POLITECNICO DI TORINO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Integrazione di un sottosistema di comunicazione per nanosatellite tollerante ai guasti

Relatori:
Prof. Leonardo Reyneri
Prof. Dante Del Corso

Candidato:
Luca D'Amato

Settembre 2011

A mia madre

Ringraziamenti

Sono finalmente giunto alla fine di questa esperienza durata anni ma che mi ha aiutato a crescere. E' quindi doveroso ringraziare tutte le persone che con le quali ho condiviso questo percorso.
Il primo ringraziamento và ai prof. Reyneri e Del Corso per la loro disponibilità e per avermi dato la possibilità di fare una tesi in un settore interessante come è quello aerospaziale. Un grazie và anche a Danilo per la disponibilità a rispondere alle mie numerose domande.
Vorrei ringraziare tutti i "colleghi" che hanno condiviso con me questo percorso di studi, in particolare l'amico Gabriele, compagno di innumerevoli laboratori e della famosa "nottata".
Un ringraziamento và agli amici del paese per avermi accompagnato nei periodi di vacanza a San Donaci. Un grande "grazie" agli amici che sono a Torino e hanno reso "Corso De Gasperi 59" una seconda casa.
Un ringraziamento speciale và ai miei amici Antonio e Fabio, con i quali ho condiviso tutti questi anni tra gioie e dolori, supportandoci a vicenda e condividendo la vita quotidiana sia nello studio che nei momenti di svago.
Un doveroso e sentito ringraziamento và ai miei familiari, ai miei nonni, zia Mina e zio Giuseppe, Angelo e Maria Lucia per il sostegno e per l'affetto.
"Grazie" immenso a mio padre e mia sorella per il sostegno economico e morale, per la pazienza avuta in tutti questi anni e per i sacrifici che hanno fatto per permettermi di laurearmi fuori sede. Un ultimo ma più importante ringraziamento a mia madre, la persona che più di tutti ha creduto in me e mi ha dato la forza per riuscire a concludere e raggiungere questo obiettivo.

Grazie anche a chi non ho nominato e a chi mi vuole bene…

Sommario

Da qualche anno l'interesse industriale e accademico nello spazio è in rapida crescita. L'avionica per satelliti è un mercato in espansione, soprattutto grazie alla disponibilità di vettori a basso costo di lancio. Tale riduzione dei costi ha attivato molte istituzioni (grandi industrie, ma anche università) per sviluppare i loro satelliti.

Anche il Politecnico di Torino ha sviluppato un suo satellite di nome PiCPoT (Piccolo Cubo del Politecnico di Torino), a forma cubica con lato di circa 12 cm, che aveva come scopi principali il test sul funzionamento dei componenti commerciali (COTS) in ambito spaziale e l'acquisizione di immagini e parametri dell'ambiente spaziale. Oltre ad obiettivi di tipo tecnico-scientifico, la realizzazione di questo satellite aveva anche il compito di accomunare molti dipartimenti del Politecnico di Torino nella progettazione dei moduli che compongono il satellite coinvolgendo docenti, ricercatori, studenti e dottorandi. Con l'esperienza maturata durante la progettazione del satellite PiCPoT si sono poste le basi per un secondo progetto, ancora in fase di sviluppo, il satellite AraMiS (Architettura Modulare per Satelliti).

Con il progetto di quest'ultimo, sono stati definiti dei moduli standard compatibili meccanicamente ed elettronicamente tra loro, che possono essere 'assemblati' nelle quantità e nelle modalità richieste dalla singola missione, determinando un abbattimento dei costi di progettazione e sviluppo non indifferente. L'obiettivo principale era di valutare la possibilità di utilizzare componenti commerciali COTS in un progetto spaziale al fine di ridurre notevolmente i costi.

Il satellite AraMiS sarà composto da due tipi di moduli standard (o *tiles*):

- ***Power Management tile:***Questi moduli si occupano di ricavare dall'energia solare l'energia elettrica necessaria per ricaricare le batterie tampone, di generare le alimentazioni per la circuiteria di bordo, di rilevare i latch-up e di fornire (assieme ad altri moduli di Power Management) l'assetto desiderato al satellite. I moduli di Power Management saranno montati sull'esterno del satellite in quanto su di essi risiedono i pannelli solari utilizzati per ricavare energia elettrica, inoltre fungono anche da struttura portante per il satellite. Grazie alla loro architettura è possibile decidere, in base allo stato di carica delle batterie e alla loro temperatura, quale batteria ricaricare e con quale pannello ricaricarla (solo i pannelli illuminati dal sole forniscono energia elettrica), infatti i moduli di Power Management sono progettati per interagire tra di loro tramite dei bus dati e di potenza. Sfruttando la presenza di più moduli di Power Management il satellite è in grado di ruotare in ogni direzione, utilizzando gli attuatori d'assetto (solenoidi e ruote d'inerzia). Ogni modulo comunica anche con il computer centrale (On Board Computer, OBC) per ricevere comandi sulla gestione dei sottosistemi o per trasmettere i dati di telemetria misurati localmente.

- ***Telecommunication tile:*** Questi moduli rendono possibile la comunicazione bidirezionale fra satellite e stazione di terra tramite due distinti canali di comunicazione. Un canale di comunicazione operante alla frequenza di 437MHz è stato pensato per permetterne l'utilizzo ai radioamatori che possono ricevere il beacon del satellite, mentre il secondo opera alla frequenza di 2,4GHz . La comunicazione è gestita da un microcontrollore a bordo di questo modulo che si occupa della gestione del protocollo di comunicazione, ma anche di interpretare i comandi di assetto provenienti dalla stazione di terra in modo da fornire ai singoli moduli di Power Management il comando relativo all'assetto che devono generare. Inoltre interagisce con le Power Management Tile attraverso l'invio di comandi di controllo e la ricezione dei dati di telemetria.

L'architettura è quindi altamente integrata, nonostante sia composta da moduli separati, e tutti gli scambi di informazione tra i diversi elementi del satellite avvengono per mezzo dell'On-Board Data Bus, il sottosistema trattato in questa tesi.

Il lavoro è iniziato studiando la documentazione di tesi precedenti che trattavano il sottosistema di comunicazione tollerante ai guasti. Nella prima parte sono stati analizzati i principali bus di comunicazione, in particolar modo quelli tolleranti ai guasti tipo il bus automotive e lo SpaceWire, il bus standard per applicazioni aerospaziali. Il passo successivo è stato studiare l'interfaccia per la trasmissione e la ricezione di dati e segnali da parte dei microcontrollori al bus e quindi l'analisi e la revisione della scheda della precedente tesi che conteneva 6 interfacce collegate insieme.

Avendo Aramis molti moduli il compito principale è quello di progettare l'interfaccia tra il processore contenuto in ognuno di questi moduli e la comunicazione con il canale. L'interfaccia è divisa in 3 parti in base alle loro funzioni. Si tratta di un trasmettitore, un ricevitore e la parte di accoppiamento.

Per la parte di trasmissione, viene usato un transistor NMOS, il segnale di ingresso proveniente dal microcontrollore attraversa il Gate del MOS, questo genera una corrente variabile nel percorso del Drain del MOS che scorre attraverso il lato primario del trasformatore e induce un segnale in tensione al lato secondario.

Per la parte di ricezione, viene usato un comparatore, il segnale di tensione proveniente dal bus sul secondario del trasformatore induce un segnale di tensione sul primario, poi l'amplificatore operazionale riceve il segnale dal trasformatore come input quindi genera un segnale di tensione in uscita che arriva al microcontrollore.

La parte di accoppiamento magnetico avviene tramite un trasformatore d'impulso che garantisce isolamento galvanico e permette l'uso della codifica RZI utilizzata nel protocollo IRDA tramite l'uso di segnali impulsivi.

Successivamente, si è passati alla realizzazione delle singole schede di interfaccia, prima realizzando lo schematico e poi sviluppando il PCB con il programma

Expedition della Menthor Graphics. Finito il montaggio dei componenti delle schedine sono stati fatti dei test elettrici di prova per verificarne l'effettivo funzionamento. Il test consisteva nel trasmettere un onda rettangolare con duty cycle del 10 % su un modulino e verificare l'effettiva ricezione sugli altri. In seguito è stata fatta la prova di tolleranza ai guasti verificando il funzionamento con diversi tipi di guasti provocati sul bus differenziale, come ad esempio l'interruzione o il corto-circuito di un filo.

La seconda parte si è sviluppata analizzando una scheda ibrida millefori contenente 4 connettori per microcontrollori e adattandola con vari collegamenti alle interfacce. Questo è stato fatto montando un connettore unico al quale sono collegati i pin di Tx e Rx della Uart dei 4 microcontrollori MSP430 della Texas Instruments ed i relativi segnali di alimentazione. Grazie a questi collegamenti e alle relative interfacce è possibile far comunicare fra loro le 4 unità.

Nella terza parte è stato analizzato ed adattato il codice C che permette la comunicazione tra i microcontrollori attraverso il protocollo di comunicazione utilizzando una struttura single-master.

Il primo esempio di comunicazione è stata la trasmissione di un byte da un unità ad un'altra. La seconda prova consisteva in una comunicazione bidirezionale sempre tramite la ricezione e ritrasmissione di un byte. L'ultima prova è stata la comunicazione bidirezionale in loop tra 2 unità sia in condizioni normali, sia in presenza di guasti provocati sul bus.

Il lavoro è suddiviso nei seguenti capitoli:

- **Capitolo 1:** introduzione e descrizione del progetto Aramis;
- **Capitolo 2:** descrizione degli standard esistenti di comunicazione in ambiente aerospaziale;
- **Capitolo 3:** descrizione dell'interfaccia di comunicazione bus in Aramis con il processore;
- **Capitolo 4:** realizzazione e collaudo del prototipo e test sulla comunicazione;
- **Capitolo 5:** conclusioni.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni si è assistito ad un continuo incremento della realizzazione di satelliti di piccole dimensioni, molti dei quali in ambito universitario. Varie università infatti hanno partecipato attivamente alla progettazione e realizzazione di nano-satelliti (peso compreso tra il kilogrammo e i dieci kilogrammi) e pico-satelliti (peso inferiore al kilogrammo).

Si è quindi arrivati alla definizione di una piattaforma internazionale specifica, che definisca degli standard progettuali ben precisi e permetta alle Università di realizzare il proprio satellite e mandarlo nello spazio a costi contenuti.

Nasce così lo standard per picosatelliti denominato Cubesat, sviluppato nel 2001 dal Professor Robert Twiggs, docente alla Stanford University USA. Questo standard prevede la realizzazione di satelliti di forma cubica di 10 cm di lato e con una massa di 1 Kg, la cui struttura è definita in funzione dell' adattamento al lanciatore POD (Picosatellite Orbital Deployer)[1].

Numerose università si sono occupate della progettazione di un satellite, anche per il forte valore didattico dell'esperienza, che permette agli studenti di confrontarsi con un problema progettuale serio e di sviluppare le proprie capacità e l'abilità di lavorare in team.

Anche il Politecnico di Torino ha aderito all'iniziativa internazionale di progettare un satellite universitario, prima con il progetto PicPot e attualmente con il progetto Aramis.

1.1 Progetto Pic Pot

Il progetto PicPot, acronimo di *PICcolo satellite del POlitecnico di Torino*, è un satellite di forma cubica di 13 cm di lato, con una massa di circa 2,5 Kg e un tempo di vita previsto di 90 giorni, progettato con la partecipazione di alcuni dipartimenti dell'ateneo, in particolare quello di Ingegneria Elettronica e Ingegneria Aerospaziale[6].
Gli obiettivi principali del progetto erano:

- Verificare il comportamento di componenti COTS (Commercial Off-The-Shelf) in ambiente spaziale;
- Acquisire dati relativi alle condizioni ambientali dell'orbita LEO (Low Earth Orbit) e trasmetterli alla Stazione di Terra;
- Scattare fotografie a bassa risoluzione dalla superficie terrestre attraverso tre fotocamere.

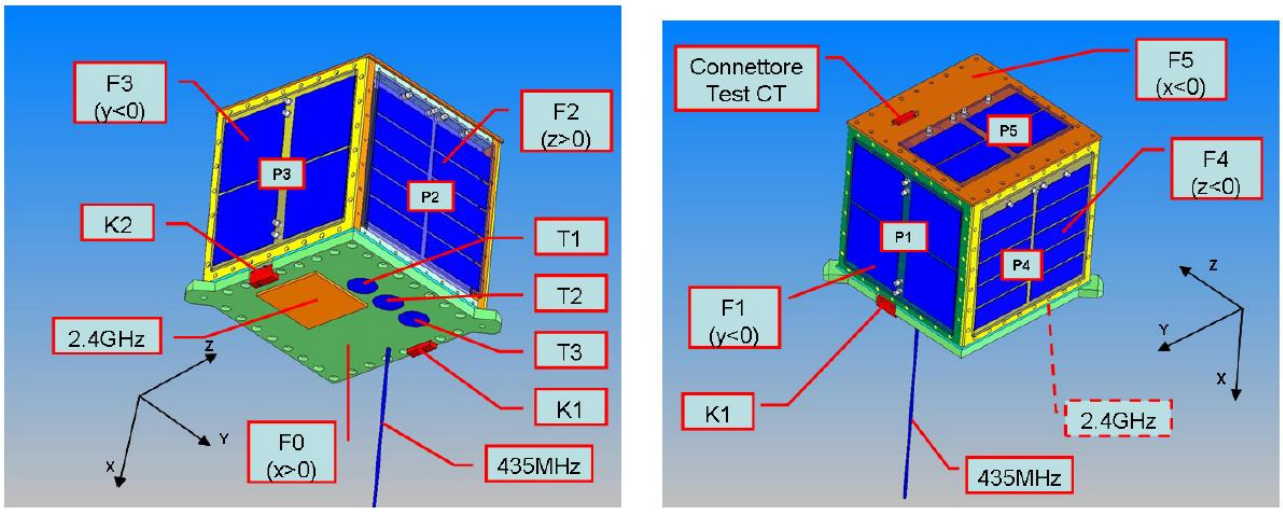


Figura 1.1 Vista esterna del satellite PiCPoT

La struttura esterna del satellite si presenta come un cubo composto da sei facce quadrate e ortogonali tra di loro in lega di alluminio tipo 5000 AlMn. Su di esse sono presenti cinque pannelli solari (P1, P2, P3, P4, P5), due antenne (alle frequenze di 437 MHz e 2.4GHz), tre fotocamere (T1, T2, T3), due kill-switch (K1, K2), e un connettore di test per testare la parte elettronica del satellite dopo che è stato montato.

La parte elettronica interna di PiCPoT è costituita dalle seguenti schede:

- **Power Switch**: genera le tensioni di alimentazione per tutti i sottosistemi del satellite, sceglie la batteria da utilizzare e gestisce gli anti latch-up;

- **Power Supply**: mantiene cariche le batterie a bordo del satellite e monitora lo stato dei pannelli solari e delle batterie;
- **ProcA e ProcB**: sono i due processori di bordo ed eseguono le stesse operazioni. Acquisiscono i dati dai sensori per le telemetrie e gestiscono il payload. Possono inoltre sia inviare che ricevere i dati a terra;
- **Payload**: acquisisce le immagini da una delle tre telecamere a bordo, il processore presente sulla scheda permette la conversione dal formato PAL al formato JPEG così da trasmettere l'informazione in formato compresso;
- **TxRx**: trasmette i comandi al satellite e riceve da esso i dati. Le frequenze utilizzate sono 437MHz per la trasmissione e 2,4 GHz per la ricezione.

Il progetto PicPot si è concluso con il lancio il 26 Luglio 2006 dalla base russa di Baiknaur (KAZ) su un razzo vettore Dnepr-LV, di derivazione militare ma a causa di un problema idraulico del razzo il lancio non andò a buon termine[2].

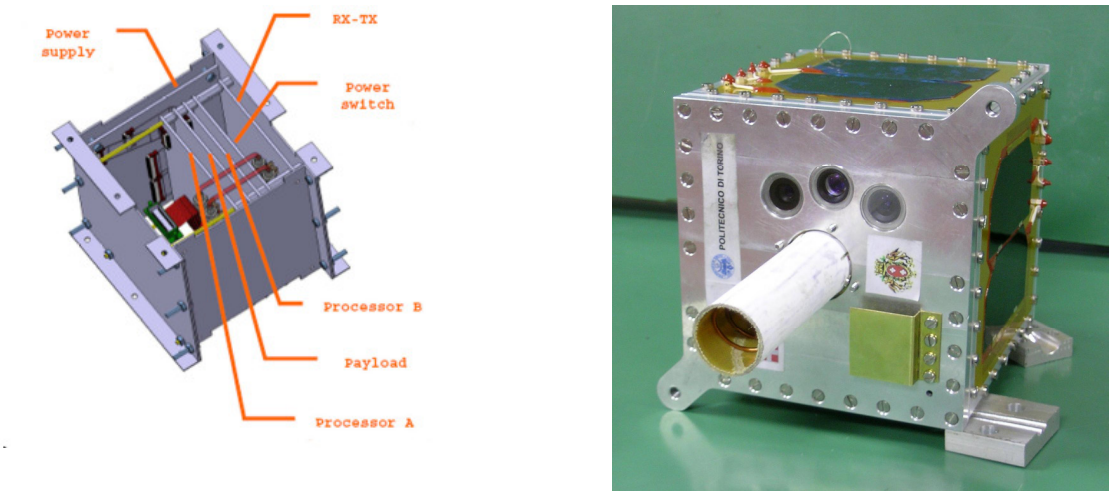


Figura 1.2 Satellite PicPot

1.2 Progetto Aramis

La successiva evoluzione del progetto PiCPoT è il progetto ARaMiS. L'obiettivo è la definizione di un'architettura modulare standard a basso costo per micro e nano satelliti di piccole dimensioni per orbite tra i 500 e gli 800 km di quota. Bisogna infatti considerare che la progettazione e realizzazione di satelliti è generalmente molto costosa e quindi non accessibile a tutti: lo sviluppo di un'architettura modulare, insieme all'utilizzo di componenti COTS contribuisce in maniera notevole ad abbattere i costi fissi di progettazione e sviluppo, i quali sarebbero condivisi tra numerose missioni.
I moduli o tile sono progettati per essere assemblati e utilizzati in modo semplice e flessibile ed hanno dimensioni fisiche e caratteristiche comuni.

