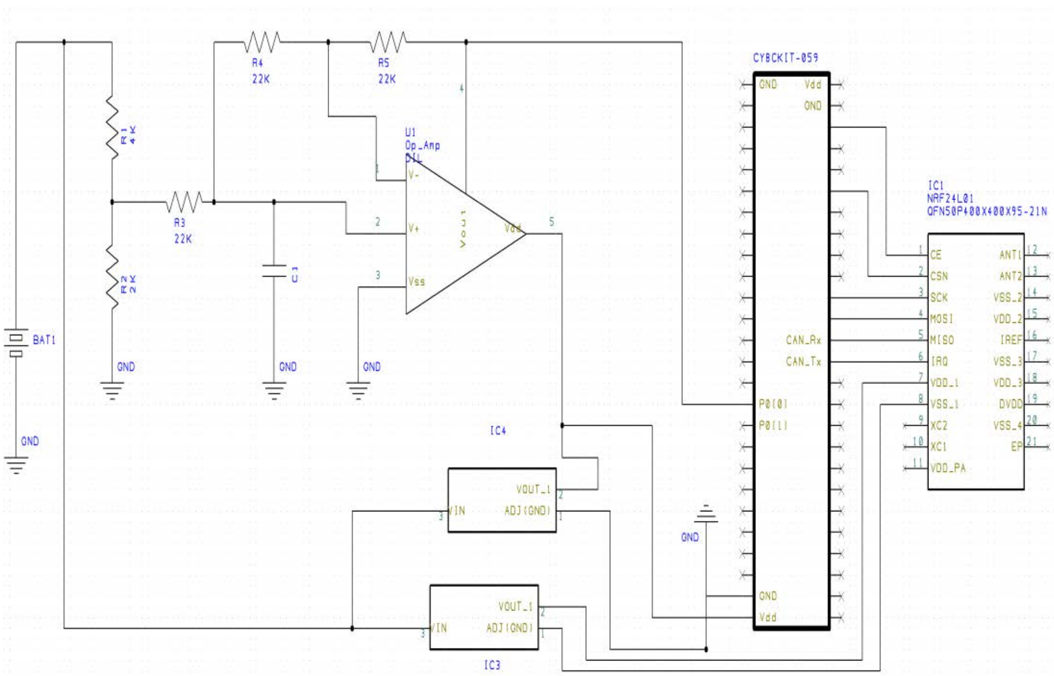


Figura 11.2.1: schema circuitale di un nodo Slave della rete di comunicazione radio

Sulla parte in alto a sinistra in figura 11.2.1 si può vedere il circuito di acquisizione della tensione della batteria realizzato mediante partitore di tensione e filtro passa-basso; il segnale analogico acquisito viene posto in ingresso ad un pin analogico dedicato del kit CY8CKIT_059 a cui corrisponde un canale dedicato del convertitore analogico-digitale della scheda. La batteria alimenta, come evidenziato nella parte bassa della medesima figura, sia l'amplificatore operazionale del filtro passa-basso e il kit a 5V tramite regolatore di tensione LM7805CT/NOPB che il chip nRF24L01 a 3.3V tramite regolatore di tensione AP1117 TO220-03. Si deve evidenziare in questa fase progettuale come il componente AP1117 TO220-03 necessiti dell'utilizzo di due condensatori per garantirne una sicura alimentazione e un corretto funzionamento filtrando le tensioni in entrata e in uscita dal componente, come viene riportato in figura 11.2.2.

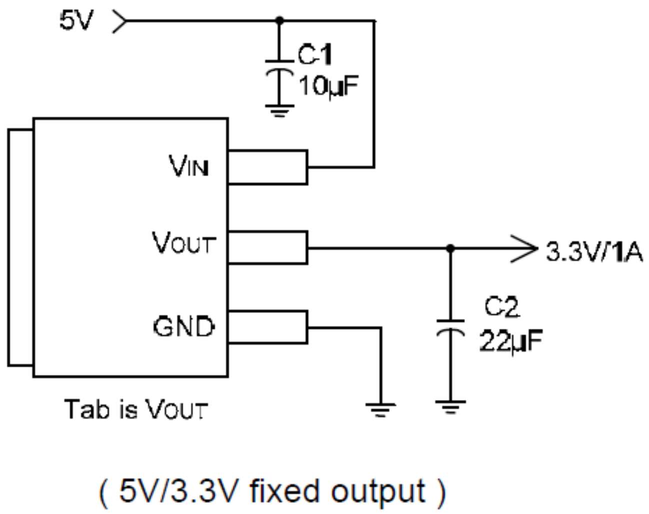


Figura 11.2.2: corretto utilizzo del componente AP1117 TO220-03

Il kit CY8CKIT-059 viene poi collegato al chip nRF24L01 come illustrato in precedenza nella trattazione; in figura 11.2.1 l’attribuzione dei diversi pin della scheda Cypress per comunicare tramite SPI con il ricetrasmettitore è puramente schematica e non rispecchia l’effettivo collegamento dei pin realizzato nel progetto in esame. Va sottolineato inoltre come la medesima figura non restituisca visivamente la semplificazione del cablaggio del circuito hardware apportata in seguito alla sostituzione della comunicazione tramite CAN bus con quella radio, fattore invece evidenziato in figura 11.2.3.

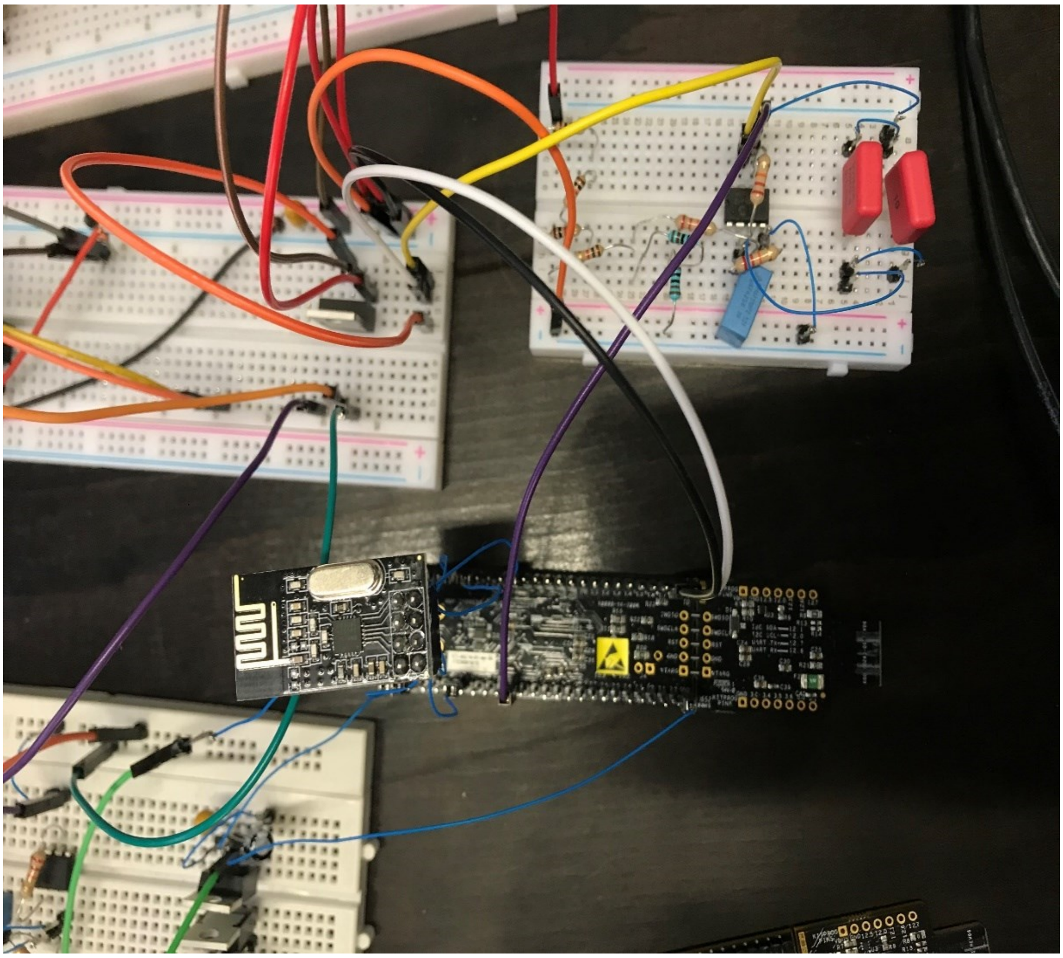


Figura 11.2.3: setup hardware di un nodo Slave della rete di comunicazione radio

Il circuito relativo al nodo Master della rete di comunicazione è del tutto analogo a quello precedentemente sviluppato per la trasmissione dei dati tramite CAN bus; va sostituito solamente il collegamento al CAN bus con un chip nRF24L01.

Il primo test che si conduce è svolto al solo scopo di testare la comunicazione radio tra un nodo Slave e quello Master, implementando come ulteriore verifica del corretto funzionamento della rete di comunicazione radio la trasmissione tramite UART dei dati scambiati tra le due schede al PC. Il setup hardware necessario per implementare il nodo che si affaccia al PC è dunque ancora più semplice di quello appena presentato per il nodo Slave per realizzare il primo test sperimentale; si deve tenere in considerazione che nell'esperimento finale deve essere aggiunto allo schema riportato nel seguito il circuito di acquisizione della corrente. In un primo momento dunque lo schema hardware da realizzare è costituito solamente da un kit CY8CKIT-059 alimentato tramite PCB USB del KitProg, un regolatore di tensione AP1117 TO220-03 ed un chip nRF24L01; ne viene riportato lo schema circuitale in figura 11.2.4.

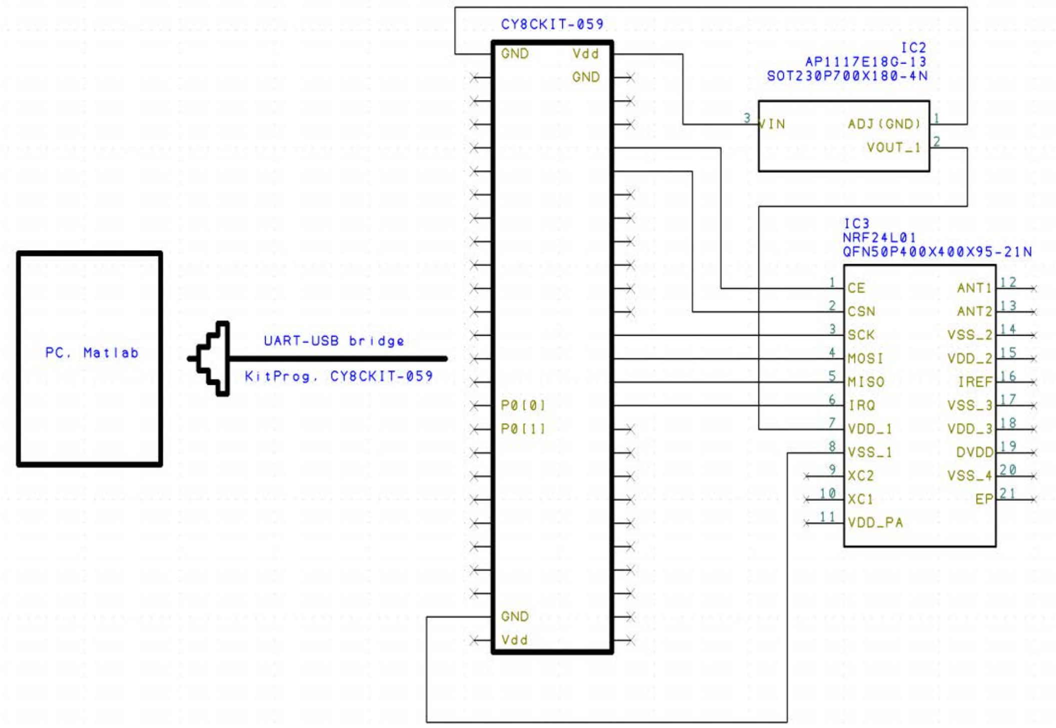


Figura 11.2.4: schema circuitale del nodo Master della rete di comunicazione radio

Terminata la descrizione del circuito hardware realizzato per il primo test, si procede a descrivere l'implementazione della parte software.

11.3. Progettazione e implementazione del software

Per quanto concerne la parte software, sono già state illustrate nei precedenti capitoli le configurazioni dei componenti ADC_SAR_Seq e UART, il primo necessario per convertire in un segnale digitale la tensione acquisita dalla batteria monitorata da un nodo Slave e il secondo per inviare tramite trasmissione asincrona i dati raccolti dal nodo Master al PC, così da poterli visualizzare graficamente rielaborandoli tramite Matlab. Se da un lato non è più necessario che un nodo Slave utilizzi il componente CAN del kit CY8CKIT-059, dall'altro si deve introdurre ed impiegare il componente SPI Master messo a disposizione dalla scheda per interfacciarsi con il chip nRF24L01. Graficamente il componente SPI Master si presenta come in figura 11.3.1.

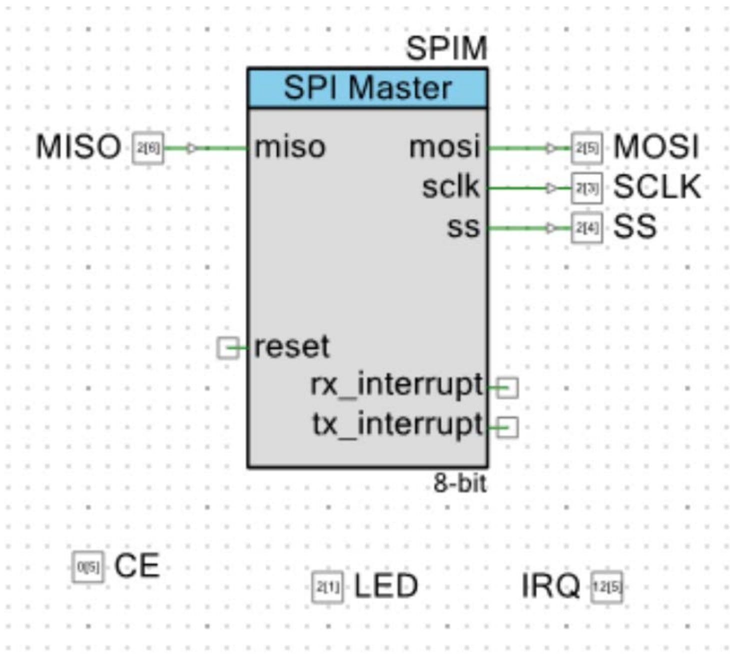


Figura 11.3.1: componente SPI Master del kit CY8CKIT-059

I pin CE e IRQ non fanno parte del componente SPI Master non essendo indispensabili alla comunicazione SPI ma sono comunque necessari per configurare e controllare il chip nRF24L01 come già illustrato in precedenza. Così come per gli altri componenti della scheda Cypress, cliccandoci sopra due volte si apre una finestra grafica di configurazione; per il progetto in esame il componente SPI Master deve essere impostato come visibile in figura 11.3.2. La finestra generale di configurazione permette di impostare diversi parametri tra cui:

- Mode, definisce la fase (CPHA) e la polarità (CPOL) del clock che si desidera utilizzare per realizzare la comunicazione SPI
- Data Lines, definisce quale interfaccia è utilizzata per la comunicazione SPI (a 4 fili MOSI + MISO o a 3 fili bidirezionale)
- Data bits, definisce la larghezza in bit di ogni singolo trasferimento di dati quando si utilizzano le apposite funzioni di lettura e scrittura SPI
- Shift Direction, definisce la direzione in cui vengono trasmessi i dati seriali; quando impostato su MSBit First, il bit più significativo viene trasmesso per primo spostando i dati a sinistra mentre quando impostato su LSBit First, il bit meno significativo viene trasmesso per primo spostando i dati verso destra

- Bit Rate, se il parametro ‘Clock selection’ è impostato su ‘Clock interno’ allora definisce la velocità del clock sul pin SCLK in Hertz; la frequenza di clock del clock interno sarà pari a due volte la frequenza SCLK