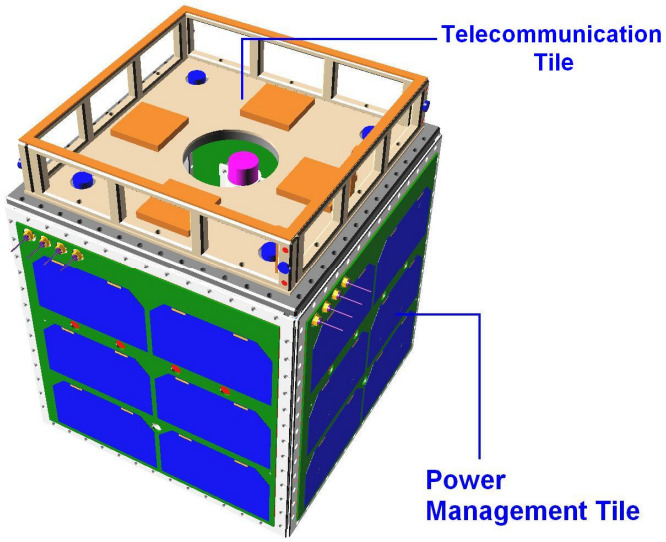


Figura 1.3 Vista esterna satellite Aramis

Sono definiti due tipi di moduli:

- Power Management Tile è costruita su di una lastra di alluminio di 1.5mm di spessore e presenta sulla faccia esterna i pannelli solari in GaAs a tripla giunzione che forniscono l'energia al satellite. Internamente sono fissate, tramite una struttura metallica, le due batterie agli ioni di litio che immagazzinano l'energia e la ruota di inerzia con il relativo motore brushless per il controllo di assetto. Al di sotto della struttura metallica, sono incollati con colle termoconduttive alla lastra di alluminio, il circuito stampato principale del modulo e il solenoide del controllo di assetto. Sul circuito stampato saranno presenti i circuiti switching di conversione dell'energia proveniente dai pannelli solari, i circuiti di carica delle batterie, i sensori per la determinazione dell'assetto del satellite e un

microcontrollore che gestisce il funzionamento di tutti i sottosistemi. Tutti i moduli di questo tipo prevedono un'interfaccia con bus di potenza e bus dati, i quali possono collegare tra loro i vari sottosistemi.

- Telecommunication Tile è invece costruita con una lastra di alluminio più spessa (5mm) e rappresenta il punto preferenziale di ancoraggio del satellite al vettore. Esternamente presenta le due antenne (a 437 MHz e 2.4 GHz) per la comunicazione verso terra e l'eventuale struttura di ancoraggio. Sulla faccia esterna inoltre c'è uno spazio centrale che può essere utilizzato per il posizionamento di una fotocamera di osservazione terrestre, essendo il modulo di telecommunication quello generalmente puntato verso terra. Sul lato interno trovano invece spazio i moduli a radiofrequenza nelle due bande e due OBC (On-Board Computer) ridondati. La Telecommunication Tile è il cuore dell'intero sistema e interagisce con le Power Management Tile attraverso l'invio dei comandi di controllo e la ricezione dei dati di telemetria. Gestisce inoltre le comunicazioni verso terra attraverso la ricezione e l'attuazione dei telecomandi e la trasmissione in downlink dei dati di telemetria o dei dati di un eventuale payload.

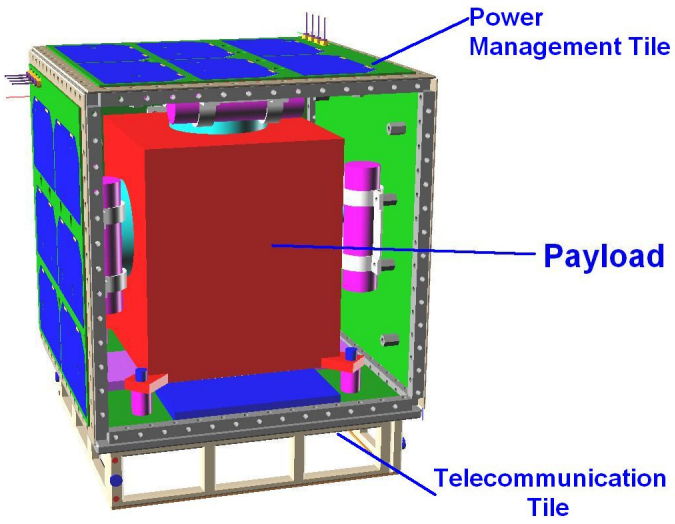


Figura 1.4 Vista interna satellite Aramis

Capitolo 2

Standard esistenti

2.1 SpaceWire

SpaceWire è un protocollo sviluppato e pubblicato nel gennaio del 2003 da ESA in collaborazione con le principali agenzie spaziali mondiali (NASA, JAXA e RKA) ed è attualmente utilizzato in diverse missioni spaziali ([8] Space Engineering *SpaceWire*).

Il protocollo è stato progettato per collegare insieme sensori ad alta velocità dati, unità di elaborazione, dispositivi di memoria e telemetria/telecomando a bordo di sotto-sistemi spaziali. Gli obiettivi che lo standard si propone sono:

- una elevata velocità di comunicazione (da 2 a 200 Mbits / s) bidirezionale *full-duplex*, quindi la possibilità di comunicare alla massima velocità in entrambe le direzioni;
- le reti possono essere costruite per soddisfare le particolari applicazioni utilizzando collegamenti punto-punto o topologie più complesse attraverso *switch di routing*;
- facilitare la costruzione di sistemi di comunicazione dati a bordo ad alta performance;
- ridurre i costi di integrazione del sistema;
- migliorare la compatibilità tra dati di dispositivi e sottosistemi;
- favorire il riutilizzo di dispositivi in diverse missioni spaziali.

L'utilizzo del protocollo SpaceWire assicura che il dispositivo sia compatibile a livello di componenti e sotto-sistemi. Unità di elaborazione, unità di memoria di massa e sistemi di telemetria *down-link* che utilizzano interfacce SpaceWire per una missione possono essere facilmente utilizzate per altre missioni, riducendo i costi di sviluppo e migliorando l'affidabilità.
Nella seguente figura 2.1 è possibile osservare un esempio di una rete di comunicazione a bordo di un veicolo aerospaziale.

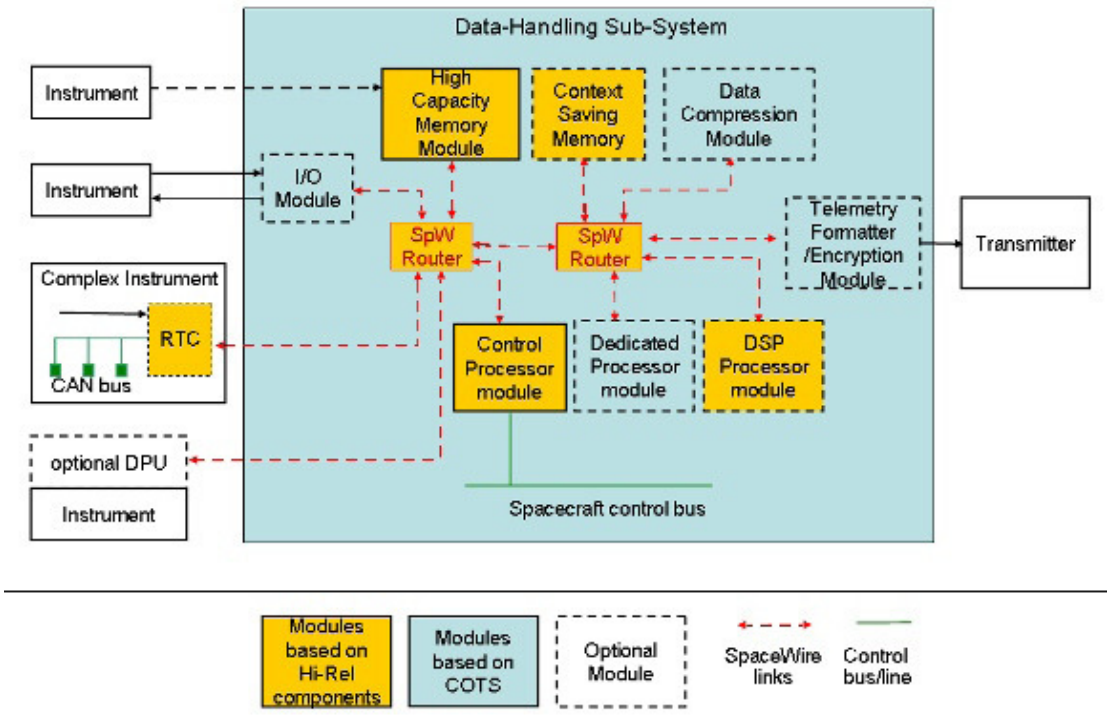


Figura 2.1 Esempio rete di comunicazione SpaceWire

Lo strumento in alto a sinistra è uno strumento ad alto *data-rate*. Un collegamento SpaceWire punto-punto viene utilizzato per il flusso di dati da questo strumento direttamente nel modulo di memoria ad alta capacità. Quest'ultimo è collegato con il resto dell'architettura con un *Router SpaceWire*. Dati provenienti da altri strumenti possono essere memorizzati nel modulo di memoria ad alta capacità utilizzando questo collegamento. I dati memorizzati nella memoria ad elevata velocità possono essere utilizzati attraverso questo collegamento per l'elaborazione, la compressione e l'invio al trasmettitore di telemetria *down-link*.

Un secondo strumento è connesso ad un modulo di I/O. Questo modulo viene utilizzato per la connessione a strumenti che non hanno connessioni dirette SpaceWire. Lo strumento passa i dati al modulo I / O, possibilmente su un bus dati parallelo. Il modulo I/O traduce questi dati in pacchetti SpaceWire e li invia alla destinazione richiesta sulla rete SpaceWire, questa potrebbe essere un modulo di memoria ad alta capacità, un modulo del processore o un modulo di compressione dei dati.

Il terzo strumento, il *complex instrument*, ha molti sotto-sistemi che devono essere controllati separatamente. Per fare ciò viene utilizzato ad esempio il bus CAN per controllare e raccogliere dati provenienti dai vari sotto-sistemi. Un Remote Terminal Interface (RTI) è utilizzato per fornire un ponte tra SpaceWire e il bus locale (ad esempio CAN). Il terminale remoto del computer è stato progettato specificamente per supportare questa funzione.

Lo standard SpaceWire definisce i seguenti livelli di protocollo normativo:

- **Physical Level**: definisce i connettori e cavi;
- **Signal Level**: definisce la codifica del segnale, i livelli di tensione, i margini di rumore, le specifiche EMC e le velocità dei segnali;
- **Character Level**: definisce i caratteri di dati e di controllo utilizzati per gestire il flusso dei dati attraverso un collegamento;
- **Exchange Level**: definisce il protocollo per l'inizializzazione di collegamento, controllo di flusso e rilevazione d'errore;
- **Packet Level**: definisce come i dati devono essere trasmessi tramite un link spacewire e come suddividerli in pacchetti;
- **Network Level**: definisce la struttura di una rete Spacewire e il modo in cui i pacchetti sono trasferiti da un nodo sorgente ad un nodo di destinazione attraverso una rete. Definisce la modalità di gestione degli errori a livello collegamento e rete.

SpaceWire si basa su due standard esistenti commerciali, IEEE 1355-1995 e Low Voltage Differential Signal (LVDS), che sono stati combinati e adattati per essere utilizzati a bordo di veicoli spaziali.

2.1.1 Descrizione del protocollo

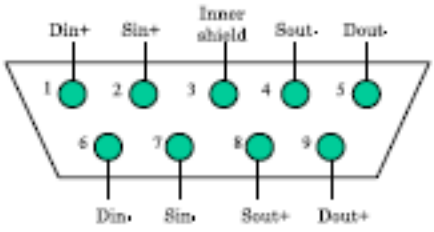


Figura 2.2 Connettore SpaceWire

Il *livello fisico* dello standard si occupa della definizione dei cablaggi, dei connettori e delle specifiche EMC. Il cablaggio è composto da quattro doppini di fili intrecciati schermati fra di loro ed un altro schermo totale. Per ottenere una trasmissione dati ad alta velocità il cavo deve avere impedenza caratteristica adattata a impedenza di terminazione di linea, bassa distorsione, attenuazione del segnale e basso cross-talk. Il connettore deve avere otto pin di segnale più un pin di terminazione schermo.

Il *signal level* è caratterizzato dalla tecnica Low Voltage Differential Signal o LVDS. Quest'ultima utilizza segnali bilanciati per fornire un interconnessione ad alta velocità utilizzando un *low voltage swing* (tipicamente 350mV). Il segnale bilanciato o differenziale fornisce un adeguato margine di rumore e permette l'utilizzo di basse

tensioni in sistemi pratici. LVDS è appropriato per collegamenti tra schede su una distanza di 10 m o più. Nella seguente figura è mostrato un tipico driver e receiver LVDS.

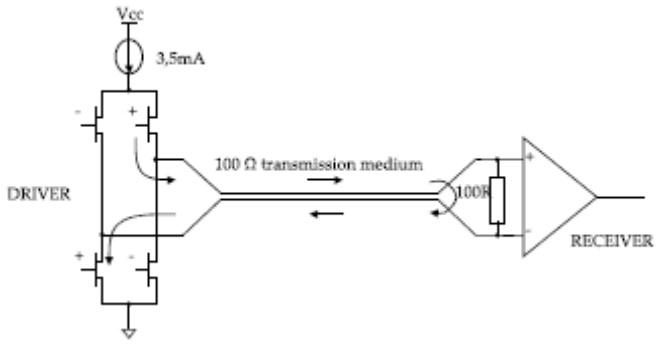


Figura 2.3 Esempio segnale LVDS

La tecnologia LVDS utilizza per la segnalazione un loop di corrente. Una fonte costante di corrente di circa 3.5 mA esce fuori dal driver, lungo il mezzo di trasmissione, attraversa la resistenza di terminazione di 100 ohm e ritorna al driver. Due coppie di interruttori (transistor) del driver controllano la direzione del flusso di corrente attraverso la resistenza di terminazione. A seconda del livello logico da trasmettere, la corrente sarà iniettata in uno dei due conduttori e chiusa verso la tensione di riferimento attraverso l'altro. Nel caso in cui venga chiusa la coppia di transistor "+" e aperta la coppia "-", la corrente scorrerà nella direzione indicata dalle frecce. Nel caso in cui sia la coppia "-" a essere attivata e la coppia "+" venga inibita, la corrente scorrerà nella direzione inversa.

Il receiver LVDS avrà un alta impedenza in ingresso in modo da far confluire tutta la corrente nella resistenza di terminazione di 100 ohm che genererà sull'ingresso del ricevitore una tensione differenziale di ±350mV. Questa tensione verrà tradotta in uno stato logico del segnale secondo i livelli di tensione definiti nello standard.

SpaceWire utilizza una codifica *Data-Strobe* (DS). Si tratta di uno schema di codifica che utilizza due segnali Data e Strobe ed il segnale di clock viene semplicemente recuperato facendo una XOR tra i due segnali. Il segnale Data viene trasmesso direttamente ed il segnale Strobe effettua una transizione ogni volta che il segnale Data non varia rispetto al bit precedente. Viene usata la codifica DS per migliorare la tolleranza di skew di quasi 1 bit a ciclo rispetto ai 0.5 di una semplice codifica di clock. Un canale bidirezionale SpaceWire è quindi composto da 2 segnali (Data e Strobe) per ogni direzione di comunicazione entrambi in tecnologia LVDS, per un totale di 8 conduttori raggruppati in 4 coppie. Un collegamento SpaceWire utilizza generalmente 4 *twisted-pair* schermati singolarmente raggruppati in un cavo ulteriormente schermato.

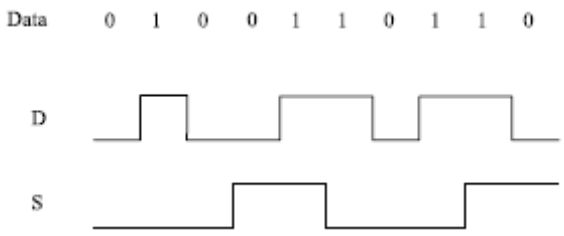


Figura 2.4 Codifica Data-Strobe

Il *livello carattere* contiene due tipi:

- caratteri **Data** formati da 8 bit, contenente un bit di parità, un *Data-Control flag* e gli 8 bit di dati. E' impostato per produrre parità dispari ed il Data Control flag è impostato a 0 per indicare che il carattere corrente è un carattere Data.
- caratteri **Control** hanno 2 bit di controllo. Ogni carattere Control è formato da un bit di parità, un data control flag e due bit di controllo. Il data control flag è settato a 1 per indicare che il carattere corrente è un carattere Control. Uno dei quattro caratteri di controllo possibile è il codice di *escape* (ESC). Questo può essere usato per formare i codici di controllo più lunghi. Un codice di controllo più lungo è specificato che è il codice NULL. Quest'ultimo è formato da un carattere ESC seguito da un FCT e viene trasmesso sul canale qualora non vi siano altre trasmissioni in corso per mantenere il collegamento attivo e rendere possibile il rilevamento di un'eventuale disconnessione.

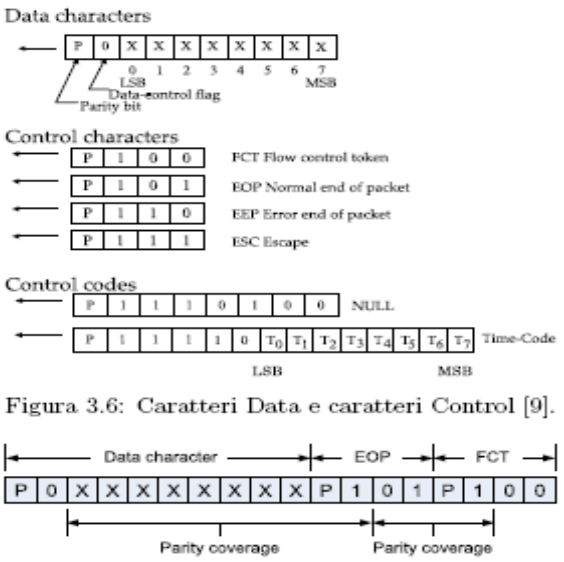


Figura 3.6: Caratteri Data e caratteri Control [9].

Figura 2.5 Formato carattere Data e Control

Il *livello Exchange* del protocollo provvede ai seguenti servizi:

- **Inizializzazione**: in seguito ad un reset, la trasmissione sul canale viene disattivata per un periodo di tempo. Dopo viene attivata la sequenza di inizializzazione per stabilire che entrambi i lati del collegamento possano trasmettere e ricevere i dati correttamente. Ogni lato del collegamento invia NULL, attende di ricevere un valore NULL, quindi invia FCT (*Flow Control Token*) e attende di ricevere un FCT. Dal momento che da un'interfaccia di collegamento non è possibile inviare FCT fino a quando non ha ricevuto un NULL, il ricevimento di uno o più NULL seguito dal ricevimento di un FCT significa che l'altra estremità del collegamento ha ricevuto NULL con successo e che la completa connessione è stata raggiunta.
- **Controllo di flusso**: solo ad un trasmettitore è consentita la trasmissione di caratteri data se c'è spazio nell'*host system buffer* di ricezione per loro. Il system host indica che c'è spazio per otto o più data caratteri richiedendo al collegamento trasmettitore di inviare un controllo di flusso token (FCT). L'FCT è ricevuto all'altra estremità del collegamento abilitando il trasmettitore ad inviare fino a otto o più FCT. Se non vi è più spazio nel buffer di ricezione ospite allora FCT multiple possono essere inviate, una per ogni otto posti nel buffer di ricezione. Di conseguenza, se si ricevono più FCT, significa che vi è una corrispondente quantità di spazio disponibile nel buffer del ricevitore ad esempio, quattro FCT corrispondono allo spazio per 32 caratteri di dati.
- **Rilevamento di errori di disconnessione**: la disconnessione di un collegamento avviene quando non viene ricevuto alcun bit per un determinato perido di tempo (850 ns). Una volta che l'errore è stato rilevato si cerca una soluzione per risolverlo.
- **Rilevamento errore di parità**: quando viene rilevato un errore di parità in un carattere, il sistema procede al *link error recovery*.
- **Link error discovery**: in seguito ad un errore o ad un reset della connessione, il collegamento tenta una ri-sincronizzazione tramite un protocollo a scambio di silenzio (vedi figura). Un lato del collegamento che viene resettato o trova un errore causa la cessazione della trasmissione, questo verrà interpretato dall'altra estremità del collegamento come una condizione di disconnessione. Dopo un'attesa di 6.4 µs, il sistema abilita la ricezione e dopo altri 12.8 µs viene abilitata anche la trasmissione. Questi periodi di tempo sono sufficienti per assicurare che i ricevitori ad entrambe le estremità del collegamento siano pronti a ricevere caratteri prima che una delle estremità inizi la trasmissione. Le due estremità del collegamento provvedono all'*handshaking* di inizializzazione con codici NULL e FCT.

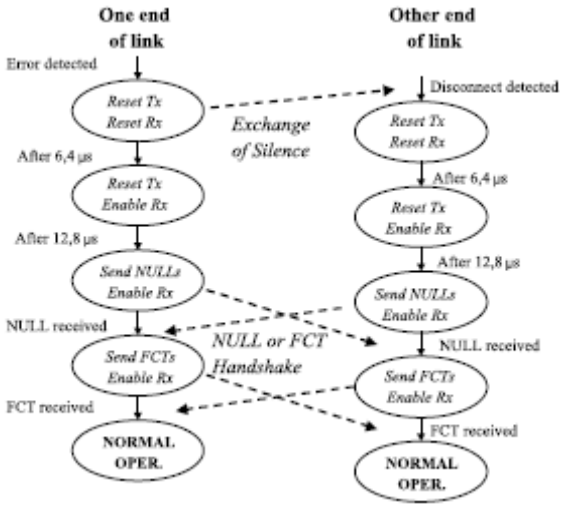


Figura 2.6 Schema di handshaking e error recovery

Il *livello packet* definisce la modalità di impacchettamento per il trasferimento dalla sorgente alla destinazione. Il formato del pacchetto illustrato in figura è diviso in tre campi. Il "*Destination Address*" è un elenco di uno o più data character che rappresentano l'identità destinazione. Questa lista di caratteri di dati rappresenta il codice identificativo del nodo di destinazione o del percorso che il pacchetto farà per arrivare al nodo di destinazione. Il "*Cargo*" è il dato che deve essere trasferito dalla sorgente alla destinazione. La "*End of Packet Marker*" è usato per indicare la fine di un pacchetto. Due tipi di pacchetto sono definiti:

- **EOP** Normal End of packet marker, indica la fine del pacchetto;
- **EEP** Error End of packet marker, indica che il pacchetto è stato terminato prematuramente a causa di un errore di collegamento.

Poichè non viene definito un indicatore di inizio pacchetto, il primo carattere Data immediatamente successivo ad un indicatore di fine pacchetto è da considerarsi l'inizio del pacchetto successivo.

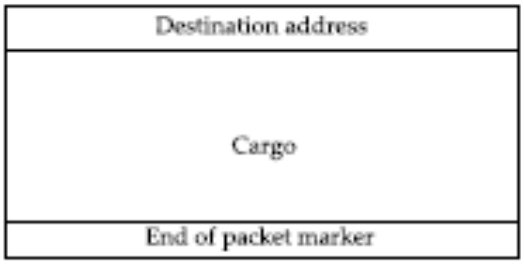


Figura 2.7 Formato del pacchetto

Il *livello di rete* definisce una rete SpaceWire e descrive gli elementi che la compongono, spiega come i pacchetti vengono trasferiti attraverso una rete SpaceWire e come recupera dagli errori.
Una rete SpaceWire è costituito da un numero di nodi interconnessi da SpaceWire routing switch. Questi nodi sono le sorgenti e le destinazioni dei pacchetti e forniscono un interfaccia ai sistema di applicazione. I nodi SpaceWire possono essere direttamente collegati tra loro tramite Link SpaceWire o possono essere interconnessi attraverso SpaceWire routing switch che utilizzano collegamenti SpaceWire per effettuare una connessione tra il nodo e lo switch di routing. Un interruttore SpaceWire routing ha altri collegamenti ad interfacce che sono collegate tra loro all'interno di una matrice che permette a qualsiasi Ingresso di passare i pacchetti che riceve a ogni uscita per la ri-trasmissione.

2.2 CAN

Il protocollo CAN (Controller Area Network) è un bus seriale di comunicazione digitale di tipo "*broadcast*". Esso permette il controllo real-time distribuito con un livello di sicurezza molto elevato. E' stato introdotto dalla *Bosch* nei primi anni 80 per applicazioni automobilistiche, per consentire cioè la comunicazione fra i dispositivi elettronici intelligenti montati su un autoveicolo, ma si è diffuso ormai in molti settori dell'industria ([7] *An Overview of Controller Area Network (CAN)*).
Le caratteristiche principali di questo protocollo sono:

- **Semplicità e flessibilità del cablaggio**: CAN è un bus seriale tipicamente implementato su un doppino intrecciato (schermato o meno a seconda delle esigenze). I nodi non hanno un indirizzo che li identifichi e possono quindi essere aggiunti o rimossi senza dover riorganizzare il sistema o una sua parte.
- **Alta immunità ai disturbi**: lo standard ISO11898 raccomanda che i chips di interfaccia possano continuare a comunicare anche in condizioni estreme, come l'interruzione di uno dei due fili o il cortocircuito di uno di essi con massa o con l'alimentazione.
- **Elevata affidabilità:** la rilevazione degli errori e la richiesta di ritrasmissione viene gestita direttamente dall'hardware con cinque diversi metodi (due a livello di bit e tre a livello di messaggio).
- **Confinamento degli errori**: ciascun nodo è in grado di rilevare il proprio malfunzionamento e di autoescludersi dal bus se questo è permanente. Questo è uno dei meccanismi che consentono alla tecnologia CAN di mantenere la rigidità delle temporizzazioni, impedendo che un solo nodo metta in crisi l'intero sistema.
- **Maturità dello standard**: la larga diffusione del protocollo CAN in questi venti anni ha determinato un'ampia disponibilità di chip rice-trasmettitori , di microcontrollori che integrano porte CAN, di tools di sviluppo, oltre che una sensibile diminuzione del costo di questi sistemi. Questo è molto importante per far sì che uno standard si affermi nell'ambito industriale.

Per garantire la trasparenza e la flessibilità CAN è stata suddivisa in diversi livelli secondo la norma ISO / OSI.
Modello di riferimento:

- Il livello di Data Link: è composto dai sottolivelli Logical Link Control (LLC) e Medium Access Control (MAC);
- Il livello fisico.

Gli scopi del sottosistema LLC sono:

- fornire servizi per il trasferimento dei dati e per la richiesta di dati a distanza;
- decidere quali messaggi ricevuti dal sottolivello LLC sono in realtà da accettare;

- gestione del recupero e sovraccarico di notifiche.

Lo scopo principale del sottolivello MAC è definire il protocollo di trasferimento, ossia controllando il *Framing*, eseguendo arbitrazione, controllo d'errore, segnalazione e contenimento dei guasti. Il sottolivello MAC decide se il bus è libero di iniziare una nuova trasmissione o se la ricezione è appena agli inizi. Anche alcune generali caratteristiche di temporizzazione dei bit sono considerate come parte del sottolivello MAC.

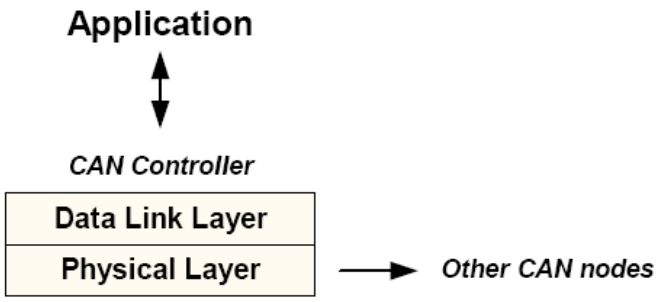


Figura 2.8 Livello fisico CAN

2.2.1 CAN Physical Layer

Rappresentazione bit

In genere il supporto fisico utilizzato dal sistema CAN è una coppia di fili differenziale. Questo rende la trasmissione del segnale molto attendibile nonostante i livelli bassi di segnale e gli errori di modo comune. I due fili sono chiamati CAN_H e CAN_L e sono terminati usando resistori da 120 ohm. E 'anche tipico usare cavi *twisted pair* per ridurre le interferenze elettromagnetiche.

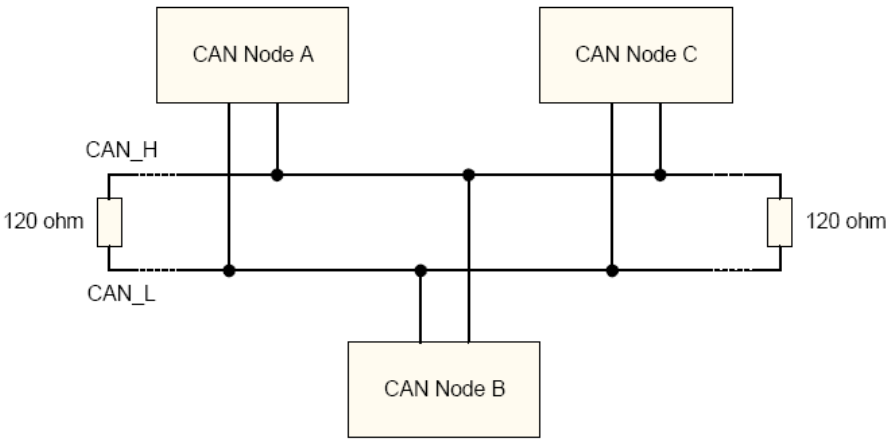


Figura 2.9 Esempio cablaggio CAN

CAN utilizza una topologia a bus, economica, che permette la connessione tra nodi in modo semplice, ed è meno soggetta a guasti di rete. Ad esempio, in una configurazione di rete a stella, l'intera rete dipende da un hub centrale. Se questo hub si guasta, l'intera rete smette di funzionare. Una rete *token ring* non dispone di un hub centrale, ma se uno qualsiasi dei singoli nodi nella rete fallisce, l'anello è rotto, ed i nodi superstiti sulla rete non possono più comunicare tra loro.

Codifica Non-Return to Zero vs. Manchester Bit Encoding

Ci sono vari modi per codificare i bit nei sistemi digitali. Descriviamone due qui, Non-Return to Zero (NRZ) e Manchester. Entrambi i metodi hanno vantaggi e svantaggi.

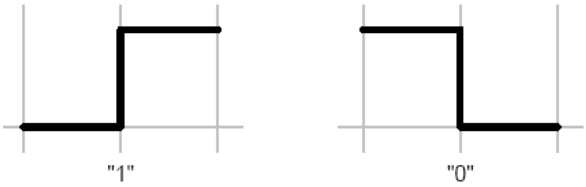


Figura 2.10 Codifica Manchester

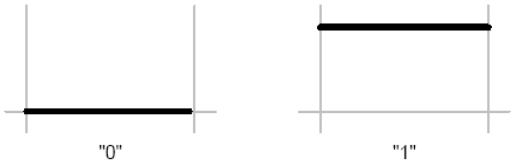


Figura 2.11 Codifica NRZ

La codifica Manchester rappresenta i bit come le transizioni tra '0' e '1' o '1' a '0'. Anche se un frame ha una stringa di '1' o '0' la transizione è sempre necessaria. Lo schema di codifica Manchester è molto utile in sistemi di comunicazione asincrona in quanto non vi è sempre una transizione del segnale per la sincronizzazione dei bit. Il ricevitore può sempre dire quando ha raggiunto il limite o la fine di un segnale. Lo svantaggio principale di questa codifica è che richiede più banda in quanto ogni bit deve avere due colpi da codificare.

La codifica NRZ non richiede transizioni del segnale per rappresentare ogni bit. Il segnale rimane '0' o '1' per l'intera fascia di clock. Se un frame ha una serie di '1' o '0', il segnale rimarrà costante per il tempo necessario. Lo svantaggio di NRZ è che non esiste un modo semplice per dire dove ogni bit inizia o finisce quando ci sono più di due '1' o '0' di fila. L'unico modo per sapere dove un bit inizia o termina è avere

per il ricevitore un clock sorgente che è identico al trasmettitore in modo che possa decifrare il flusso di bit. Questa si chiama comunicazione sincrona.

Gli oscillatori utilizzati con i controller CAN sono sufficientemente accurati per fare un gran numero di check point previsti che nella codifica Manchester sono inutili. Eliminando le transizioni non necessarie richieste dalla codifica Manchester un sistema CAN è in grado di comunicare a quasi il doppio della velocità per una determinata frequenza di clock. Tuttavia, è molto difficile mantenere due oscillatori esattamente sincronizzate per molto tempo alle velocità utilizzate dalla CAN. Per superare questo, CAN utilizza una tecnica chiamata *bit stuffing*.

Bit stuffing

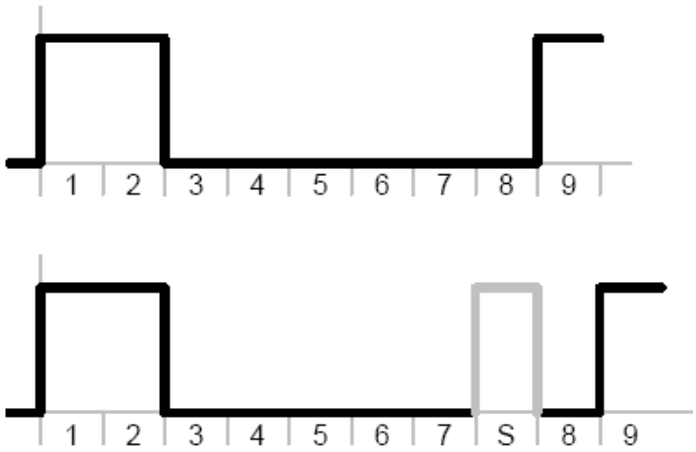


Figura 2.12 Bit stuffing

Poiché i segnali NRZ potrebbero rimanere costanti per un lungo periodo di tempo (lunga serie di '1' o '0'), è possibile che i singoli oscillatori della rete (uno per nodo) possano non essere sincroni. Il protocollo CAN supera questo inserendo un bit di transizione ogni cinque bit uguali in una riga. Questo segnale di transizione è detto un *bit stuff*. I ricevitori utilizzano il bit stuff per sincronizzare i propri clock. Ogni volta che il ricevitore rileva cinque '0' o cinque '1' in una riga, viene automaticamente invertito il bit successivo.

Stati Recessive e Dominant

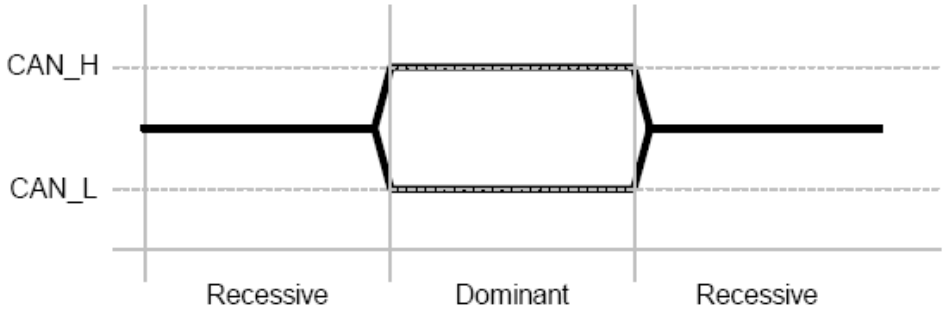


Figura 2.13 Stati Dominant e Recessive

Nel descrivere i segnali CAN è comune usare i termini *recessive* e *dominant* per descrivere lo stato del bus. Per un bus a due fili lo stato del bus recessive si verifica quando il CAN_L e le linee CAN_H sono allo stesso potenziale (CAN_L = CAN_H = 2,5 V), e lo stato del bus dominant si verifica quando vi è una differenza di potenziale (CAN_L =1.5V e 3.5V CAN_H =). Il bus CAN rimane nello stato recessive quando è inattivo. E' importante fare la distinzione tra '1' e '0' e stati del bus recessive e dominant. '1' e '0' sono utili per rappresentare i dati utilizzando il sistema binario numerico, ma non ti dicono nulla sulla situazione del bus. In realtà, CAN definisce '0' come uno stato dominant e '1' come uno stato recessive. Questo è un po' contro intuitivo. Ma il concetto di stato bus dominant e recessive è un concetto particolarmente importante quando si discute di arbitraggio del bus e campi di controllo CAN.

Bit timing e sincronizzazione

Il protocollo CAN utilizza una trasmissione dati di tipo "sincrono": due eventi che sono coordinati nel tempo. Per i sistemi CAN ciò significa che ogni nodo invia e riceve utilizzando la stessa frequenza di clock, e tutte le frequenze di clock della rete si basano su un unico punto di riferimento. Questo rende la trasmissione dei dati più efficiente, ma è difficile tenere i due clock sincronizzati nel tempo senza un qualche tipo di segnale di riferimento. I clock comunemente perdono la loro sincronizzazione a causa della deriva dell'oscillatore, ritardi di propagazione o errori di fase.

Nodi possono utilizzare due diversi metodi per sincronizzare i propri clock, *Hard Synchronization* and *Resynchronization Hard Synchronization*. Hard Synchronization si verifica solo una volta durante una trasmissione del messaggio, all'inizio di un nuovo messaggio frame. Prima che un frame inizi, il bus CAN è in uno stato recessive (idle). Il primo bit di un frame è una *Start of Frame* bit che è sempre trasmesso in modo dominante. Ogni nodo sincronizza il proprio clock usando la transizione creata

da questo Start Frame bit. I clock non sono in grado di rimanere sincronizzati attraverso l'intero frame quindi deve continuamente risincronizzare.

La risincronizzazione si verifica ogni volta che il bus effettua una transizione da recessive a dominant. Se c'è una serie di '0' o '1' i nodi CAN possono dipendere dalla transizione creata dal bit stuff per risincronizzare i loro clock.

Ogni sistema CAN è configurato con un bit rate. Quest'ultimo è il numero di bit che passano in un dato punto di una rete per secondo. Il bit rate (o data rate) è misurato in migliaia di bit (kilobit o Kbps) o milioni di bit (megabit o Mbps) al secondo. Il bit rate nominale è il numero di bit al secondo trasmessi da un trasmettitore ideale senza risincronizzazione. Ogni nodo sul bus CAN deve avere lo stesso bit rate nominale. La deriva dell'oscillatore e altri problemi possono rendere il bit rate nominale diverso dal bit rate attuale, rendendo una sincronizzazione del traffico dei messaggi necessaria. Il bit time nominale è la quantità di tempo necessaria per trasmettere un singolo bit in rete. Sistemi CAN usano il bit time per stabilire se il bus è in un stato recessive o dominant. Per fare questo, il bit time nominale è spezzato in segmenti. Ogni segmento è diviso in unità chiamate *time quanta*.