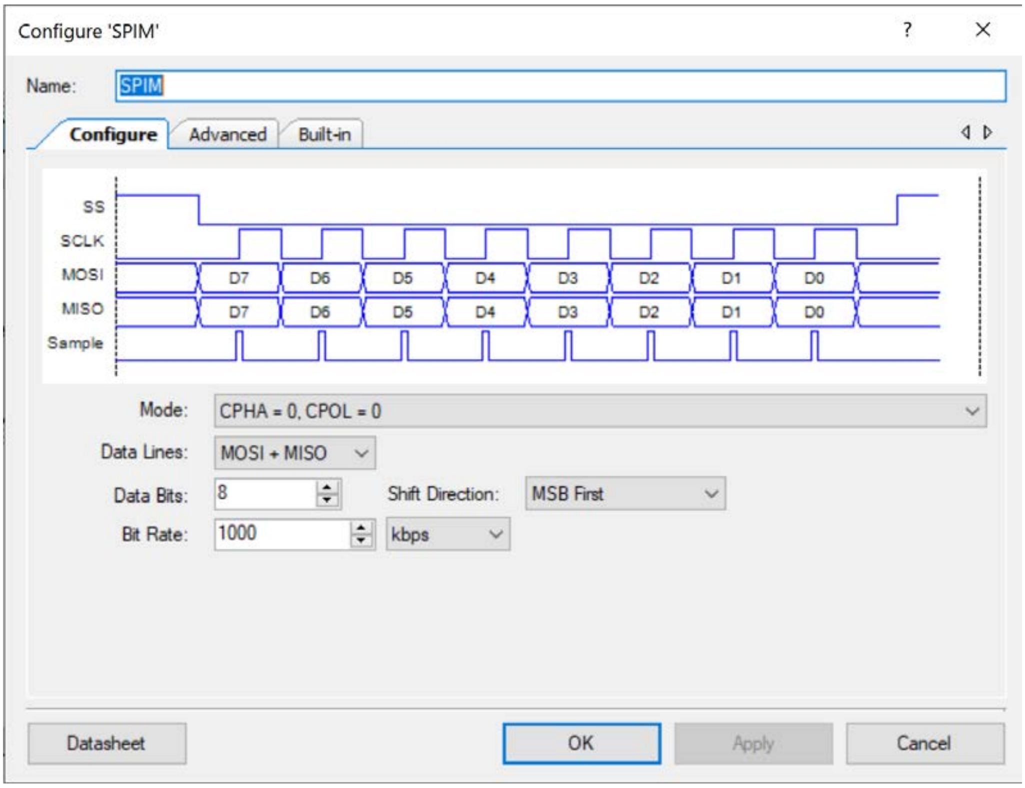


Figura 11.3.2: finestra di dialogo generale per settare il componente SPI Master

È disponibile inoltre una seconda finestra di configurazione avanzata, riportata in figura 11.3.3, che permette di scegliere tra un clock configurato internamente o un clock configurato esternamente per controllare la velocità dei dati e la generazione SCLK, abilitare o meno la modalità di comunicazione ad alta velocità, impostare il numero di byte dedicati ai buffer di ricezione RX e trasmissione TX e abilitare degli eventi di interrupt legati non solo allo stato della comunicazione SPI ma anche a quello dei registri TX FIFO e RX FIFO.

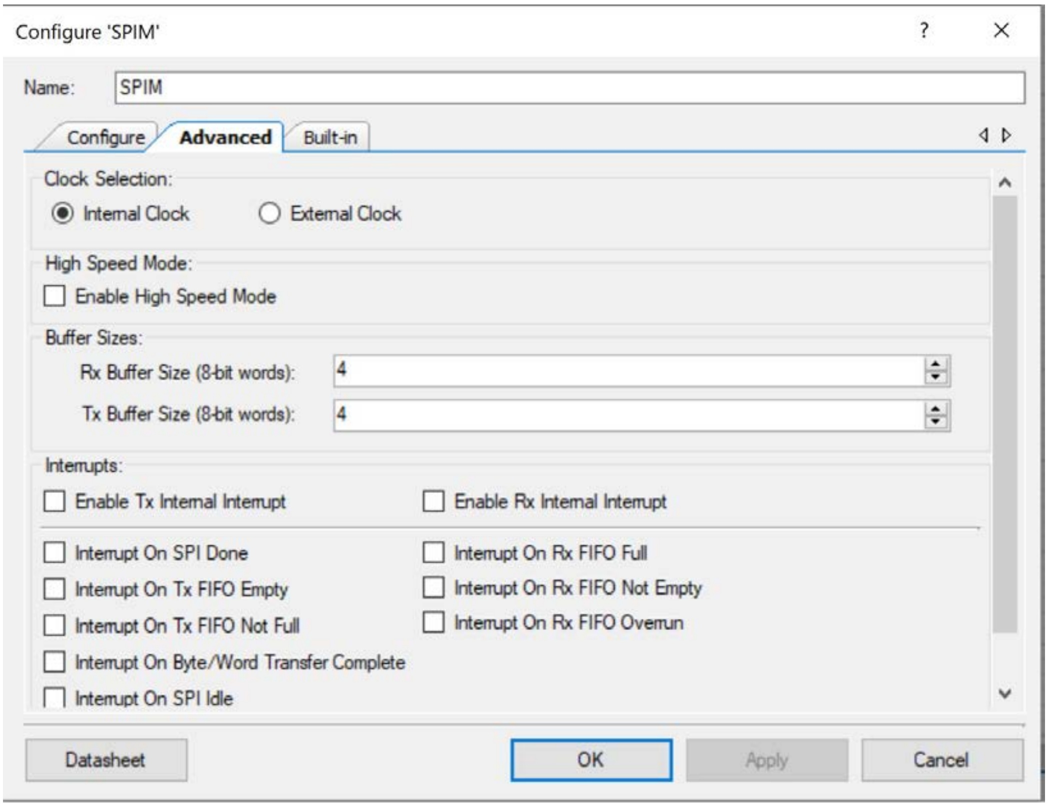


Figura 11.3.3: finestra di configurazione avanzata del componente SPI Master

L'unico parametro che si seleziona per il progetto in esame rispetto ai valori predefiniti del componente SPI Master è quello relativo alla selezione del clock di riferimento, scegliendo in questo caso particolare quello disponibile internamente al kit CY8CKIT-059.

Una volta realizzati i circuiti hardware e configurati i componenti software secondo le considerazioni ed indicazioni appena descritte, il passo successivo è sviluppare un codice opportuno sia per il nodo Slave che per quello Master della rete di comunicazione radio affinché svolgano quanto desiderato dall'utente. In particolare, si sono svolti diversi test, dapprima per verificare il funzionamento fisico dei circuiti realizzati, poi per assicurarsi della corretta configurazione dei diversi componenti elettronici utilizzati.