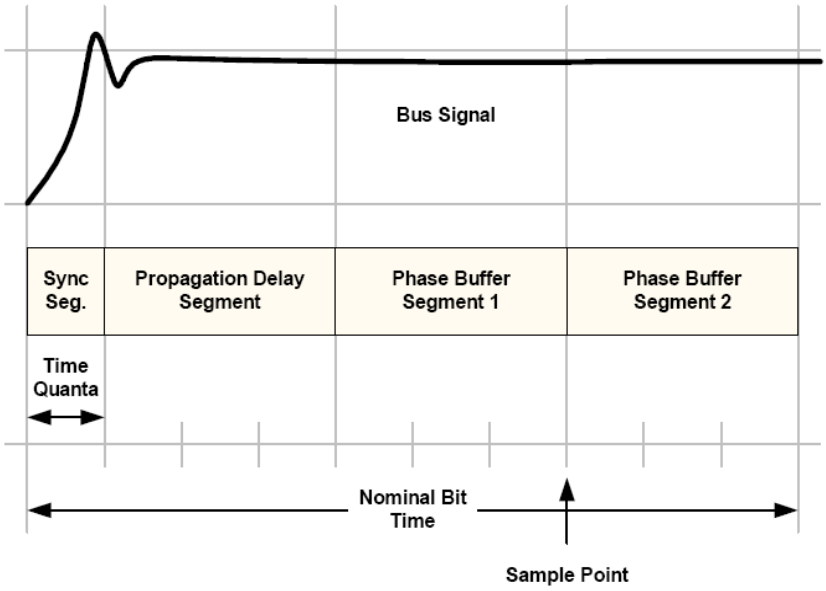


Figura 2.14 Bit Time Segment

I segmenti del bit time nominale sono:

- **Synchronization Segment**: Il segmento di sincronizzazione è il primo segmento del bit time nominale. Qui è dove ci si aspetta la transizione del segnale. Il segmento di sincronizzazione è sempre fisso ad un time quanta.
- **Propagation Delay Segment**: Un periodo di tempo è necessario per i segnali a viaggiare da un capo del bus all'altro e viceversa. Un periodo di tempo è necessario anche per i componenti elettronici a reagire. Il ritardo di propagazione del segmento esiste per compensare questi ritardi.

- **Fase Buffer Segment 1 & 2**: Le fasi di buffer esistono per compensare la deriva dell'oscillatore e altri problemi con i tempi. Qui è dove ha luogo la risincronizzazione. I progettisti di sistemi CAN possono fare il segmento di fase 1 più lungo o il segmento di fase 2 più breve per compensare gli errori di timing. I due segmenti sono chiamati Synchronization Jump Width.
- **Sample point**: il punto di campionamento è dove lo stato del bus è misurato. Esso si verifica sempre tra Segmenti di fase 1 e 2. Il Punto campione può essere impostato per essere semplice o multiplo. Il controller CAN campiona il bus una volta in modalità singolo campione e tre volte in modalità multipla. In modalità multipla di campionamento il valore è determinato dalla decisione a maggioranza. Cioè, se tutti e tre i campioni sono dominant, o se due dei tre sono dominant, il campione viene registrato come dominant.

Bit rate e lunghezza bus

La lunghezza del bus si riferisce alla lunghezza effettiva del cablaggio di una rete. Per un Controller Area Network, la massima lunghezza del bus possibile dipende dal bit rate o dal data rate. Ogni sistema CAN in commercio ha la lunghezza del bus dipendente dalla velocità di bit. Questo non è un limite fissato dal protocollo CAN, ma dalle leggi della fisica. Un ciclo di clock è necessario ad un segnale per muoversi da un capo all'altro del bus e ritornare di nuovo prima della trasmissione del prossimo segnale. I circuiti elettronici necessitano di questo tempo per trasmettere e ricevere i segnali. Questo periodo di tempo è chiamato Propagation Delay. I progettisti del sistema CAN devono tenere a mente che le trasmissioni tendono a diventare più affidabili con una velocità più lenta ed una lunghezza del bus più corta.

2.2.2 CAN Data Link Layer

Arbritazione del bus

Il livello Data Link svolge una funzione chiamata Medium Access Control (MAC) per evitare i conflitti sulla rete. Se due o più trasmettitori cercano di inviare messaggi attraverso la rete nello stesso tempo, MAC farà in modo che a ciascuno di essi sia data la possibilità di trasmettere uno alla volta, in modo che i messaggi non interferiscano tra loro. Il metodo di controllo di accesso medio che utilizza una rete ha un impatto significativo sulle prestazioni. I metodi di controllo di accesso sono di due tipi, determinato e casuale.

Con il controllo di accesso determinato, il diritto di accesso al bus è definito prima che un nodo tenti di accedere al bus, garantendo che non si verifichino conflitti. Questo tecnica richiede un organismo centrale per gestire l'accesso di rete, o di un accordo decentrato tra i nodi (come un *token passing*). Metodi di accesso con controllo centralizzato sono più vulnerabili ai guasti del sistema. Una delle ragioni è

che se l'ente centrale non lavora quindi tutta la rete fallisce. Metodi di accesso decentrati sono più complessi di quelli centralizzati. Diventa anche difficile da assegnare dinamicamente la priorità ai nodi utilizzando metodi decentrati.

Con il controllo di accesso casuale ogni nodo può accedere al bus, non appena è disponibile. La maggior parte dei metodi di controllo ad accesso casuale si basano su "*Carrier-Sense Multiple Access*" (CSMA). Con questa tecnica tutti i nodi monitorano la rete e attendono che diventino disponibili. Una volta che la rete è idle tutti i nodi che hanno un messaggio da trasmettere tenteranno di accedere alla rete, nello stesso tempo. Naturalmente solo un nodo alla volta è in grado di trasmettere, quindi un metodo deve essere trovato per ordinare quale nodo ha la priorità.

Se la rete CSMA è impostata per limitare o prevenire le collisioni tra i messaggi, è chiamata *Collision Avoidance*. Se la rete è impostata per limitare i conflitti di messaggio, ma poi interviene a rilevare e ripulire questi conflitti, si chiama *Collision Detection*. Con il metodo Collision Detection ogni nodo controlla per vedere se il bus è libero prima di trasmettere. Se due o più nodi trasmettono contemporaneamente, i nodi che stanno trasmettendo sono in conflitto, si interrompe la trasmissione, e quindi si prova a ri-trasmettere dopo un tempo casuale. Il problema principale con il metodo Collision Detection si verifica quando c'è molta contesa sulla rete. I frame sono costantemente interrotti e ritrasmessi con spreco di banda e creazione di lunghi ritardi.

Non-Destructive Bit-Wise Arbitration

I sistemi "Controller Area Network" utilizzano "Carrier-Sense Multiple Access con Collision Avoidance" (CSMA / CA). Il protocollo CAN chiama questo "Non-Destructive Bit Wise Arbitration". Non è centralizzato, e si concede l'accesso ai nodi del bus in base alla priorità.

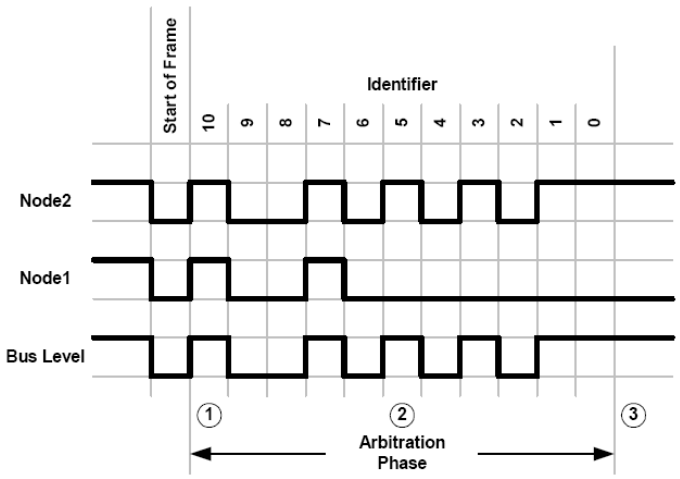


Figura 2.15 Esempio di Bit Wise Arbitration

Il protocollo CAN controlla il traffico del bus, consentendo l'accesso al bus ai messaggi ad alta priorità rispetto a quelli con priorità inferiore. Ogni messaggio inizia

con il campo arbitrazione, che identifica il messaggio e ne determina la priorità. Ogni nodo attribuisce questo campo ad un messaggio che intende trasmettere. Ogni volta che un nodo trasmette il campo arbitrazione, ascolta il bus al tempo stesso. Se il nodo che sta trasmettendo un bit recessive rileva un bit dominant sul bus, si interrompe automaticamente la trasmissione e diventa un ricevitore del messaggio dominant. Una volta che il messaggio dominant è stato trasmesso e il bus diventa disponibile, i nodi meno dominant sono in grado di provare ancora. Il risultato è che la larghezza di banda non è sprecata, ci sono metodi che un progettista può utilizzare per prevedere il ritardo più lungo possibile prima che un qualsiasi messaggio venga recapitato. Si noti che ogni nodo della rete vedrà ogni messaggio trasmesso sul bus. Nella maggior parte del tempo ogni nodo ignorerà la maggior parte dei messaggi che vede.

Formato del frame

Un frame è un pacchetto di informazioni che contiene un messaggio completo. I sistemi CAN hanno quattro tipi di frame:

- **Data Frame**: Un messaggio standard utilizzato per trasmettere dati attraverso la rete.
- **Remote Frame**: Un messaggio inviato da un ricevitore per richiedere dati a un altro nodo sulla rete.
- **Error Frame**: Un messaggio inviato da un ricevitore per distruggere un frame che contiene errori. L'errore Frame dice al trasmettitore di inviare nuovamente il messaggio.
- **Overload Frame**:E'simile ad un Error Frame. Un ricevitore in genere invia un Overload Frame per chiedere ad un trasmettitore di ritardare l'invio del prossimo messaggio.

Data Frame

Sistemi CAN utilizzano Data Frame per trasmettere dati attraverso la rete. Un frame di dati contiene un identificatore e vari campi di informazioni di controllo, e può contenere fino a otto byte di dati. Abbiamo due versioni dei Data Frame, il formato di Base e il formato esteso. La Robert Bosch Corporation ha introdotto il formato esteso nei primi anni '90 nelle specifiche CAN 2.0B. Quando i sistemi CAN sono stati sviluppati per i grandi sistemi con alto traffico di messaggi, come per gli autobus e camion, l'attuale formato di Base era inadeguato. Il numero di messaggi creati da trasmettitori in rete è stato maggiore del numero di codici ID possibile che il sistema CAN potrebbe assegnare loro per assicurare che ogni messaggio sia unico. Con l'aggiunta di un campo identificatore più lungo, 29 bit, il sistema è in grado di creare fino a 512 milioni di messaggi diversi unici e di priorità.

Formato base Data frame

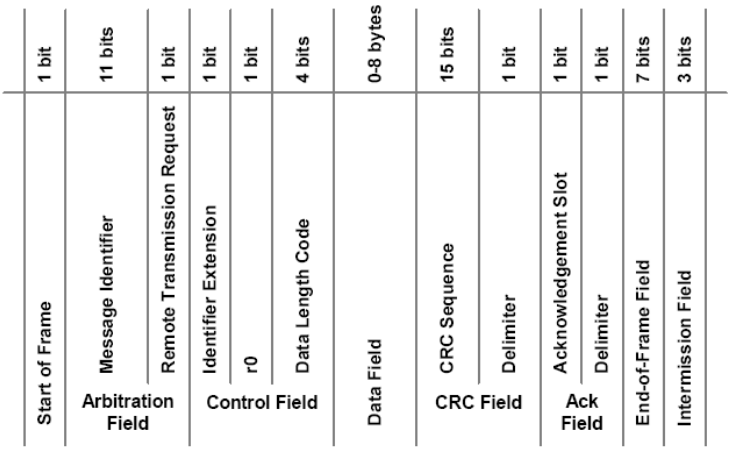


Figura 2.16 Formato base Data frame

Questi sono i campi del formato base:

- **Start of Frame**. Un singolo bit dominant segna l'inizio del frame di dati. Il bus CAN è in stand-by (recessive) prima della trasmissione di questo bit. Tutti i ricevitori sul bus devono utilizzare questo bit per sincronizzare i loro clock.
- **Arbitration Field**. Il campo arbitrazione contiene l'identificatore di campo e la richiesta di trasmissione a distanza (RTR) bit. Questo campo ha una duplice finalità. Viene utilizzato per determinare quale nodo ha accesso al bus e per identificare il tipo di dati che il messaggio contiene.
    - **Message Identifier**. L'identificatore del messaggio è lungo 11 bit nel formato Data Base Frame. Questo significa che è possibile avere 2048 messaggi unici. Più basso è il valore di questo campo, maggiore è la priorità.
    - **Remote Transmission Request** (RTR) bit. Il bit di RTR indica se il frame è un Data frame o un Remote frame. Questo bit deve essere dominant per un frame di dati.
- **Control Field**. Il campo di controllo contiene il bit *Identifier Extension*, un bit riservato (*r0*), e un *Data Length Code*.
    - **Identifier Extension** (IDE) bit. Se il bit IDE è dominant, allora il frame è nel formato Base Frame. In caso contrario, il frame è nel formato Extended.
    - **Reserved bit 0** (r0). Il bit riservato non viene utilizzato e dovrebbe essere sempre dominant.
    - **Data Length Code**. Il numero di byte è memorizzato nel campo lunghezza di codice. Questo campo di solito contiene i valori da zero a otto. Se il valore della lunghezza di codice è superiore a otto quindi si presume che il frame contenga otto byte.

- **Data Field**. Contiene da 0-8 byte e viene trasmesso prima il bit più significativo.
- **Cyclic Redundancy Check** (CRC). Il campo CRC è di 15 bit. Il ricevitore utilizza il valore in questo campo per vedere se la sequenza di bit di dati nel frame è stata danneggiata durante la consegna.
- **Acknowledgement Field**. Il campo di riconoscimento è di due bit. Contiene Acknowledgement.
- **Slot bit e Acknowledgement Delimiter Bit**. Il nodo che trasmette il frame di dati invia questi due bit in modo recessive. Il trasmettitore si aspetta che almeno un ricevitore riconosca la ricezione del messaggio trasmesso. Ogni nodo che riceve il messaggio senza errore, sovrascriverà lo Slot con un bit dominant. Il bit *Acknowledge Delimiter* rimane sempre recessive per distinguere una conferma positiva dall'inizio di un errore di frame. Se non riceve dal ricevitore un messaggio di acknowledge, il trasmettitore continuerà a tentare di inviare il messaggio fino a che il nodo si spegne.
- **End-of-Frame and Intermission Fields**. Ogni frame di dati termina con 7 bit di *End-of-Frame* e 3 bit di *Intermission Field*.
    - **End-of Field-Frame**. Questo campo deve essere trasmesso in modo recessive per contrassegnare un trasmissione completa senza errori. Se l'Acknowledgement Delimiter bit o uno degli End-of-Frame bits sono trasmessi in modo dominant allora questo segna l'inizio di un *Error or Overload Frame.*
    - **Intermission Field**. Rappresenta la quantità minima di spazio tra Data o Remote frame adiacenti. Tutti e tre i bit devono essere trasmessi recessive. Il bus potrebbe continuare a rimanere idle dopo questo, oppure un nuovo frame sarà indicato con un *Start-of-Frame* bit. Se uno dei primi due bit di questo campo è dominant allora si presume che un *Overload Frame* è iniziato.

Formato Data frame esteso

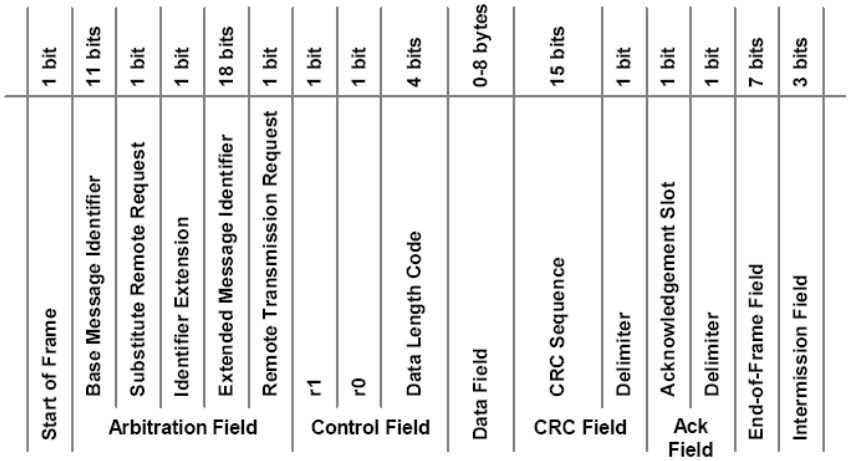


Figura 2.17 Formato esteso Data frame

Il formato *Extended Data Frame* è quasi identico al formato *Data Base Frame*. Le uniche differenze tra questi due formati si trovano nell'inserimento del bit di identificazione (IDE) nel settore del controllo, e la dimensione e la disposizione del campo di arbitrazione. Il formato base e quello esteso possono coesistere nello stesso sistema CAN. La regola di base è che i frame Formato Base hanno sempre la priorità sui frame Formato Esteso.
La seguente è una descrizione dei campi di controllo e arbritazione:

- **Arbitration Field**. E' più grande perché contiene 29-bit di identificatore. Più di 536 milioni di messaggi unici possono essere utilizzati con il formato esteso.
  - **Base Message Identifier**. E' identico al Message Identifier Field del formato base. È lungo 11 bit e contiene i bit più significativi dell'identificatore.
  - **Substitute Remote Request** (SRR) Field. Il bit SRR sostituisce il bit RTR nel formato base. Il suo unico scopo è quello di essere un segnaposto in modo che il bit Identifier Extension rimanga nella stessa posizione nella quale si trova nel formato Base. Questo bit è sempre trasmesso recessive. Il bit RTR per questo formato è stato spostato alla fine dell'Arbitration Field.
  - **Identifier Extension** (IDE) bit. Se il bit IDE è recessive, allora il frame è nel formato esteso altrimenti il frame è nel formato Base.
  - **Extended Message Identifier Field**. Aggiunge altri 18 bit al Base Message Identifier. Questo porta il numero totale di bit di identificazione a 29.

- **Transmission Request** (RTR) bit. E' identico al bit RTR nel formato base.

- **Control Field**. Il campo di controllo nel formato Extend frame contiene due bit di dati riservati e la Lunghezza del codice (DLC).

Remote frame

I Remote Frame vengono utilizzati dai ricevitori per richiedere informazioni da un altro nodo. Il formato per un Remote Frame è identico a quello di un frame di dati. Entrambi i tipi di struttura hanno formati base ed esteso, ed entrambi hanno un singolo *Remote Transmission Request* bit e un *Arbitration Field*. Con un Remote Frame, questo bit viene trasmesso in modo recessive per identificarlo come Remote Frame. Per Data frame il bit RTR è sempre trasmesso dominant.

Error frame

I ricevitori inviano un Error Frames ogni volta che individuano un frame che contiene un errore. L'Error Frame può essere inviato da qualsiasi punto di una trasmissione e viene sempre inviato prima che un frame di dati o Remote Frame sia stato completato con successo. Il trasmettitore controlla costantemente il bus durante la trasmissione. Quando il trasmettitore rileva l'Error Frame, si interrompe il frame corrente e si prepara a inviare di nuovo una volta che il bus si libera nuovamente. L'Error Frame contiene i seguenti campi:

- **Error Flag Field**. L'Error Frame viola deliberatamente le regole di bit stuffing con l'invio di sei recessive bit o sei bit dominant nel campo Error Flag. La determinazione del fatto che il flag di errore è recessive o dominant dipende dallo stato di errore del nodo.
- **Error Delimiter Field**. Il delimitatore di errore è una sequenza di otto bit recessive e indica la fine del frame. Il bus entrerà quindi in uno stato di idle o un nuovo Data Frame o Remote Frame avrà inizio. Si noti che non a tutti i nodi di una rete è permesso l'invio di Error Frames.