11.4. Verifica della rete di comunicazione radio

Il primo test viene effettuato come verifica della corretta progettazione e realizzazione della rete di comunicazione radio illustrata nel presente capitolo. Il proposito dell'esperimento è programmare il nodo Master della comunicazione inizialmente come PTX (trasmettitore) e fargli inviare una sequenza numerica a dente di sega al nodo Slave inizialmente configurato come PRX (ricevitore); una volta che il nodo Slave legge il messaggio ricevuto si configura come PTX e invia al nodo Master, nel frattempo configuratosi come PRX, il valore letto nel messaggio ricevuto incrementato di 1. Quando il nodo Master riceve la risposta del nodo Slave, invia sia il proprio messaggio iniziale che la risposta ricevuta dal nodo Slave al PC tramite UART così da poterne visualizzare graficamente in tempo reale i valori scambiati. Si può facilmente intuire che gli andamenti dei dati visualizzati che ci si aspetta di riscontrare nei grafici Mtalab debbano essere entrambi a dente di sega; in particolare, l'andamento associato alla risposta del nodo Slave deve trovarsi al di sopra di quello relativo al messaggio inviato dal nodo Master, dato che la risposta del nodo Slave deriva dall'incremento della sequenza inviata dal nodo Master. Si riporta nel seguito solamente il codice in C sviluppato per programmare il kit CY8CKIT-059 operante da Master della comunicazione radio data la notevole lunghezza che lo caratterizza; lo script con cui programmare il kit Slave può essere scritto facendo riferimento alla spiegazione del test appena riportata e allo script del nodo Master per quanto concerne la sintassi di programmazione e la strutturazione del codice.

```
int main(){
    uint8 spiData[2];
    uint8 spiDataDoppio[4];
    uint8 dati=0;
    CyGlobalIntEnable;
    SPIM_Start();                // avvio l'SPI
    UART_Start();

    for(;;){
    CE_Write(0);
    SS_Write(1);
    LED_Write(0);  // spengo il LED
    SPIM_ClearRxBuffer();
    // Settaggio in scrittura del registro EN_RXADDR
    spiData[0]=W_REGISTER | EN_RXADDR;
    spiData[1]=nRF_EN_RXADDR_INIT;
    SS_Write(0);
```

```
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData, 2);    // abilito il data pipe 0
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro EN_AA
spiData[0]=W_REGISTER | EN_AA;
spiData[1]=nRF_EN_AA_INIT;
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData, 2);    // Disabilito EN
                              // AutoAcknowledgment
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro SETUP_AW
spiData[0] = W_REGISTER | SETUP_AW;
spiData[1] = nRF_SETUP_AW_INIT;        // 0000 0001
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData, 2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro SETUP_RETR
spiData[0] = W_REGISTER | SETUP_RETR;
spiData[1] = nRF_SETUP_RETR_INIT;       // 0000 0000
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData, 2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro TX_ADDR
spiDataDoppio[0] = W_REGISTER | TX_ADDR;
spiDataDoppio[1] = nRF_TX_ADDR_INIT_B0;    // 1110 0110
spiDataDoppio[2] = nRF_TX_ADDR_INIT_B1;    // 1110 0110
spiDataDoppio[3] = nRF_TX_ADDR_INIT_B2;    // 1110 0110
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiDataDoppio, 4);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro FEATURE
spiData[0] = W_REGISTER | FEATURE
spiData[1] = nRF_FEATURE_INIT;
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData, 2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);
// Settaggio del pacchetto da inviare ACTIVATE
spiData[0] = 0b01010000;
spiData[1] = 0x73;
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData, 2);   // abilito comando
```

```
                                // W_TX_PAYLOAD_NOACK
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);
// Settaggio del pacchetto da inviare W_TX_PAYLOAD_NOACK
spiData[0] = W_TX_PAYLOAD_NOACK ;
spiData[1] = DUMMYDATA;
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);
// Settaggio in scrittura del registro CONFIG
spiData[0] = W_REGISTER | CONFIG;
spiData[1] = nRF_CONFIG_TX_INIT;      // 0101 1010
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelay(100);
if(dati<=9){
    // Settaggio del pacchetto da inviare W_TX_PAYLOAD
    spiData[0] = W_TX_PAYLOAD;
    spiData[1] = dati;
    SS_Write(0);
    CyDelayUs(10);   // aspetto 10 microsecondi
    SPIM_PutArray(spiData,  2);
    CyDelayUs(10);   // aspetto 10 microsecondi
    SS_Write(1);
    CyDelayUs(10);   // aspetto 10 microsecondi
   }else{
    dati=0;
    // Settaggio del pacchetto da inviare W_TX_PAYLOAD
    spiData[0] = W_TX_PAYLOAD;
    spiData[1] = dati;
    SS_Write(0);
    CyDelayUs(10);   // aspetto 10 microsecondi
    SPIM_PutArray(spiData,  2);  // abilito l'invio 'dati'
    CyDelayUs(10);   // aspetto 10 microsecondi
    SS_Write(1);
    CyDelayUs(10);   // aspetto 10 microsecondi
   }
    CE_Write(1);  // attivo la trasmissione
    CyDelayUs(145);
    CE_Write(0);   // disattivo la trasmissione
    CyDelayUs(15);
    if(IRQ_Read() == 0){
        // Pulizia TX FIFO
        spiData[0]=FLUSH_TX; // 1110 0001, pulisco il TX FIFO
        spiData[1]=DUMMYDATA;
        SS_Write(0);
        CyDelayUs(10);   // aspetto 10 microsecondi
        SPIM_PutArray(spiData,  2);
        CyDelayUs(10);   // aspetto 10 microsecondi
        SS_Write(1);
        CyDelayUs(10);
        // Pulizia interrupt TX_DS sull'IRQ
```

```
spiData[0]=W_REGISTER | STATUS; // 0010 0111, pulisco
                                // TX_DS
spiData[1]=TX_DS_bit;  // 0010 0000, mask bit TX_DS
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);    // pulisco interrupt
                               // TX_DS
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
/*  ======================================== */
// Configuro il chip in modalità PRX per ricevere la
// risposta
/*  ======================================== */
// Pulizia RX FIFO
spiData[0]=FLUSH_RX; // 1110 0010, pulisco l'RX FIFO
spiData[1]=DUMMYDATA;     // 0000 0001
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);
// Settaggio in scrittura del registro EN_AA
spiData[0]=W_REGISTER | EN_AA;
spiData[1]=nRF_EN_AA_INIT;     // 0000 0000
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);
// Settaggio in scrittura del registro EN_RXADDR
spiData[0]=W_REGISTER | EN_RXADDR;
spiData[1]=nRF_EN_RXADDR_INIT;     // 0000 0001
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);    // abilito il data pipe 0
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro SETUP_AW
spiData[0] = W_REGISTER | SETUP_AW;
spiData[1] = nRF_SETUP_AW_INIT;        // 0000 0001
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);
// Settaggio in scrittura del registro SETUP_RETR
spiData[0] = W_REGISTER | SETUP_RETR;
spiData[1] = nRF_SETUP_RETR_INIT;       // 0000 0000
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);
```

```
// Settaggio in scrittura del registro RX_ADDR_P0
spiDataDoppio[0] = W_REGISTER | RX_ADDR_P0;
spiDataDoppio[1] = nRF_RX_ADDR_P0_INIT_B0;
spiDataDoppio[2] = nRF_RX_ADDR_P0_INIT_B1;
spiDataDoppio[3] = nRF_RX_ADDR_P0_INIT_B2;
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiDataDoppio,  4);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro RX_PW_P0
spiData[0] = W_REGISTER | RX_PW_P0 ;
spiData[1] = nRF_RX_PW_P0_INIT;       // 0000 0001
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
// Settaggio in scrittura del registro CONFIG
spiData[0] = W_REGISTER | CONFIG;
spiData[1] = nRF_CONFIG_RX_INIT;      // 0011 1011
SS_Write(0);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_PutArray(spiData,  2);
CyDelayUs(10);   // aspetto 10 microsecondi
SS_Write(1);
CyDelayUs(10);   // aspetto 10 microsecondi
SPIM_ClearRxBuffer();
CyDelay(2);
CE_Write(1);  // attivo la ricezione
CyDelay(250);
CE_Write(0);   // disattivo la ricezione
CyDelayUs(15);
if(IRQ_Read() == 0){
    LED_Write(1);
    // Lettura del pacchetto ricevuto R_RX_PAYLOAD
    spiData[0] = R_RX_PAYLOAD ;
    spiData[1] = DUMMYDATA;
    SS_Write(0);
    CyDelayUs(10);   // aspetto 10 microsecondi
    SPIM_PutArray(spiData,  2);
    CyDelayUs(10);   // aspetto 10 microsecondi
    SS_Write(1);
    CyDelayUs(10);   // aspetto 10 microsecondi
    uint8 controllo26 = SPIM_ReadRxData();
    uint8 messaggio_ricevuto = SPIM_ReadRxData();
    char TransmitBuffer[20];
sprintf(TransmitBuffer, "\r%d %d",dati,messaggio_ricevuto);
    UART_PutString(TransmitBuffer);
    // Pulizia RX FIFO
    spiData[0]=FLUSH_RX; // 1110 0010, pulisco l'RX FIFO
    spiData[1]=DUMMYDATA;
    SS_Write(0);
    CyDelayUs(10);   // aspetto 10 microsecondi
    SPIM_PutArray(spiData,  2);
    CyDelayUs(10);   // aspetto 10 microsecondi
    SS_Write(1);
```

```
                CyDelayUs(10);   // aspetto 10 microsecondi
                // Pulizia interrupt RX_DR sull'IRQ
                spiData[0]=W_REGISTER | STATUS; // 0010 0111,
                                                // pulisco RX_DR
                spiData[1]=RX_DR_bit;  // 0100 0000, mask bit RX_DR
                SS_Write(0);
                CyDelayUs(10);   // aspetto 10 microsecondi
                SPIM_PutArray(spiData,  2);    // pulisco interrupt
                                               // RX_DR
                CyDelayUs(10);   // aspetto 10 microsecondi
                SS_Write(1);
                CyDelay(100);
                dati++;
               }
    }
}
/* [] END OF FILE */
```

Nel codice sopra riportato ricorre l'utilizzo frequente di costanti, evidenziate dall'essere scritte con il colore viola e che non sono specificate in alcun punto della routine; qualora il lettore fosse interessato ad approfondirne le definizioni e le inizializzazioni, si rimanda alla consultazione dell'Appendice B a fine trattazione e del datasheet del chip nRF24L01 per una maggiore comprensione dei valori assunti dalle costanti.

Non si riporta in questo capitolo nemmeno il codice Matlab sviluppato per acquisire i dati dal nodo Master tramite ponte USB-UART, in quanto molto simile a quelli già sviluppati e illustrati a proposito dei test effettuati per la rete di comunicazione basata sul CAN bus. In figura 11.4.1 si può vedere il grafico restituito da Matlab in merito al test sperimentale appena esposto.

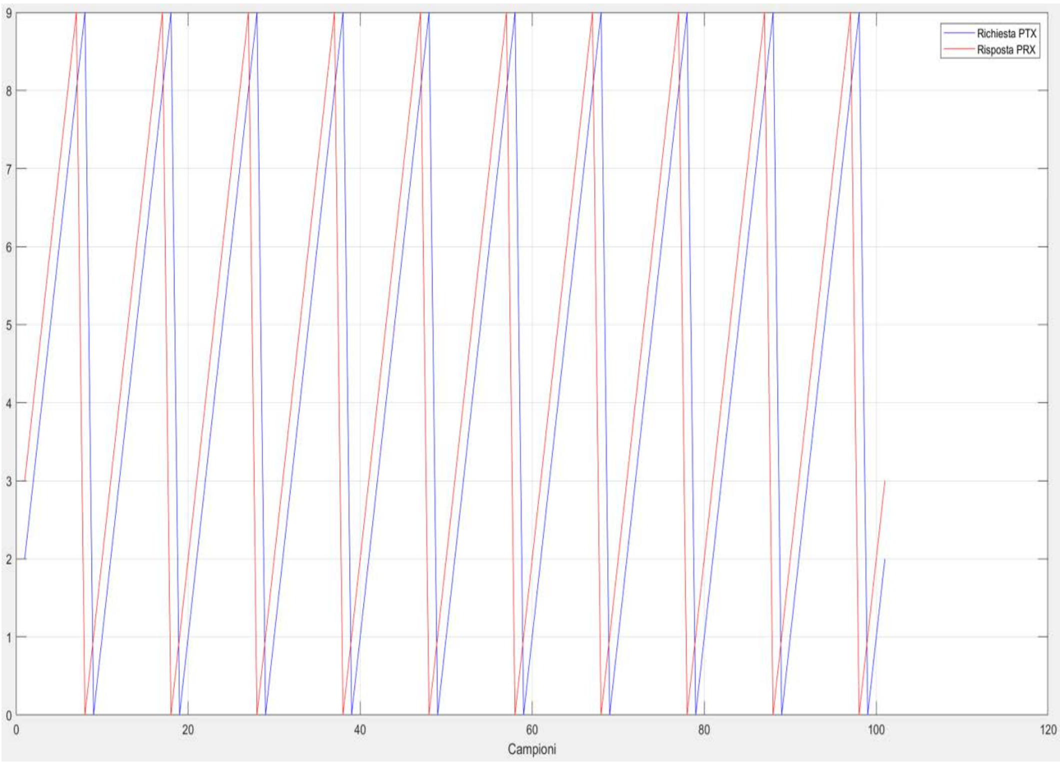


Figura 11.4.1: andamento in real-time dei dati scambiati via radio tra il nodo Master e quello Slave

Come si può vedere dalla figura 11.4.1 i due andamenti restituiti graficamente corrispondono ad un centinaio di routine ‘richiesta-risposta’ tra nodo Master e nodo Slave; così come evidenziato nella legenda, l’andamento rosso a dente di sega relativo alla risposta del dispositivo PRX (nodo Slave) anticipa di un ciclo il valore del corrispondente andamento blu relativo alla richiesta del chip nRF24L01 configurato come PTX (nodo Master), il tutto in linea con le aspettative sperimentali. Si può concludere quindi che sia il circuito hardware realizzato che la programmazione software dei kit CY8CKIT-059 e dei chip nRF24L01 rispondono perfettamente agli scopi secondo cui sono progettati e implementati.

Si procede quindi ad applicare il sistema di comunicazione radio al monitoraggio della tensione e corrente nelle fasi di carica e scarica di un pacco composto da quattro batterie al piombo acido poste in serie, analogamente a quanto fatto per la comunicazione via CAN bus. Il fine ultimo è quello di verificare che la rete di comunicazione radio sia in grado di rispondere repentinamente alle variazioni di tensione e corrente delle batterie monitorate, garantendo un funzionamento accurato in tempo reale e restituendo graficamente l’andamento delle grandezze d'interesse tramite Matlab. Per svolgere questo test conclusivo rispetto all'esperimento precedente si deve realizzare nel nodo Slave il circuito di acquisizione della tensione di una batteria e nel nodo Master la parte hardware di acquisizione della corrente, oltre che modificare la programmazione dei kit CY8CKT-059, processi ampiamente illustrati nei capitoli relativi alla verifica della rete di comunicazione basata sul CAN bus. Il setup hardware necessario per realizzare il test si presenta come in figura 11.4.2. Nella parte centrale della figura si può distinguere il nodo Master contraddistinto dal circuito di acquisizione della corrente tramite sonda LEM; nella parte alta invece si riconosce il nodo Slave con relativo circuito di acquisizione della tensione della batteria monitorata.

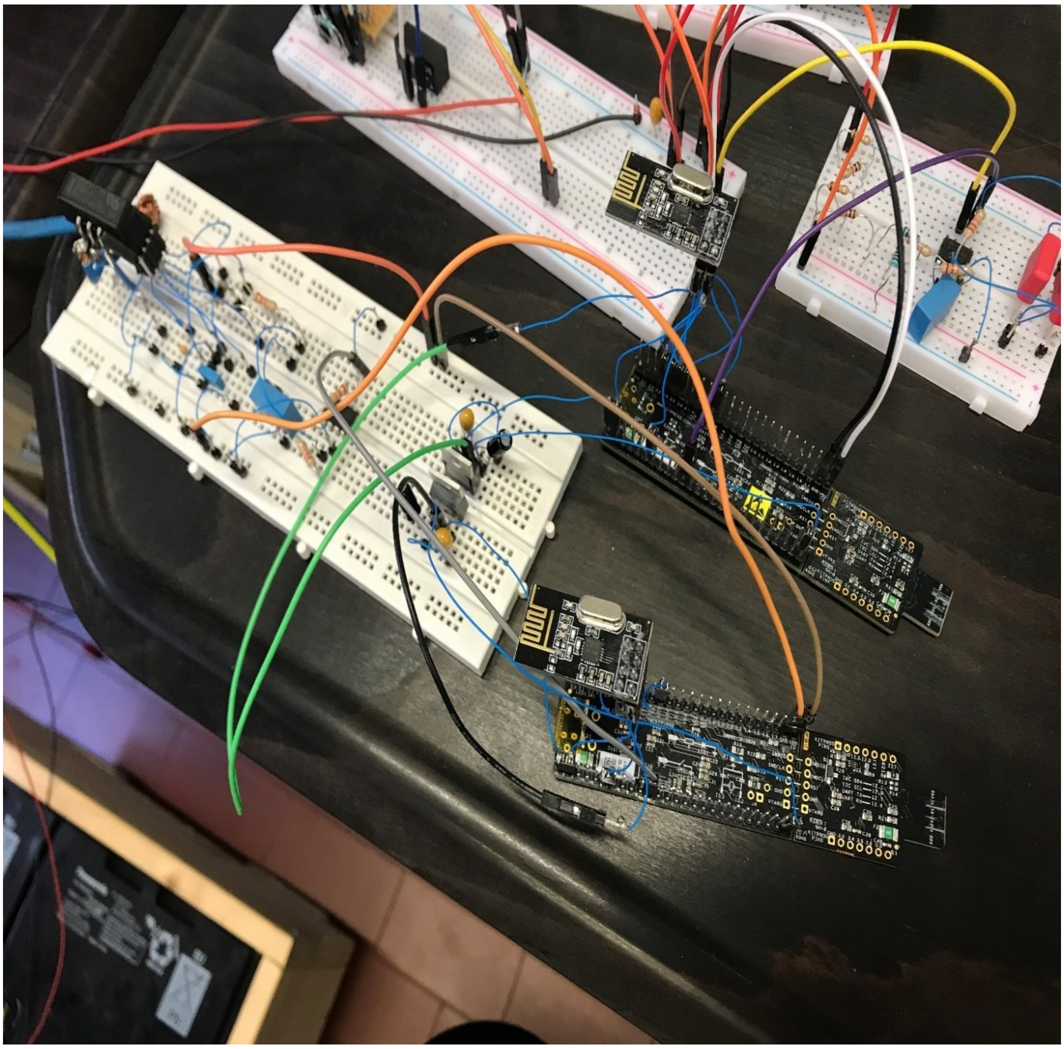


Figura 11.4.2: setup hardware per l'esperimento conclusivo

È altresì evidente in figura 11.4.2 la semplificazione del cablaggio necessario da realizzare rispetto alla rete di comunicazione basata sul CAN bus e sviluppata all'inizio della trattazione; è più difficile notare invece l'assenza di un collegamento elettrico tra i due nodi dato che il mezzo fisico trasmissivo per la comunicazione radio è l'aria.