Overload frame

Overload Frame può essere considerato come uno speciale Frame Error. Viene utilizzato per chiedere ad un trasmettitore di ritardare ulteriormente i frame o per segnalare problemi con la Intermission Field. L'Overload Frame ha lo stesso formato di un Error Frame ma a differenza sua non causa la ritrasmissione del frame precedente. Se Overload Frame viene utilizzato per ritardare ulteriormente le trasmissioni, allora non più di due Overload Frames possono essere generate successivamente. Un Overload Frame include un campo Flag che contiene una sequenza di sei bit dominant, seguito da un delimitatore Campo, una serie di otto bit recessive. Quando un nodo invia Overload flag tutti i nodi sulla rete lo rilevano e inviano i loro overload flags, sospendendo tutto il traffico di messaggi sul sistema

CAN. Poi, tutti i nodi ascoltano per una sequenza di otto bit recessive. La quantità massima di tempo necessario per recuperare in un Sistema CAN dopo Overload Frame è di 31 bit.

Error detection

Il Data Link nei sistemi "Controller Area Network" è molto efficace nella rilevazione degli errori. Ogni frame è contemporaneamente accettato o respinto da ogni nodo della rete. Se un nodo rileva un errore, trasmette un flag di errore per ogni altro nodo della rete e distrugge il frame trasmesso.
CAN utilizza i seguenti metodi per individuare gli errori:

- **Bit Check**. Ogni nodo controlla i propri messaggi nel modo in cui li trasmette attraverso il bus, controllando i bit di ogni frame per gli errori. Il nodo rileverà un errore di bit se si manda un bit dominant, quando si suppone di trasmettere un bit recessive, o un bit recessive quando il messaggio è chiamato per un bit dominant.
- **Frame Check**. Ogni tipo di frame contiene i campi di bit specifici che hanno sempre valori fissi. Per esempio, il primo bit in ogni frame deve essere sempre dominant bit. Ogni nodo controlla questi campi fissi ad ogni frame che riceve. Un nodo è in grado di rilevare un errore di frame se un campo di bit fisso contiene un valore illegale. Errori di questo tipo sono chiamati *Form Errors*.
- **Cyclic Redundancy Check**. E' un metodo molto efficace di ricerca di errori in una rete mediante l'applicazione di una equazione polinomiale ad un blocco di dati trasmessi. Un polinomio è un'espressione matematica complessa, dove una serie di valori si sommano, e ogni valore include una variabile o variabili elevate a potenza, moltiplicato per un coefficiente. Questo semplice polinomio ha una sola variabile:
  $a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x^1 + a_0 x^0$
  Qui a è il coefficiente e x rappresenta la variabile. Il nodo calcola il codice di ridondanza ciclico (CRC) per ogni messaggio che invia, e poi lo aggiunge al valore risultante alla fine del frame. Il nodo che riceve il messaggio applica lo stesso polinomio al frame e confronta il suo risultato col codice CRC fornito dal mittente. Se i due valori non corrispondono, il ricevente invia un messaggio di errore al trasmettitore. Il metodo di ridondanza ciclica è molto affidabile per la ricerca di errori nelle trasmissioni.
- **Acknowledgement Check**. Ogni nodo che trasmette un messaggio ascolta e aspetta per un Ack nello Slot ACK che proviene da un altro nodo della rete. Se il trasmettitore non riceve una risposta allora questo è considerato un errore di riconoscimento. Il nodo continuerà a ritrasmettere il frame finché non riceve un riconoscimento.
- **Stuff Rule Check**. Ogni nodo controlla se c'è una violazione della regola di bit stuff. Se un nodo rileva più di cinque bit uguali nella stessa riga, allora uno *Stuff Error* si è verificato e un *Error Frame* viene inviato. Il nodo registra un bit stuff error, e invia un Error Frame al trasmettitore.

Contenimento del guasto

Il rischio maggiore con qualsiasi sistema di bus seriale è che un singolo nodo difettoso possa rovinare l'intera rete. Per affrontare questo, il protocollo CAN è stato progettato per rilevare automaticamente un nodo guasto e scollegarlo dalla rete. CAN richiede due contatori di errore per ogni nodo, uno per tenere traccia di errori di trasmissione e l'altra per tenere traccia degli errori ricevuti. Se un errore di trasmissione o di ricezione si verifica, il rispettivo contatore è incrementato di un valore ponderato a seconda del tipo di errore e quale nodo ha causato l'errore. Per ogni trasmissione o ricezione di successo il rispettivo contatore viene decrementato di uno.
Sulla base di questi contatori, un nodo può trovarsi in uno dei tre stati:

- **Error Active**. Il nodo sta funzionando normalmente. Prende parte nelle comunicazioni di rete e invia un messaggio di errore quando rileva un errore, distruggendo il frame trasmesso.
- **Error Passive**. Uno dei contatori di errore del nodo ha superato 127. Il nodo può ancora inviare e ricevere messaggi sul bus, ma non gli è più permesso inviare Error Frames con flag di errore attivo. Un nodo Error Passive può solo rispondere a un errore con l'invio di Error Frame con un flag di errore passivo. Inoltre, se un nodo Error Passive invia un frame di dati con un errore, si devono attendere otto bit prima di tentare la ritrasmissione. Entrambe queste misure limitano i nodi che hanno un più alto error rate medio, in modo che non possano interferire con la comunicazione attraverso il bus. Un nodo può diventare errore attivo di nuovo quando entrambi i contatori di errore scendono sotto 128.
- **Bus Off**. Al nodo CAN bus che è fuori dal bus non è consentito influenzare il bus in alcun modo. E' consentito solo ricevere messaggi. Un nodo può diventare bus off solo a causa di errori di trasmissione. Un nodo entra nello stato bus off quando il contatore di trasmissione supera 255. Ciò significa che un nodo difettoso verrà disattivato se trasmette 32 messaggi consecutivi con errori. Un nodo che è entrato nello stato bus off può solo uscire da questo stato resettando contatori e CAN controller. Il resto della rete continuerà a funzionare anche se un nodo è impostato su Bus Off.

Si noti che gli errori a volte appaiono in una rete a causa di fattori esterni, per esempio se la rete è esposta a temporanee interferenze elettromagnetiche.

2.3 LIN, FlexRey, Most

Tra i bus esistenti per applicazioni ad elevata affidabilità, se ne possono individuare ancora tre tra i principali, tutti nati in campo auto-motive.

LIN è stato progettato dal consorzio LIN e la prima specifica è stata pubblicata nel 1999. E' stato progettato particolarmente per la comunicazione a basso costo tra sensori intelligenti e attuatori in applicazioni automobilistiche. E' destinato ad essere utilizzato quando comunicazioni ad alto bit rate come il CAN bus non sono necessarie. Può essere impostata sull'interfaccia hardware UART /USCI, software UART o macchine a stati([9] *Introduction to LIN (Local Interconnect Network)*).
Le caratteristiche principali del LIN sono:

- bassi bit rate (massimo 20 kbit /s);
- comunicazione su linea singola;
- architettura master/slave;
- ridotti requisiti hardware (semplicità delle macchine a stati, implementabile su microcontrollori a basso costo);
- sincronizzazione automatica degli slave, che non necessitano di quarzi o risuonatori ma solo di un oscillatore RC;
- latenza massima 100 ms;
- lunghezza massima del collegamento 40m e 16 slave massimi.

Una rete LIN è costituita da un master LIN e uno o più slave LIN. Di solito in applicazioni auto-motive, il bus LIN è collegato tra sensori intelligenti o attuatori e una unità di controllo elettronica (ECU) che spesso è un gateway con bus CAN. I vantaggi di questo protocollo sono:

- facilità d'uso;
- disponibilità dei componenti;
- Più economico di bus CAN e altre comunicazioni;
- riduzione dei cablaggi;
- affidabilità veicoli;
- facile da implementare.

FlexRay è un bus per applicazioni x-by-wire in ambito auto-motive introdotto dall'omonimo consorzio formato nel 2000 da Bmw, Bosch, Daimler, Freescale, General Motors, Nxp Semiconductors e Volkswagen. È espressamente progettato per applicazioni critiche, come appunto i controlli di attuazione mediante filo (quali ad esempio il controllo della frenata o della sterzata nelle autovetture). Sono previsti per questi, meccanismi di tolleranza e contenimento dei guasti e bus guardian. Lo standard supporta meccanismi di comunicazione time-triggered ed event-triggered. Implementazioni tipiche prevedono un periodo base con una finestra temporale per l'invio di messaggi statici secondo uno schema time-triggered e una per messaggi dinamici event-triggered. Il mezzo fisico è un doppino twistato; la massima lunghezza del bus è specificata fino a 24 metri. La topologia della rete è di tipo multidrop o punto-punto, anche in questo caso con supporto di hub per connessioni a stella. Il

numero massimo di nodi che possono essere connessi al bus è 64. Per le connessioni per l'invio di messaggi statici può essere implementata ridondanza a livello fisico. Il data-rate massimo sostenuto è 10 Mbps (20 Mbps se si utilizzano connessioni ridondate). La comunicazione è di tipo half-duplex, la modalità di accesso al mezzo fisico Tdma per i messaggi statici; nelle finestre dinamiche, invece, sono previsti mini-slot di trasmissione con arbitraggio sul bus. Sono previsti una base globale dei tempi ed un meccanismo di sincronizzazione; è garantita latenza programmabile nella spedizione dei messaggi fino a 6 µs. Sono disponibili core IP per Fpga qualificate per applicazioni spaziali. Le principali caratteristiche sono:

- data rate massimo per canale 10 Mbit/s (20 Mbit/s con l'utilizzo del canale doppio);
- accesso al canale tramite TDMA (Time Division Multiple Access), senza necessità di arbitrazione tra i nodi e con latenze massime garantite;
- sincronizzazione fault-tolerant degli offset e dei rate degli orologi locali di ogni nodo;
- supporto per l'utilizzo di bus guardian che impedisce l'indisponibilità del canale a causa di *babbling idiot*.

Il protocollo **MOST** (Media Oriented System Transport) si differenzia dalle altre reti di comunicazione in quanto utilizza principalmente collegamenti a fibre ottiche plastiche (POF, anche se è definito un livello fisico su UTP, Unshielded Twisted Pair). Lo standard nasce per supportare lo sviluppo di applicazioni multimediali in ambiente auto-motive ed è quindi caratterizzato da un'elevata capacità di trasporto, ma anche da un'operatività real-time e dalla possibilità di utilizzo di topologie di rete ridondate. La rete è formata da collegamenti punto-punto che possono essere disposti in topologie ad anello, stella o daisy-chain e sono definiti diversi livelli di accesso. E' quindi possibile il collegamento di periferiche a bassa complessità senza capacità di auto-motive (come i convertitori D/A per altoparlanti) o di periferiche che richiedano meccanismi di controllo più sofisticati e bitrate maggiori.
Le principali caratteristiche del protocollo MOST sono:

- bit rate massimo per trasferimenti sincroni 50Mbit/s;
- possibilità di trasferimenti asincroni a velocità variabile;
- elevata tolleranza ai disturbi elettromagnetici con l'utilizzo di comunicazione ottica;
- prevedibilità nelle emissioni elettromagnetiche grazie all'architettura sincrona;
- ritardo di accesso garantito tramite arbitrazione a priorità.

Il protocollo copre le funzionalità di tutti i 6 livelli OSI inferiori fino al livello Presentazione e consente il trasporto trasparente di altri protocolli che aderiscano allo stesso modello come, ad esempio, il TCP/IP. Questo contribuisce alla flessibilità di utilizzo del protocollo e ne permette l'utilizzo anche in applicazioni diverse da quelle strettamente auto-motive.

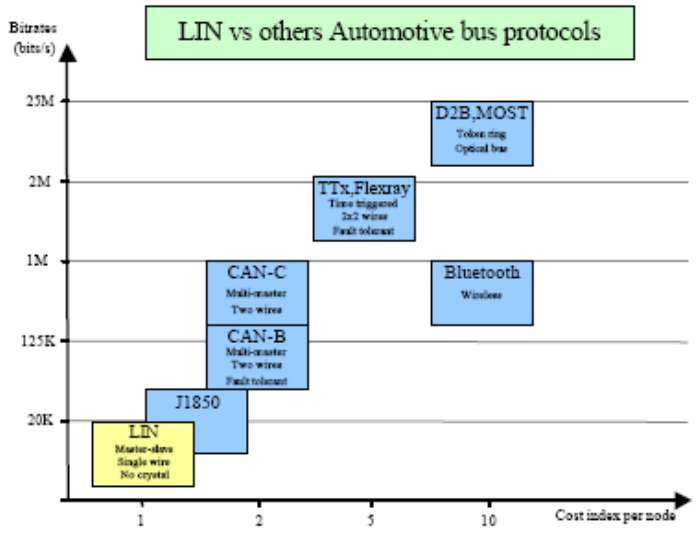


Figura 2.18 Distribuzione dei vari standard automotive in funzione di bit rate e costo

Capitolo 3

Interfaccia di comunicazione bus in Aramis

Ogni modulo Aramis ha necessità di comunicare con gli altri per lo scambio di informazioni, per la regolazione di attuatori di controllo di assetto e per comunicare con il transceiver.

La modularità del sistema di comunicazione per l'architettura AraMiS implica che l'accesso al bus da parte dei vari nodi avvenga in maniera flessibile. Un singolo modulo può ad esempio essere scollegato dall'alimentazione per questioni di risparmio energetico o di latch-up e questo non deve influire sulla disponibilità del canale.

E' anche necessario massimizzare la tolleranza ai disturbi del canale e la comunicazione sul bus deve essere possibile anche in caso di guasto del canale fisico o dei driver che comandano il canale.

Con queste condizioni, la scelta migliore è quella di introdurre un certo grado di isolamento galvanico tra i vari elementi del bus, in modo da rendere il canale insensibile alle condizioni statiche dei vari elementi che vi si collegano.

L'isolamento può in generale essere ottenuto tramite accoppiamento magnetico. Il segnale è rappresentato da un campo magnetico, generato da un lato da una corrente in un avvolgimento di un trasformatore e ricevuto, dall'altro lato, dal secondo avvolgimento su cui il campo magnetico viene indotto.
Innanzitutto la soluzione magnetica non è in grado di accoppiare tra i due lati del trasformatore la componente continua del segnale. Per avere induzione sul secondario del trasformatore è infatti necessario che il campo magnetico, proporzionale alla variazione di corrente sul primario, sia variabile e non ecceda i livelli di saturazione del nucleo. E' quindi necessaria una qualche modulazione del segnale da trasmettere che rispetti questi vincoli.

3.1 Codifica di canale

La codifica di canale utilizzata si chiama Return-to Zero (RZ) unipolare ed associa ad un livello logico la presenza di un impulso e rispettivamente all'altro livello logico l'assenza di un impulso.

Nella figura 3.1 seguente è rappresentata la codifica RZI (Return-to-Zero Inverted) utilizzata nel protocollo IRDA. Nel nostro caso al livello logico "0" è associato un impulso e al livello logico "1" è associato l'assenza di un impulso. Questo tipo di codifica è molto interessante perché può essere utilizzata nell'accoppiamento tramite i trasformatori d'impulso. Quest'ultimi sono particolarmente utilizzati con segnali impulsivi. Infatti le specifiche dei trasformatori d'impulso riportano, tra i parametri caratterizzanti, una grandezza, chiamata generalmente ET, che dimensionalmente è un prodotto tensione-tempo (un'energia) ed esprime la dimensione massima di un impulso rettangolare che non porti il nucleo del trasformatore in saturazione.

Se un trasformatore è caratterizzato ad esempio da una ET pari a 6V * µs, potrà accoppiare sia un impulso di durata 3 µs e ampiezza 2V che un impulso di durata 0.5 µs e ampiezza 12 V.

Un altro punto a favore di questa codifica è la semplicità implementativa del sistema di comunicazione, in quanto le interfacce UART effettuano la codifica e decodifica RZI.

I processori di bordo scelti per l'architettura Aramis, gli MSP430 della Texas Instrument, hanno all'interno una periferica UART in grado di utilizzare la codifica RZI.

Data la flessibilità e la semplificazione del sistema di comunicazione di questo tipo di codifica, la scelta deve ricadere sull'utilizzo della codifica RZI (con IRDA [13]).

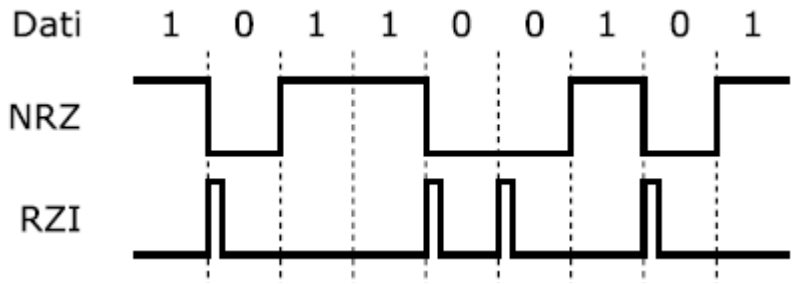


Figura 3.1 Esempio di codifiche Non-Return-to-Zero (NRZ) e Return-to-Zero Inverted (RZI)

3.2 Tolleranza ai guasti

Il lato secondario del trasformatore è connesso al bus tramite due resistenze ed è presente una presa centrale collegata alla tensione di riferimento. Questo tipo di collegamento permette al bus di tollerare diversi tipi di guasto e garantisce, tramite il collegamento della presa centrale, un riferimento di potenziale fisso per il bus. Infatti, in assenza del collegamento alla tensione di riferimento, il bus sarebbe completamente isolato [4].

Il bus tollera i seguenti guasti:

- Cortocircuito tra un conduttore del bus e la tensione di riferimento (figura 3.2 a); il semi-avvolgimento a cui è collegato il conduttore danneggiato avrà una tensione nulla ai propri capi ma il conduttore integro continuerà a funzionare; la tensione sul bus e i margini di rumore vengono dimezzati;
- Cortocircuito tra un conduttore del bus e una tensione di alimentazione (figura 3.2 b); stesso effetto del cortocircuito verso la tensione di riferimento, ma il semi-avvolgimento interessato dal guasto vede ai suoi capi la tensione di alimentazione; la resistenza posta in serie all'avvolgimento previene la saturazione del nucleo del trasformatore e il bus continua a funzionare con margini di rumore dimezzati;
- Interruzione di uno dei conduttori (figura 3.2c); il sistema continua a funzionare grazie al conduttore integro con margini di rumore dimezzati; sulla sezione del conduttore danneggiato che non è raggiunta dal segnale, viene ricreata una parte della tensione originale dall'induzione, sul semi-avvolgimento interessato dal guasto, di una parte del campo magnetico generato dal semi-avvolgimento funzionante;
- Cortocircuito del secondario di un trasformatore collegato al bus (figura 3.2d); il bus continua a funzionare normalmente e viene impedita la comunicazione solo verso il sistema che utilizza il trasformatore guasto;
- Cortocircuito tra i due conduttori (figura 3.2e); in questo caso la tensione differenziale tra i conduttori del bus è nulla, ma è comunque possibile ottenere una differenza di potenziale ai capi del trasformatore sfruttando la tensione di modo comune e uno sbilanciamento delle resistenze o del rapporto spire dei due semi-avvolgimenti; un analogo sbilanciamento al lato trasmettente genererà la tensione di modo comune a partire dal segnale.