La prima prova effettuata è stata quella di carica del pacco batterie; in figura 11.4.3 sono visibili i grafici restituiti da Matlab e tracciati acquisendo tensione e corrente del pacco batterie nell'arco di una decina di minuti.

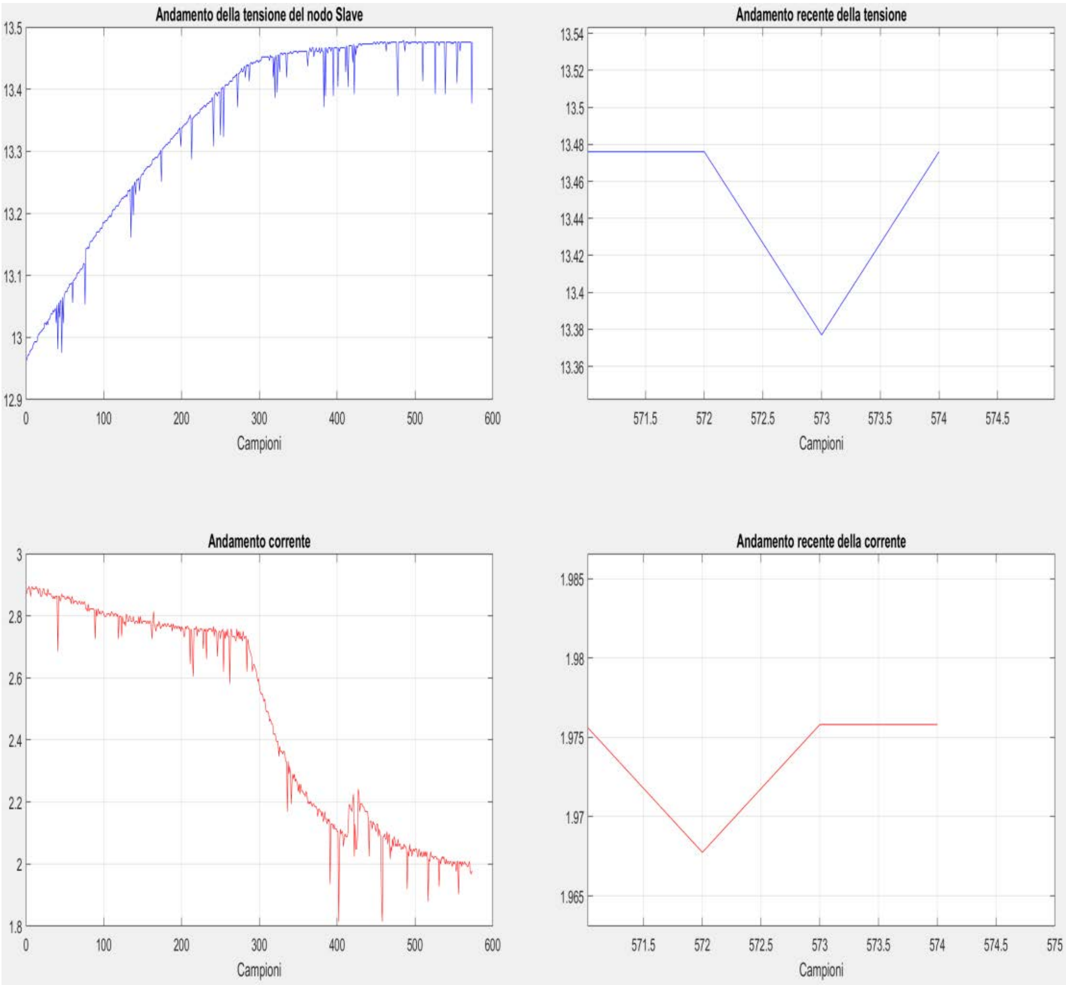


Figura 11.4.3: andamento di tensione e corrente durante la fase di carica

Come si può notare in figura 11.4.3 l'intero periodo di carica delle batterie è rilevato con precisione dal sistema di monitoraggio realizzato e l'elevata velocità di trasmissione dati della comunicazione radio implementata mediante i chip nRF24L01 permette una visualizzazione in tempo reale delle grandezze acquisite. In particolare, è ben distinguibile poco prima del trecentesimo campione acquisito il marcato decremento della corrente di carica fornita dal caricabatterie in prossimità del raggiungimento della tensione di fine carica della batteria.

Si devono poi evidenziare le fluttuazioni visibili negli andamenti riportati in figura; per quanto concerne la tensione tali oscillazioni sono dovute a disturbi legati al circuito di acquisizione dato che non interessano più campioni acquisiti consecutivamente, mentre per quanto riguarda la corrente è più probabile che siano dovute al funzionamento del caricabatterie stesso allo scopo di ottimizzare il processo di carica. Seppur evidenti in figura 11.4.3, si deve tenere conto della scala utilizzata per l'asse delle ordinate; in termini relativi infatti tali fluttuazioni si attestano su valori <1‰ nel caso della tensione della batteria monitorata mentre assumono valori molto più elevati (≃15%) nel caso della corrente del pacco batterie, a riconferma delle supposizioni sopra illustrate.

Un altro aspetto importante da sottolineare prima di procedere con la trattazione riguarda il codice Matlab utilizzato per l'esperimento; l'autore non ha dovuto effettuare nessuna correzione di guadagno e offset sulle acquisizioni di tensione e corrente. Ciò a testimonianza dell'ininfluenza sulle grandezze fisicamente monitorate del circuito hardware complessivo del sistema di monitoraggio basato su comunicazione radio realizzato in questo capitolo. Nei test effettuati nel capitolo 10 a proposito della rete di comunicazione basata sul CAN bus sono evidenti le perturbazioni causate dai numerosi collegamenti fisici tra componenti necessari all'implementazione del sistema di monitoraggio; la riduzione del cablaggio e la semplificazione del setup hardware essenziale per la comunicazione radio riduce drasticamente questo effetto di distorsione sulle grandezze acquisite. In tal caso quindi gli andamenti grafici restituiti da Matlab coincidono con l'effettiva tensione e corrente delle batterie, come confermato da misure voltmetriche e amperometriche mediante multimetro.

La seconda e ultima prova effettuata è quella di scarica del pacco batterie su un reostato variabile, in modo del tutto analogo a quanto riportato nel capitolo 10; in figura 11.4.4 sono visibili i grafici restituiti da Matlab e tracciati acquisendo tensione e corrente del pacco batterie nell'arco di una decina di minuti.

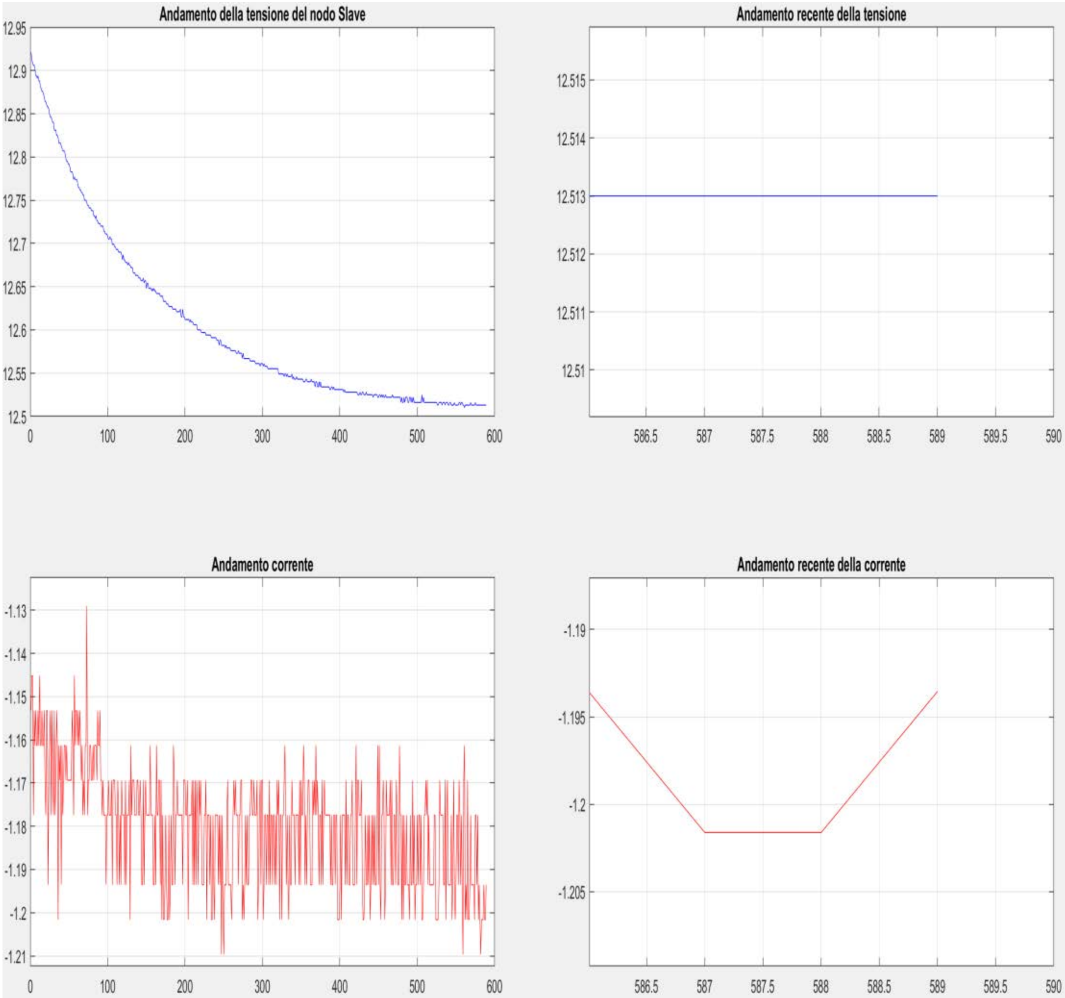


Figura 11.4.4: andamento di tensione e corrente durante la fase di scarica

L'andamento pressoché privo di distorsioni della tensione acquisita dalla batteria, visibile nel grafico in alto a sinistra della figura 11.4.4, costituisce un'ulteriore riprova dell'ininfluenza del relativo circuito di acquisizione e delle connessioni effettuate sull'acquisizione stessa; permangono invece sulla corrente delle fluttuazioni nell'ordine dell'1%, probabilmente dovute alle saldature e ai collegamenti effettuati per far funzionare correttamente la sonda LEM e comunque non influenti dal punto di vista operativo sul sistema di monitoraggio implementato.

Ricapitolando, i risultati sperimentali ottenuti evidenziano come il sistema di monitoraggio distribuito basato sulla comunicazione radio fra i nodi della rete garantisca una precisione di acquisizione dei dati decisamente superiore rispetto a quella fornita dal sistema basato sul CAN bus, ciò grazie alla semplificazione dei circuiti hardware da realizzare intesa come minimizzazione del numero di componenti elettronici da impiegare e del cablaggio che ne consegue. Oltre a contenere il volume occupato dal sistema di monitoraggio da installare sulle singole batterie di un pacco, la tensione e la corrente acquisite vengono perturbate molto meno marcatamente dai relativi circuiti di acquisizione. È evidente quindi come la comunicazione radio apporti delle migliorie notevoli al sistema di monitoraggio basato sul CAN bus e sia indubbiamente la scelta preferibile qualora si volesse implementare un sistema di monitoraggio distribuito della carica e scarica di una batteria.

12.Conclusione

La presente trattazione propone due diversi sistemi di gestione della batteria interamente funzionanti e pronti per essere applicati in casi pratici d'interesse. In particolare, nella prima parte della trattazione si progetta e realizza un sistema di monitoraggio distribuito della carica e scarica di una batteria basato sulla comunicazione dei nodi della rete tramite CAN bus. In seguito all'esposizione delle basi teoriche fondamentali da studiare e approfondire per il progetto in esame, si illustrano i diversi passaggi relativi alla realizzazione dei circuiti hardware e alla programmazione software dei kit CY8CKIT-059 distinguendo i nodi Slave da quello Master della rete di comunicazione da implementare. Infine, si riportano i test sperimentali effettuati per validare il corretto funzionamento del sistema sviluppato, insieme a considerazioni per quanto concerne il suo funzionamento e la sua eventuale applicazione ad un veicolo elettrico alimentato a batterie. Da quest'ultima parte si prende spunto per progettare e realizzare un sistema di monitoraggio distribuito alternativo basato sulla comunicazione radio mediante i chip nRF24L01 fra i nodi della rete. Questo secondo progetto viene sviluppato ed illustrato nella seconda parte della trattazione seguendo lo stesso iter procedurale utilizzato per quello precedente. Gli esperimenti conclusivi eseguiti per il sistema di monitoraggio distribuito basato sulla rete di comunicazione radio ne evidenziano non solo una corretta progettazione e realizzazione, ma anche prestazioni migliori in termini di precisione dell'acquisizione della tensione e corrente delle batterie dovute essenzialmente alla semplificazioni della circuiteria hardware e del cablaggio implementato; inoltre il sistema così creato è più compatto dal punto di vista del volume occupato e del numero di componenti elettronici necessari, risultando più adatto per un utilizzo nel settore automotive, in particolare per quanto concerne la sua installazione a bordo di un BEV (veicolo elettrico alimentato da batterie).

In definitiva, entrambi i sistemi di monitoraggio distribuito implementati in questa trattazione sono perfettamente funzionanti ed applicabili in un contesto reale, tuttavia si sottolinea come la rete di comunicazione radio presenti maggiori vantaggi e prestazioni rispetto a quella basata sul CAN bus.

Ringraziamenti

Un sincero e sentito ringraziamento va innanzitutto al Prof. Manuele Bertoluzzo e a Stefano Giacomuzzi, che hanno impreziosito le giornate trascorse in laboratorio a sviluppare i progetti di tesi; allo stesso modo un pensiero lo dedico anche ai miei compagni di avventura conosciuti durante l'intero percorso di studi, con cui ho condiviso esperienze e valori umani che nessun libro può insegnare, così come ai miei amici più stretti che mi accompagnano da molto molto tempo. Last but not least, voglio ringraziare anche la mia famiglia ed i miei parenti per avermi sopportato e supportato anche nei momenti più impegnativi; a loro devo la persona che sono oggi.

APPENDICE A

Nel corso della trattazione, per quanto concerne la programmazione software dei kit CY8CKIT-059 di Cypress Semiconductor, si citano ripetutamente le Application Program Interfaces (API): sono interfacce informatiche, nel caso specifico del kit CY8CKIT-059 associate ad ogni componente a disposizione in PSoC Creator, che definiscono le interazioni tra più intermediari di software e semplificano la programmazione stessa della scheda da parte dell'utente. Nel presente progetto sono state ampiamente utilizzate per implementare le main routine dei kit operanti sia da nodi Slave che da nodo Master della rete di comunicazione. Si riporta qui nel seguito una lista, con relativa spiegazione ed approfondimento, composta da ogni API impiegata per configurare i componenti ADC_SAR_Seq, CAN, UART e SPI_Master delle schede Cypress:

- ADC_SAR_Seq_Start(): esegue tutte le inizializzazioni necessarie per questo componente e ne abilita l'alimentazione in modo appropriato in base alla frequenza di clock
- ADC_SAR_Seq_StartConvert(): in modalità Free Running inizia il processo di conversione e funziona in modo continuo
- ADC_SAR_Seq_1_IsEndConversion(ADC_SAR_Seq_1_SAR_RETURN_STATUS): restituisce immediatamente lo stato di conversione per i canali sequenziali del convertitore analogico-digitale; se il valore restituito è zero, la conversione non è completa e questa funzione viene richiamata fino a quando non viene restituito un risultato non nullo
- ADC_SAR_Seq_GetResult16(): ritorna in formato int16 i dati disponibili nel registro risultati del canale specificato come input della funzione
- ADC_SAR_Seq_CountsTo_mVolts(): converte in millivolt l'uscita del canale specificato in input; il risultato è espresso come int16 e il calcolo dipende dal valore della tensione di riferimento (ad esempio, se l'ADC ha misurato 0,534 volt, il valore di ritorno è 534)
- CAN_Start(): imposta la variabile initVar, chiama la funzione CAN_Init() e poi chiama la funzione CAN_Enable(); questa funzione imposta il componente CAN in modalità operativa e avvia il contatore se sono disponibili caselle di posta elettronica per il polling