La comunicazione sul bus verrà quindi annullata solo dalla presenza di guasti multipli che interessino contemporaneamente entrambe le linee. Qualunque guasto statico che si verifichi sul lato primario del trasformatore vi rimarrà confinato e non interferirà sul funzionamento del bus.

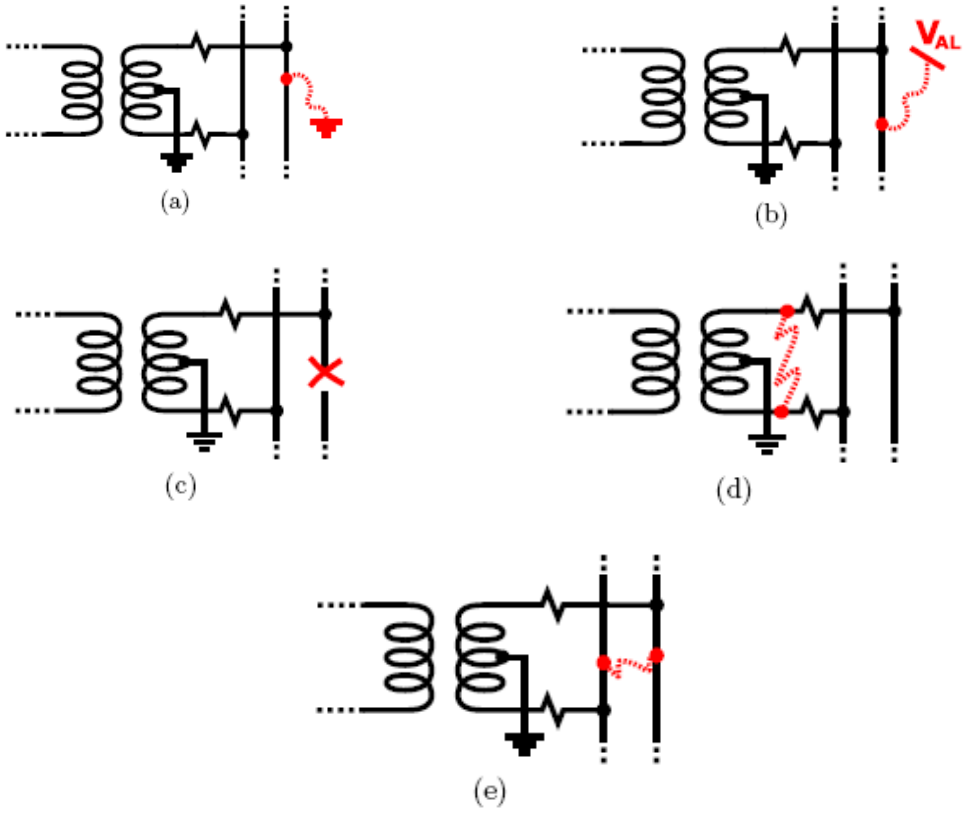


Figura 3.2 Possibili guasti sul canale

3.3 Driver

Tramite l'utilizzo di un driver open-collector e di un pulse transformer diventa possibile, con la chiusura del driver, l'invio sul bus degli impulsi RZI. In assenza degli impulsi invece, l'uscita del driver si trova in uno stato di alta impedenza e diventa possibile la ricezione dei dati dal bus tramite il receiver differenziale.
La sola criticità dello schema è nella scelta del receiver, che dovrà essere in grado di funzionare correttamente anche con una tensione di modo comune sugli ingressi prossima alla tensione di alimentazione. Sono però disponibili sul mercato dei receiver RS-485 come il MAX3281E della Maxim Integrated Products che tollera, con un'alimentazione di 3 V, una tensione di modo comune sugli ingressi compresa nel range -7V ÷ +12V.

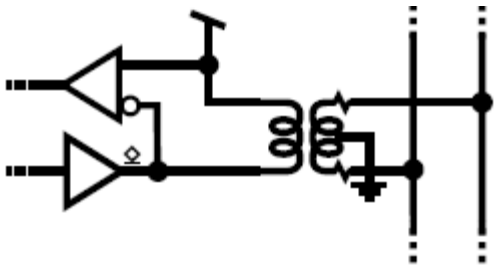


Figura 3.3 Accoppiamento magnetico e trasmissioni tramite driver open-collector

3.4 Interfaccia processore-bus

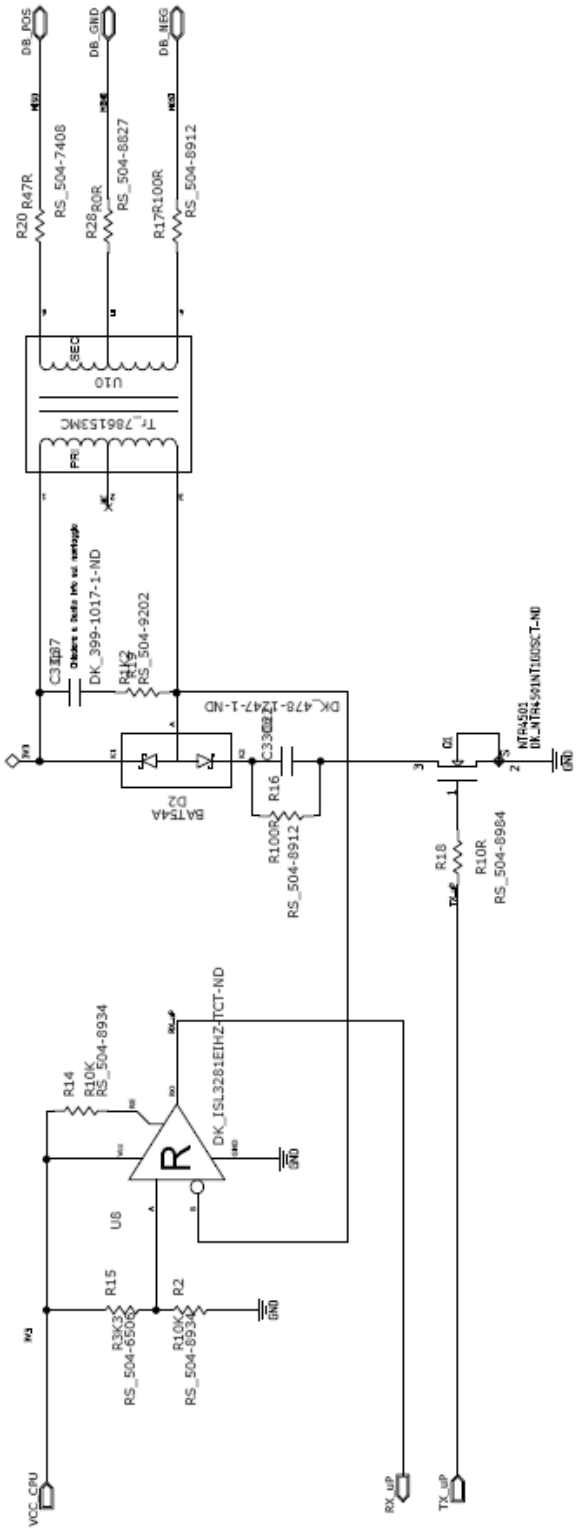


Figura 3.4 Schema elettrico interfaccia processore-bus

Avendo Aramis molti moduli il compito principale è quello di progettare l'interfaccia tra il processore contenuto in ognuno di questi moduli e la comunicazione con il canale. L'interfaccia è divisa in 3 parti in base alle loro funzioni. Si tratta di un trasmettitore, un ricevitore e la parte di accoppiamento. Il trasmettitore ha il compito di mandare il segnale sul bus, il ricevitore di ricevere il segnale proveniente dal bus al processore.
Nella figura 3.4 è rappresentato lo schema elettrico dell'interfaccia 1B411_OBDB. Nel circuito sono presenti oltre ai connettori di alimentazione anche i connettori RX e TX per comunicare col processore ed i connettori per comunicare con il canale.

3.4.1 Accoppiamento

L'accoppiamento non deve solo essere in grado di trasmettere o ricevere segnali dal bus dati, ma dovrebbe anche essere in grado di fornire due riferimenti a terra (isolamento).
Nel nostro caso si è scelto un trasformatore d'impulso della serie 786 della Murata Power Solution, come visto in Figura 3.6 per la realizzazione di tali funzioni [11].

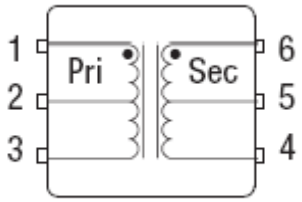


Figura 3.5 Trasformatore d'impulso

La scelta nell'usare un trasformatore d'impulsi in Aramis comporta sia dei vantaggi che degli svantaggi.
Un vantaggio è sicuramente l'isolamento. Un trasformatore è un dispositivo che trasferisce energia elettrica da un circuito all'altro attraverso conduttori ad accoppiamento induttivo, bobine del trasformatore o "avvolgimenti".

Il trasformatore si basa su due principi: in primo luogo, che una corrente elettrica può produrre un campo magnetico (elettromagnetismo) e, dall'altro, che un campo magnetico variabile all'interno di una bobina di filo induce una tensione ai capi della bobina (induzione elettromagnetica). Variando la corrente nella bobina primaria varia l'entità del campo magnetico applicato. Il flusso magnetico variabile si estende alla bobina secondaria in cui viene indotta una tensione alla sua estremità. Così possiamo trarre la conclusione che solo una variazione di corrente sul lato primario può

influenzare una tensione sul lato secondario del trasformatore, e il trasformatore fornisce isolamento tra ogni lato.

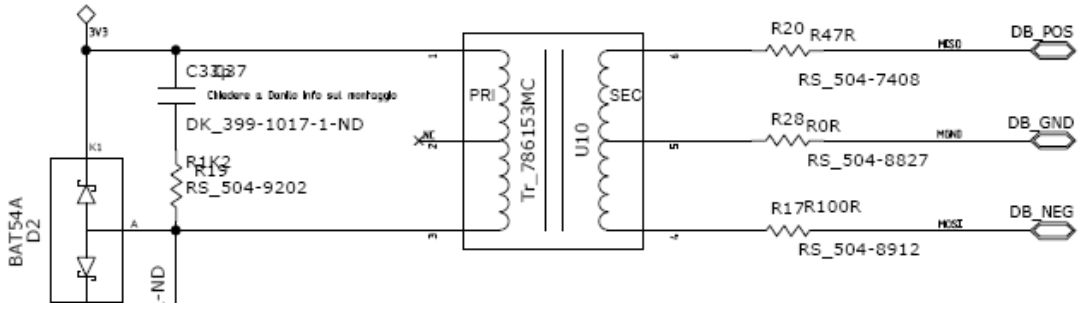


Figura 3.6 Schema del circuito di accoppiamento

Come si può vedere dalla figura 3.6 precedente un capo del lato primario è collegato alla tensione di alimentazione positiva mentre l'altro capo viene collegato alla tensione di riferimento dal MOS per la generazione dell'impulso. Il secondario del trasformatore è collegato al bus tramite le resistenze R20, R28 e R17.

3.4.2 Trasmettitore

Il trasmettitore deve anche essere compatibile con la forma d'onda del segnale in ingresso e la sua frequenza, nel nostro caso il segnale di ingresso è un'onda rettangolare con duty cycle del 10%, con frequenza di 1 MHz e larghezza d'impulso di 100ns.

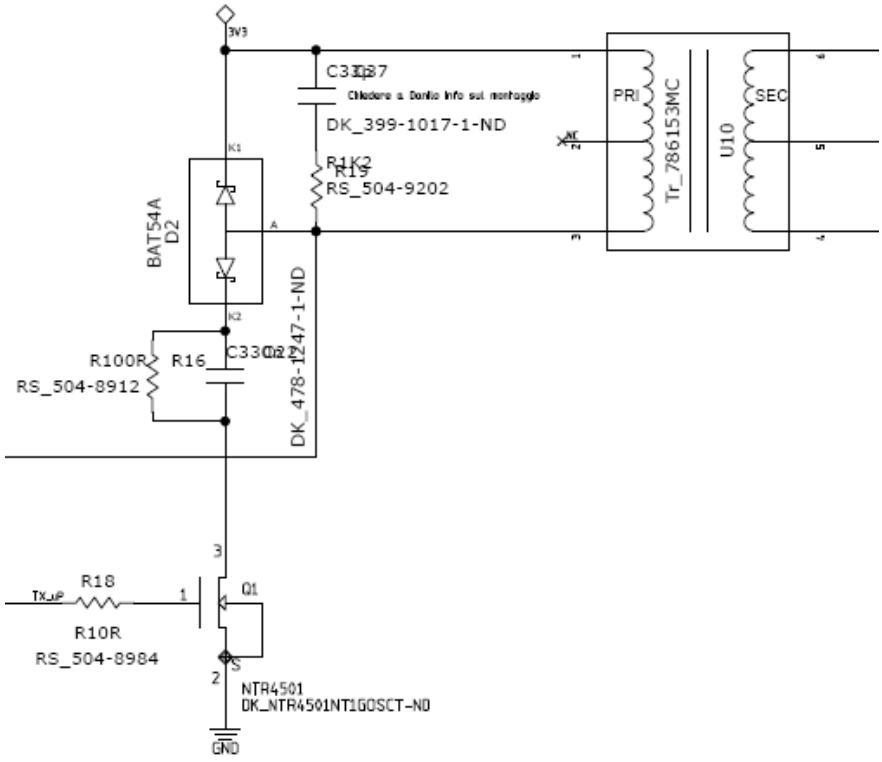


Figura 3.7 Schema del circuito di trasmissione

Come mostrato in Figura 3.7, nel circuito di trasmissione è presente un normale transistor NTR4501 NMOS [12] con resistore, e il segnale d'ingresso impulsivo proveniente dal processore viene iniettato al Gate tramite una resistenza di limitazione della corrente. Si può facilmente ottenere la curva I_D (t), come mostrato in figura 3.8.

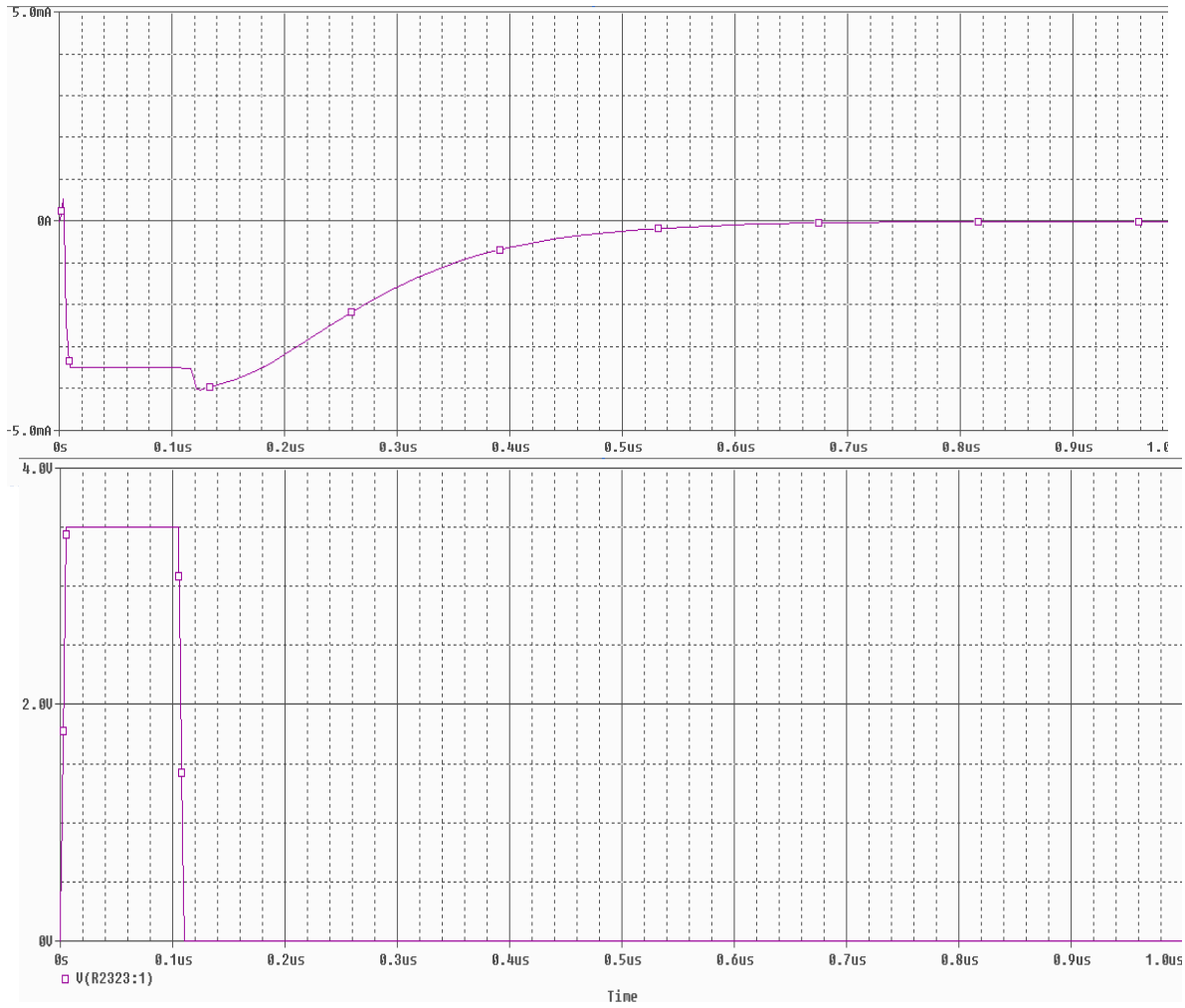


Figura 3.8 Segnale in ingresso e curva $I_D(t)$ dello switch NTR4501

Le forme d'onda osservate nelle Fig.3.8 sono molto diverse da quelle ideali. Infatti il valore di picco di corrente è molto piccolo, un valore di corrente di picco piccolo porta a un valore piccolo di segnale di tensione trasmesso sul bus, ed il tempo di salita della corrente è troppo lungo.
Questi problemi potrebbero essere risolti con l'introduzione del condensatore C22. Se costruiamo il trasmettitore in questo modo, la pendenza del tempo di salita della corrente sarà più grande.

Confrontando la forma d'onda di Fig.3.9 e Fig 3.10, è facile scoprire che con l'inserimento di C22, il valore di picco è aumentato da 3 mA a 13.5mA. Nel frattempo, il tempo di salita decresce da 0.8µs a 0.11µs. Così il risultato della simulazione è del tutto abbinato all'analisi teorica.