- CAN_ReceiveMsgX(): funzione disponibile solo per le mailbox di ricezione impostate come Full, costituisce l'entry point dell'interrupt relativo alla ricezione di un messaggio; cancella i flag di interrupt della relativa mailbox X
- CAN_SendMsgX(): funzione disponibile solo per le mailbox di trasmissione impostate come Full, costituisce l'entry point dell'interrupt relativo alla trasmissione di un messaggio; controlla se ci sono messaggi non ancora inviati pronti per la fase di arbitraggio della trasmissione e, in caso positivo, ne inizia la trasmissione
- CAN_RX_DATA_BYTEX(CAN_RX_MAILBOX_Y): acquisisce il byte numero X ricevuto nella mailbox numero Y del buffer di ricezione
- CAN_TX_DATA_BYTEX(CAN_TX_MAILBOX_Y): imposta il byte numero X all'interno del data field del CAN frame della mailbox numero Y del buffer di trasmissione
- SPIM_Start(): questa funzione chiama sia SPIM_Init() che SPIM_Enable
- SPIM_PutArray(): posiziona l'array di dati in input alla funzione, specificandone il numero di byte che lo compone, nel buffer di trasmissione
- SPIM_ReadRxData(): riporta il prossimo byte di dati ricevuti disponibili nel buffer di ricezione
- UART_Start(): metodo preferito per inizializzare il componente; imposta la variabile initVar, chiama la funzione UART_Init() e poi chiama la funzione UART_Enable()
- UART_GetChar(): restituisce l'ultimo byte di dati ricevuto; tale funzione è progettata per i caratteri ASCII e restituisce come output un uint8 di valore compreso tra 1 e 255 se si tratta di un carattere valido oppure 0 se si è verificato un errore o non sono presenti dati da ricevere
- UART_PutChar(): inserisce il byte di dati in input alla funzione nel buffer di trasmissione da inviare quando il bus è disponibile; si tratta di un'API di blocco che attende finché il buffer di trasmissione non ha spazio per contenere i dati
- UART_PutString(): invia al buffer di trasmissione una stringa, il cui puntatore va specificato come input della funzione, terminata dal carattere NULL per poi essere trasmessa

APPENDICE B

Nel capitolo 11 si sono riportati alcuni script per programmare i kit CY8CKIT-059 in modo da configurare appropriatamente i chip nRF24L01 tramite SPI a seconda dello scopo dei vari test sperimentali condotti; in essi si può notare un frequente utilizzo di costanti non dichiarate esplicitamente ma fondamentali per la corretta compilazione dei codici. Si riporta qui nel seguito la lista di tutte le costanti utili nel caso si volessero replicare i test effettuati con i ricetrasmettitori di Nordic Semiconductor. Si suggerisce, per capire appieno il loro significato specifico e la necessità di definirle, di consultare inoltre il datasheet del componente nRF24L01 e si invita a ricordare che tali costanti fanno riferimento ai settaggi specifici del ricetrasmettitore utili per il progetto sviluppato in questa trattazione.

```
// Libreria comandi utili
#define R_REGISTER            0x00 // 0000 0000 lettura di un
                                   // registro
#define W_REGISTER            0x20 // 0010 0000 scrittura di un
                                   // registro
#define R_RX_PAYLOAD          0x61 // 0110 0001 lettura del carico
                                   // utile
#define W_TX_PAYLOAD          0xA0 // 1010 0000 scrittura del carico
                                   // utile
#define FLUSH_TX              0xE1 // 1110 0001 pulizia del TX FIFO
#define FLUSH_RX              0xE2 // 1110 0010 pulizia del RX FIFO
#define R_RX_PL_WID           0x60 // 0110 0000 leggo la lunghezza
                                   // del payload ricevuto
#define W_ACK_PAYLOAD         0xA8 // 1010 1PPP scrivo il payload da
                                   // inviare con l'ACK per pipe PPP
#define NOP                   0xFF // 1111 1111 lettura del registro
                                   // STATUS
#define W_TX_PAYLOAD_NOACK    0b01011000

// Mappa dei registri dell'nRF24L01
#define CONFIG                0x00 // registro CONFIG
#define EN_AA                 0x01 // registro EN_AutoAcknowledgment
#define EN_RXADDR             0x02 // registro EN_RXAddresses
#define SETUP_AW              0x03 // registro SETUP_AddressesWidths
#define SETUP_RETR            0x04 // registro SETUP:Retransmission
#define RF_CH                 0x05 // registro RF_Channel
#define RF_SETUP              0x06 // registro RF_Setup
#define STATUS                0x07 // registro STATUS
#define OBSERVE_TX            0x08 // registro OBSERVE_Transmission
#define RPD                   0x09 // registro Carrier Detect
#define RX_ADDR_P0            0x0A // registro di settaggio
                                   // indirizzo di ricezione del
                                   // pipe 0
#define TX_ADDR               0x10 // registro di settaggio
                                   // indirizzo del trasmittente
#define RX_PW_P0              0x11 // definizione numero di byte del
                                   // payload ricevuto nel pipe 0
#define FIFO_STATUS           0x17 // registro FIFO_STATUS
```

```
#define DYNPD                 0x1C // registro Dynamic Payload
                                   // Detection
#define FEATURE               0x1D // registro FEATURE
#define DUMMYDATA             0x55

// Bit masks
#define RX_DR_bit           0b01000000   // sesto bit del registro
                                         // STATUS
#define TX_DS_bit           0b00100000   // quinto bit del registro
                                         // STATUS
#define MAX_RT_bit          0b00010000   // quarto bit del registro
                                         // STATUS
#define RX_EMPTY            0b00000001   // ottavo bit del registro
                                         // STATUS

// Libreria inizializzazione dei registri
#define nRF_CONFIG_RX_INIT      0b00111011
#define nRF_CONFIG_TX_INIT      0b01011010
#define nRF_EN_AA_INIT          0b00000000
#define nRF_EN_RXADDR_INIT      0b00000001
#define nRF_SETUP_AW_INIT       0b00000001
#define nRF_SETUP_RETR_INIT     0b00000000
#define nRF_RF_CH_INIT          0b00000010
#define nRF_RF_SETUP_INIT       0b00001110
#define nRF_RX_ADDR_P0_INIT_B0  0xE6
#define nRF_RX_ADDR_P0_INIT_B1  0xE7
#define nRF_RX_ADDR_P0_INIT_B2  0xE6
#define nRF_TX_ADDR_INIT_B0     0xE6
#define nRF_TX_ADDR_INIT_B1     0xE7
#define nRF_TX_ADDR_INIT_B2     0xE6
#define nRF_RX_PW_P0_INIT       0x01
#define nRF_DYNPD_INIT          0b00000000
#define nRF_FEATURE_INIT        0b00000001
```

Bibliografia

[1] Samuel Matrella, “Studio di un sistema per il monitoraggio della carica degli accumulatori di un veicolo elettrico”, 2020.

[2] "How to monitor a battery", [ONLINE], Access: https://batteryuniversity.com/learn/article/how_to_monitor_a_battery

[3] "BMS: sistema di monitoraggio e gestiione dell batterie agli ioni di Litio", [ONLINE], Access: https://farelettronica.it/web/bms-sistema-di-monitoraggio-e-gestione-delle-batterie-al-litio/

[4] Cypress Semiconductor, “CY8CKIT-059 PSoC ® 5LP Prototyping Kit Guide.”

[5] Cypress Semiconductor, “KitProg User Guide,” no. 001, 2016.

[6] Cypress Semiconductor, “PSoC CreatorUser Guide.”

[7] "CAN specification", Settembre 1991.

[8] PSoC 3 and PSoC 5LP – Getting Started with Controller Area Network (CAN).

[9] PSoC® Creator™ Component Datasheet, Controller Area Network (CAN) 3.0, 20 Ottobre 2017.

[10] "Basics UART communication", [ONLINE], Access: https://www.circuitbasics.com/basics-uart-communication/.

[11] PSoC® Creator™ Component Datasheet, Universal Asynchronous Receiver Transmitter (UART) 2.30, 8 Febbraio 2016.

[12] ISO1050DWR datasheet.

[13] LM7805CT/NOPB datasheet.

[14] SPA01A-05 datasheet.

[15] SM82A 48V datasheet.

[16] nRF24L01 datasheet.

[17] AP1117 TO220-3 datasheet.