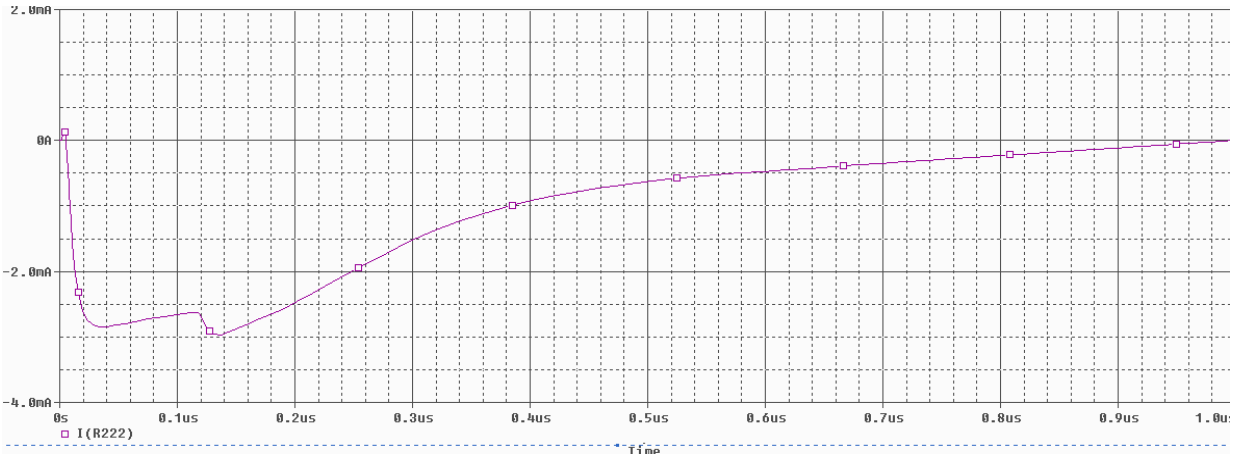


Figura 3.9 Forma d'onda senza capacità C22

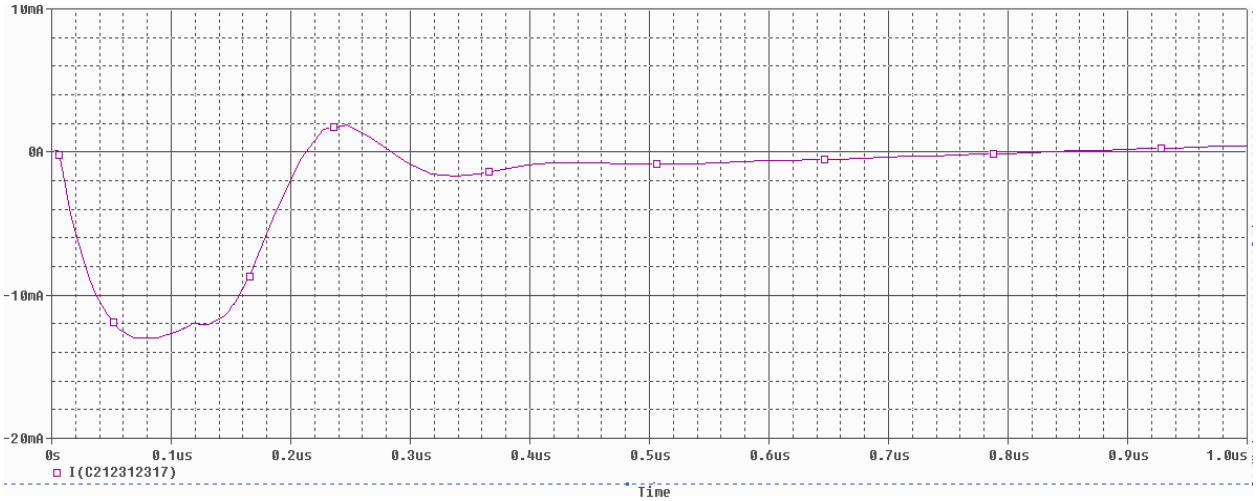


Figura 3.10 Forma d'onda con capacità C22

Analizzando le figure si nota che il valore di picco della corrente diventa più grande. Questo perché durante il periodo di salita della tensione, il segnale d'ingresso potrebbe essere considerato come la combinazione di alcune componenti AC, e solo queste componenti AC scorrono nel condensatore e la resistenza non lavora del tutto, finchè può essere visto come un circuito aperto.

In questa condizione, il guadagno in AC è molto più grande del guadagno in DC.

Si può notare anche che il tempo di discesa diventa più piccolo. Quando il MOS è chiuso, il condensatore si comporta come un corto circuito. Tuttavia, abbiamo bisogno di mantenere costante la corrente per tutta la larghezza di impulso. E questo problema può essere risolto con l'introduzione di un diodo.

Come si può notare dalla figura 3.7, sono presenti due diodi Schottkly BAT54. Il primo diodo si trova ai capi del primario del trasformatore, è normalmente in polarizzazione inversa e scaricherà l'energia accumulata nell'induttanza del primario all'apertura del MOS. Il secondo diodo in serie al trasformatore è necessario per evitare oscillazioni del segnale sulla linea.

3.4.3 Ricevitore

Il progetto del ricevitore è basato su un confronto (amplificatore operazionale open loop). Un amplificatore operazionale senza reazione negativa può essere utilizzato come un comparatore. Quando l'ingresso non invertente (V +) si trova ad una tensione superiore a quella dell'ingresso invertente (V-), l'alto guadagno dell'operazionale causa una tensione positiva. Quando l'ingresso non invertente (V +) scende sotto l'ingresso invertente (V-), l'uscita avrà una tensione negativa.

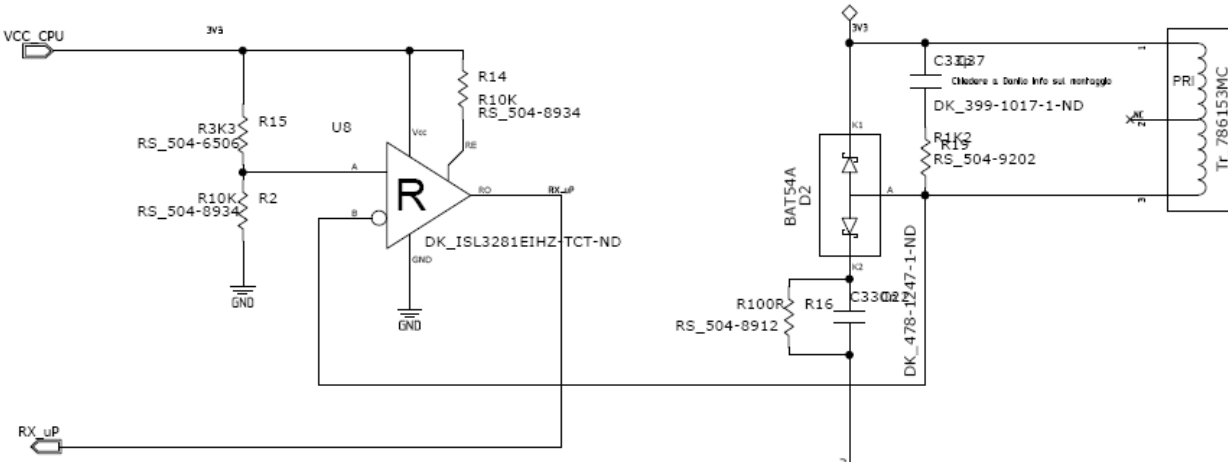


Figura 3.11 Schema del circuito di ricezione

Quando c'è un segnale trasmesso da altra unità per l'accoppiamento, il segnale di tensione sul pin 3 del trasformatore in figura 3.11 cambierà. Se il segnale è a livello logico 1, la tensione sul pin 3 scenderà ad un basso livello. Se il segnale è a livello logico 0, la tensione sul pin 3 sarà la stessa di VCC.
Secondo questa analisi, potremmo usare VCC e la tensione sul pin 3 come i segnali di ingresso dell'amplificatore operazionale. Si può connettere VCC all'ingresso non invertente ed la tensione sul pin 3 all'ingresso invertente dell'amplificatore. Durante il periodo in cui (VCC>V(pin 3)), l'uscita dell'amplificatore operazionale è V+, contrariamente, se (VCC <V(pin 3)), l'uscita sarà V-.
Questa è l'analisi di una condizione ideale.
Tuttavia, nel caso reale, l'amplificatore operazionale non si comporta esattamente in questo modo.

Quindi nel nostro caso la ricezione è affidata al receiver differenziale MAX3281E della Maxim Integrated Products.

Quando il circuito di interfaccia collegato al bus si troverà in modalità di ricezione avrà il MOS interdetto e le capacità parassite del MOS collegate in serie all'avvolgimento primario. Un impulso proveniente dal bus che si accoppi sul lato primario farà oscillare questo circuito LC e le oscillazioni si accoppieranno a loro volta sul lato secondario propagandosi sul bus. L'entità di tali oscillazioni dipendono sia dalle caratteristiche d'impulso che dalle varie capacità parassite presenti nel circuito. Per smorzare queste oscillazioni si potrebbe pensare di inserire una resistenza in serie al MOS. Questo ridurrebbe però il gradiente di corrente generato dalla chiusura del MOS, con una conseguente riduzione dei margini di rumore dell'impulso sul bus. Utilizzando il diodo invece, le oscillazioni vengono bloccate in quanto la corrente di risonanza prodotta dall'induttanza del primario riuscirà a caricare le capacità parassite del MOS, ma queste non riusciranno a scaricarsi a loro volta sull'induttanza.

3.5 Processore MSP430

I processori usati, in particolar modo nel nostro caso gli MSP430F2418 da 80 pin, contengono una completa periferica di comunicazione asincrona chiamata USCI (Universal Serial Communication Interface) che permette di svolgere i compiti di trasmissione e ricezione asincrona con la codifica e decodifica dei segnali richiesta. I pin di TX e RX che vengono collegati al connettore sono rispettivamente i pin 32 e 33 di ciascun processore.

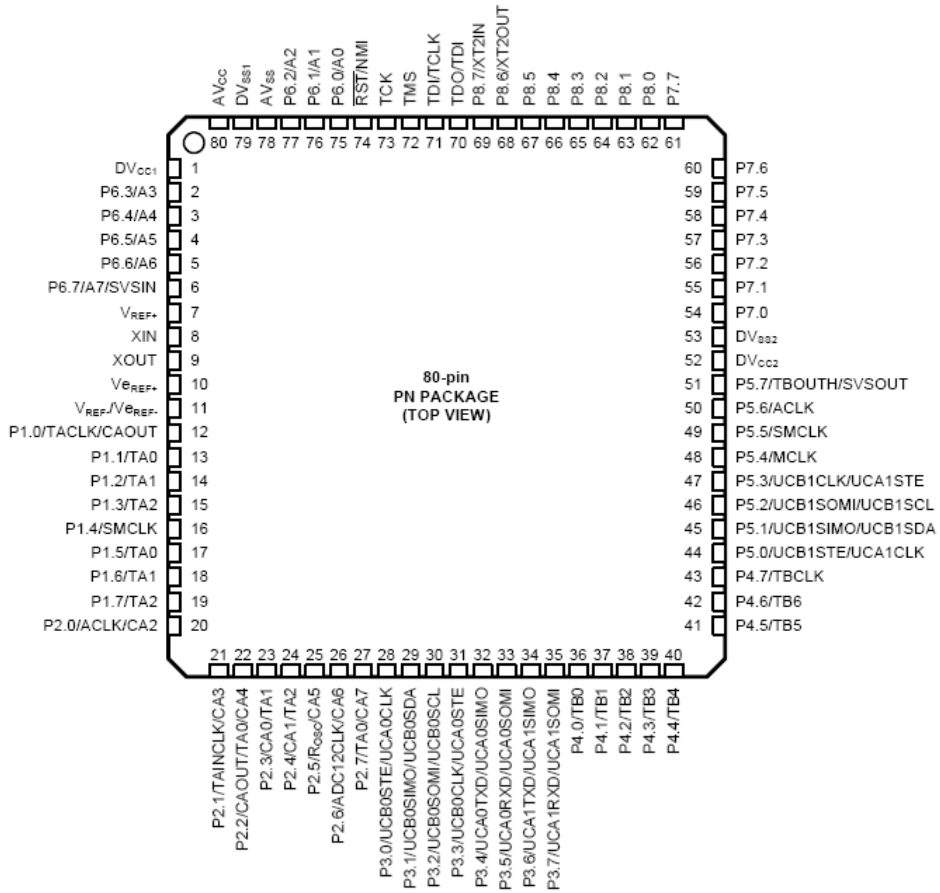


Figura 3.12 80-pin package dell' MSP430F2418 [15]

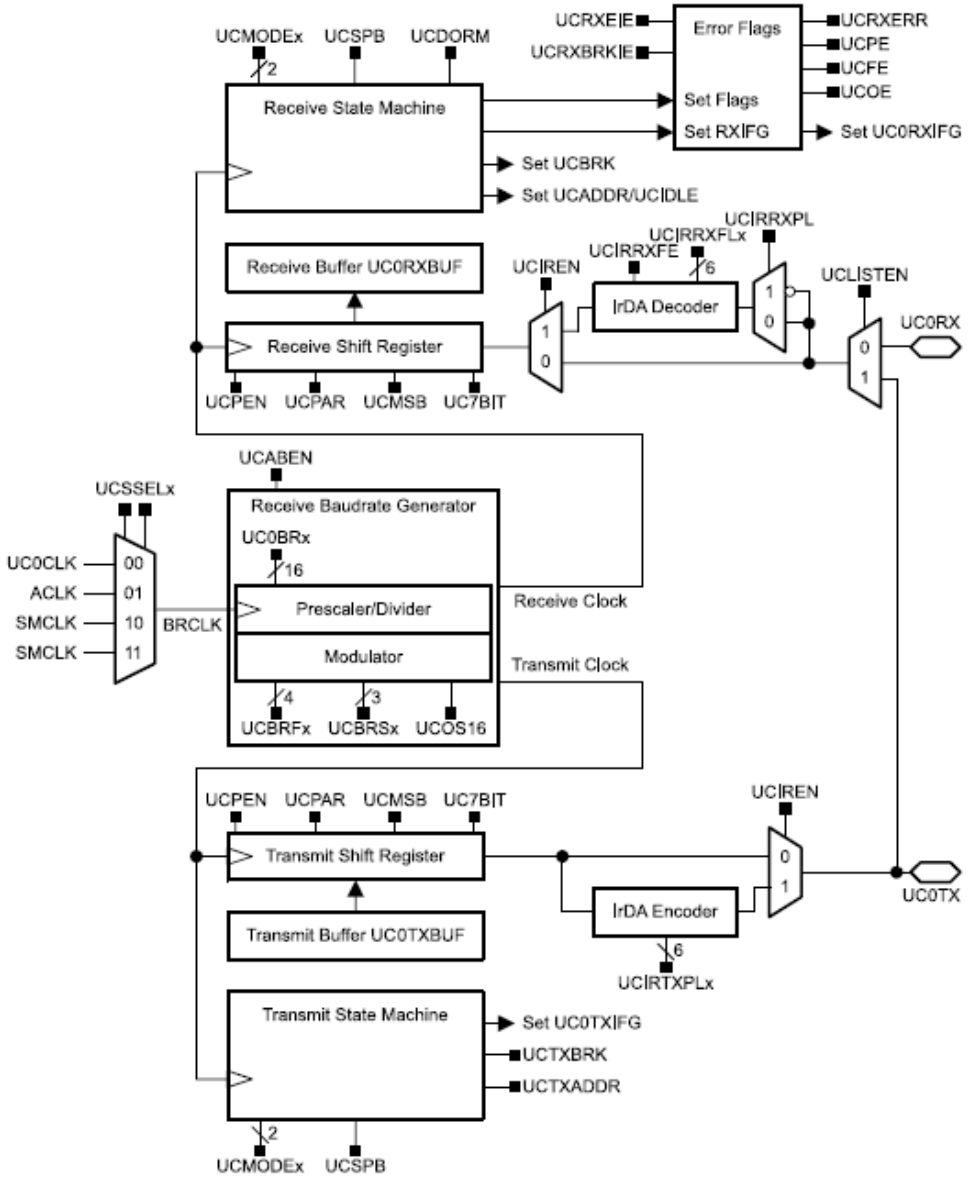


Figura 3.13 Schema interno della USCI in modalità UART di un processore della famiglia MSP430

Per avere la compatibilità con i processori MSP430 e per evitare di avere periferiche esterne di gestione del protocollo, si definiscce un pacchetto secondo lo standard seriale asincrono, con 1 bit di start, 8 bit di dati e 1 bit di stop
Il formato di pacchetto proposto è composto da i seguenti campi:

- **Identifier**: 8 bit di identificazione del destinatario o del mittente del messaggio;
- **SEQ** : 1 bit di sequenza per rilevare la perdita di un pacchetto;

- **Data Lenth**: 7 bit che identificano il numero di word di dati in un pacchetto, con massimo 127 word pari a 256 byte;
- **Data Bytes**: byte di dati trasportati dal pacchetto;
- **Crc**: controllo di ridondanza ciclico a 16 bit applicato su tutti i campi precedenti escluso i bit di start e stop.

Se un pacchetto non supera il controllo del Crc dovrà essere immediatamente scartato.

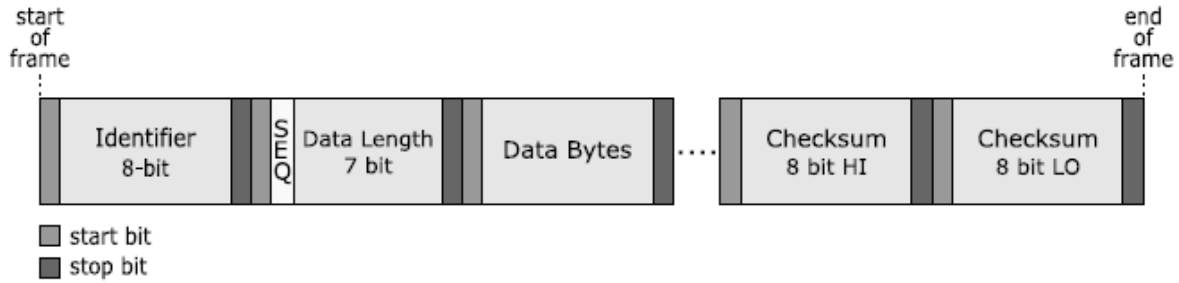


Figura 3.14 Formato del pacchetto

Nella seguente figura 3.15 è illustrato un semplice scambio di dati tra il master e uno slave.

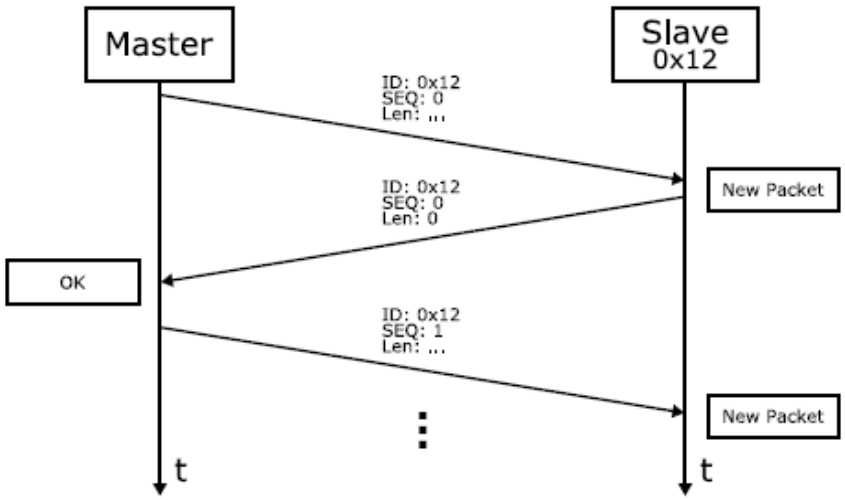


Figura 3.15 Scambio dati tra Master e Slave

Il master trasmette il pacchetto con SEQ = 0 verso lo slave, che risponde con un pacchetto di acknowledge contenente il proprio ID e SEQ uguale a quello del pacchetto a cui si riferisce. Il pacchetto potrà contenere eventuali dati di risposta che lo slave debba trasferire verso il master. Ricevuto l'acknowledge, il master può trasmettere il pacchetto successivo che dovrà avere SEQ = 1. Il pacchetto ancora successivo tornerà ad avere SEQ = 0.

Capitolo 4

Realizzazione e collaudo dell'interfaccia

Dopo aver studiato il funzionamento dell'interfaccia è stato fatto il disegno dello schematico di una singola interfaccia con il programma Menthor Graphics. Nelle seguenti figure 4.1 e 4.2 sono rappresentati sia lo schema elettrico di una singola interfaccia che il blocco base di un modulino.

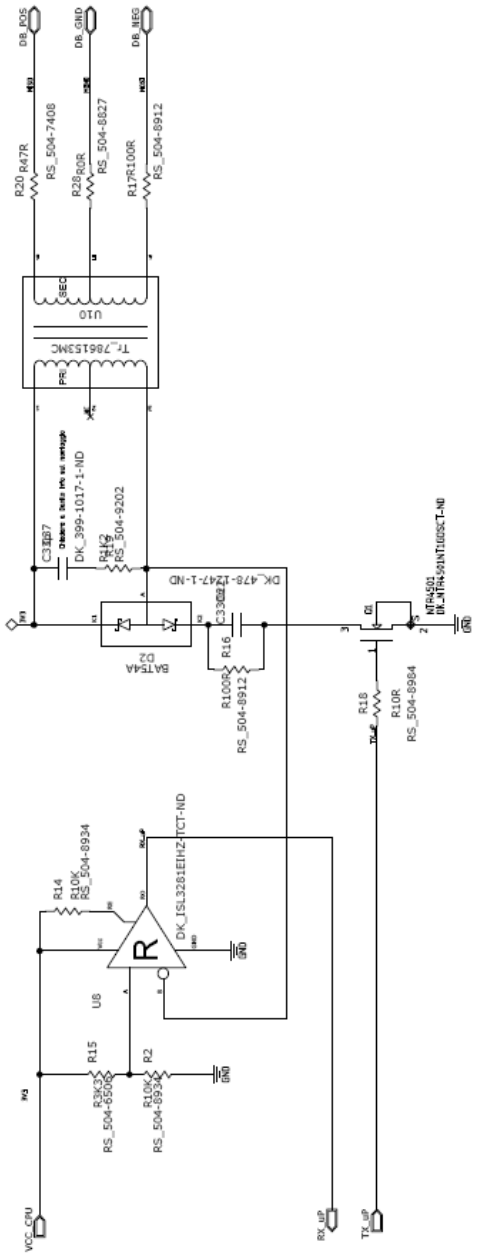


Figura 4.1 Schema elettrico interfaccia definitivo

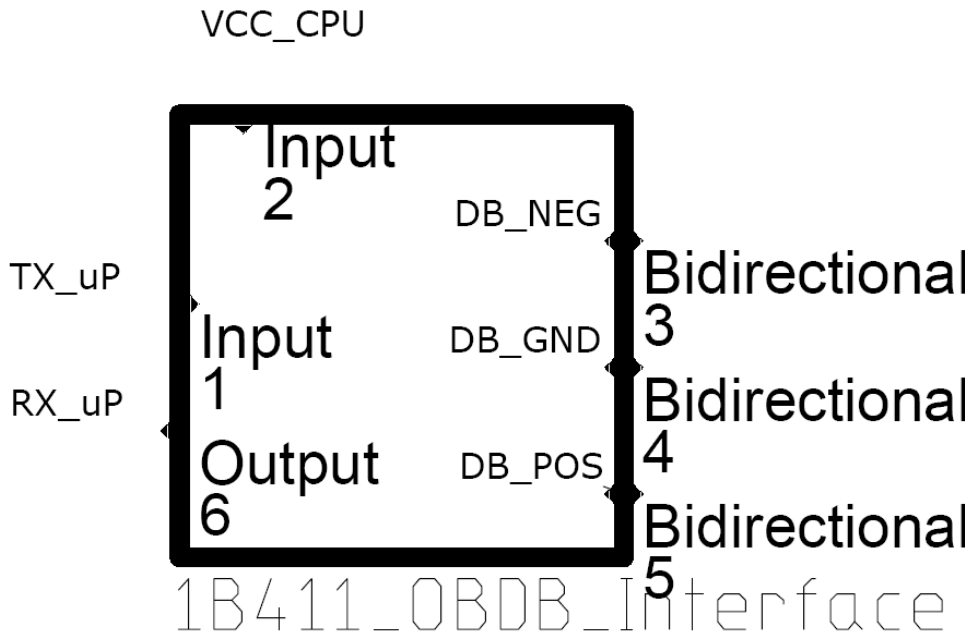


Figura 4.2 Schema Blocco interfaccia 1B411_OBDB

Nella figura 4.3 seguente vi è la tabella con l'elenco dei componenti per la realizzazione di una interfaccia.

Sono stati scelti dei connettori a pin singoli per avere facilità di test a livello elettrico, sia per i connettori di alimentazione che per i connettori del bus differenziale. Le schede hanno una dimensione di 4 x 3 cm.

Quantita'	Componenti	Valore
1	CONNETTORE	5 PIN
1	CONNETTORE	3 PIN
1	MAX3281E (A.O.)	
1	NTR4501 (NMOS)	
1	BAT54A (DIODI)	
1	78615/3 (TRASFORMATORE)	
1	R 15	3.3 KΩ
2	R 2, R 14	10 KΩ
1	R 18	10 Ω
2	R 16, R 17	100 Ω
1	R 19	1.2 KΩ
1	R 20	47 Ω
1	R 28	0 Ω
1	C 22	330 nF
1	C 37	33 pF

Figura 4.3 Elenco componenti di un'interfaccia

4.1 PCB

Una volta completato lo schema si è passati alla realizzazione del PCB con il software Expedition sempre della Mentor Graphics.

Nel progetto del layout si è stati attenti a fare i wire delle alimentazioni più spessi per evitare un'interruzione delle alimentazioni, sono stati fatti 2 strati e aggiunti dei via per avere un piano di massa. Finito il progetto è stato stampato il file Gerber ed sono state realizzate fisicamente nel laboratorio hardware le tre schede con relativo montaggio e saldatura a mano dei componenti. Terminato ciò, sono state testate tutte le schede per verificarne l'effettivo funzionamento.

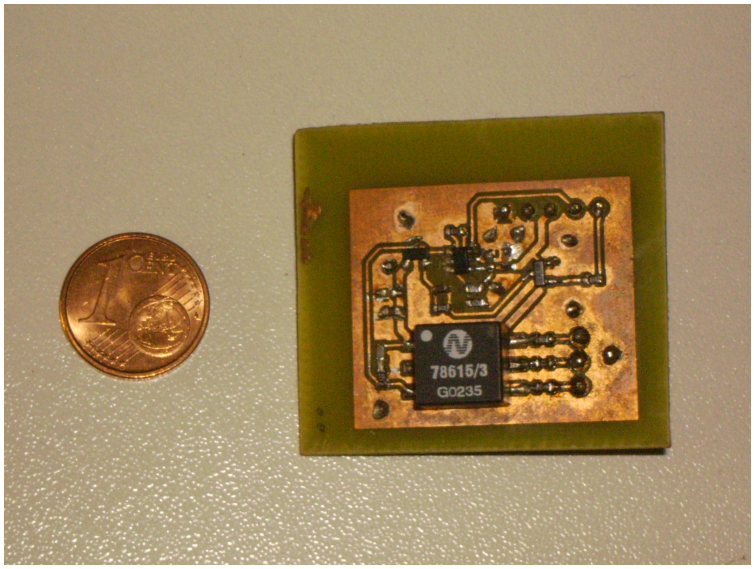


Figura 4.4 Foto singola interfaccia lato componenti

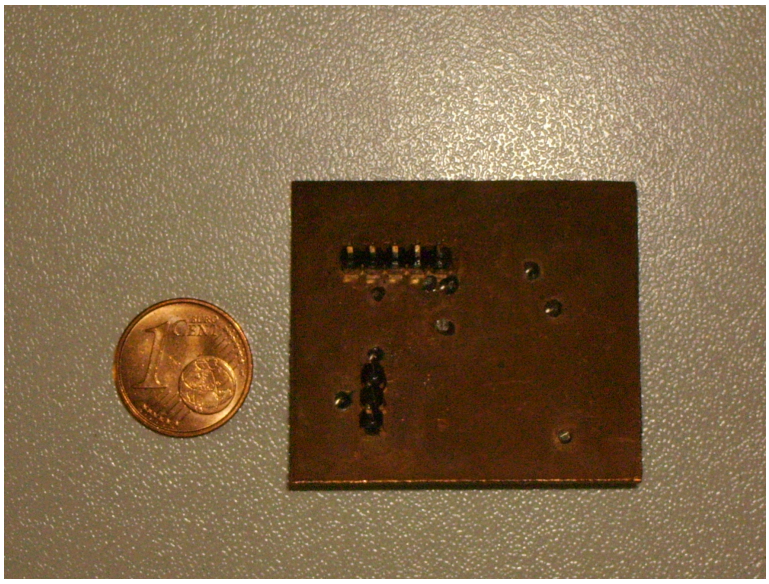


Figura 4.5 Foto singola interfaccia lato connettori

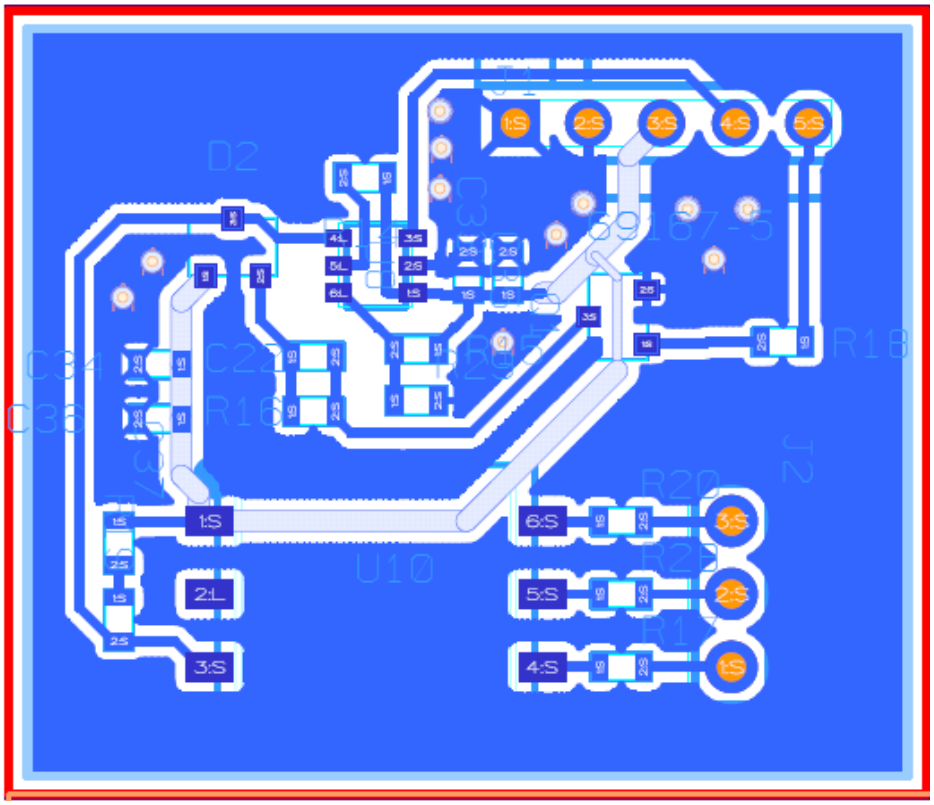


Figura 4.6 PCB interfaccia

Prima di eseguire il test a livello elettrico sono stati fatti dei piccoli connettori per poter collegare il bus differenziale dei modulini. Ogni connettore ha 3 pin, due per il +DB e -DB, l'altro per GND. La scheda ha un altro connettore di 5 pin al quale collegare l'alimentazione (Vcc e Gnd) e i pin di Tx e Rx da collegare al processore.

Lo scopo di questo lavoro è di simulare la comunicazione di moduli Aramis su un circuito ibrido. Questo sistema è stato sviluppato su una scheda millefori nella quale sono stati inseriti oltre ai terminali di alimentazione anche 4 connettori per MSP430FG come si può vedere dalla seguente figura 4.7.

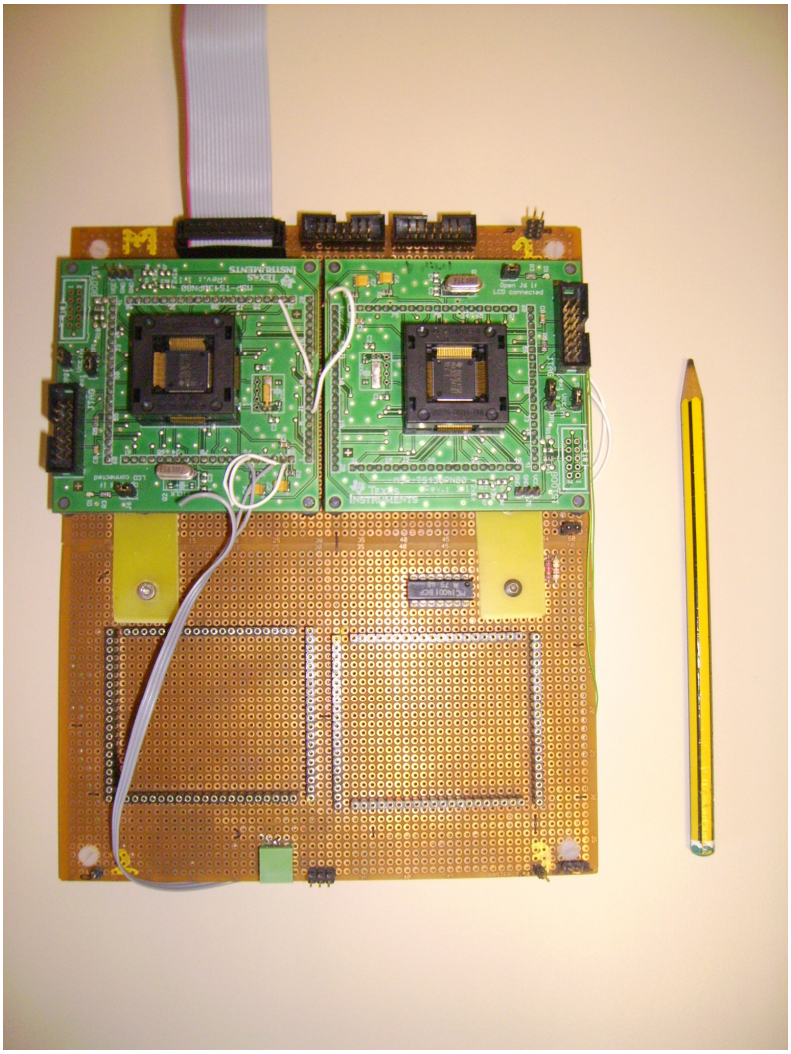


Figura 4.7 Foto scheda ibrida

Allo scopo di far comunicare ogni processore con gli altri è stato aggiunto un connettore al quale sono collegati i pin di trasmissione e ricezione della UART dei 4 processori MSP430 della Texas Instruments.

4.2 Circuito di test

Il test consisteva nel collegare le interfacce assieme tramite il bus, segnali 15 16 e 17, trasmettere un segnale sul pin Tx della prima interfaccia(segnale10) e verificare tramite un oscilloscopio se sul pin Rx delle altre schede (segnali 6 e 7) ricevamo effettivamente la stessa forma d'onda del segnale trasmesso. Questo naturalmente facendo ruotare le schede, quindi verificando il funzionamento in trasmissione e ricezione di tutte le schede.

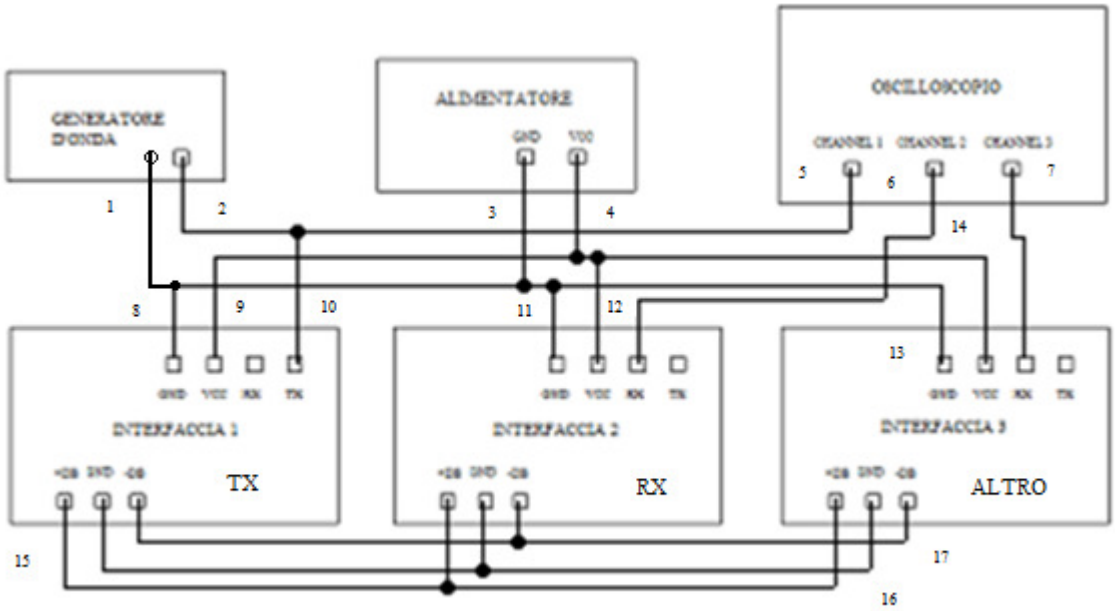


Figura 4.8 Schema di test

Il segnale di prova trasmesso (segnale 2) ha le seguenti caratteristiche:

- Frequenza 1Mhz;
- Larghezza d'impulso 100ns;
- Tempo di salita e discesa 5ns;
- Vpp 3V;
- Tensione di alimentazione della scheda 3.3V.

Dopo aver verificato il funzionamento di tutte le interfacce, si è passati al test del funzionamento del bus con i vari tipi di guasti che possono esserci sul bus.
Nel nostro caso abbiamo utilizzato la stessa modalità di test precedente e creato sul bus 4 tipi di guasti come in figura 3.2. Come si può vedere dalla figura 4.8, con l'oscilloscopio digitale siamo andati a vedere le tensioni in ingresso all'amplificatore operazionale che in questo caso funzionava da comparatore.

La figura 4.9 seguente mostra oltre al segnale trasmesso (linea gialla, canale1), anche le tensioni in ingresso al comparatore (linea viola e linea verde) senza guasti provocati, quindi in condizioni normali.

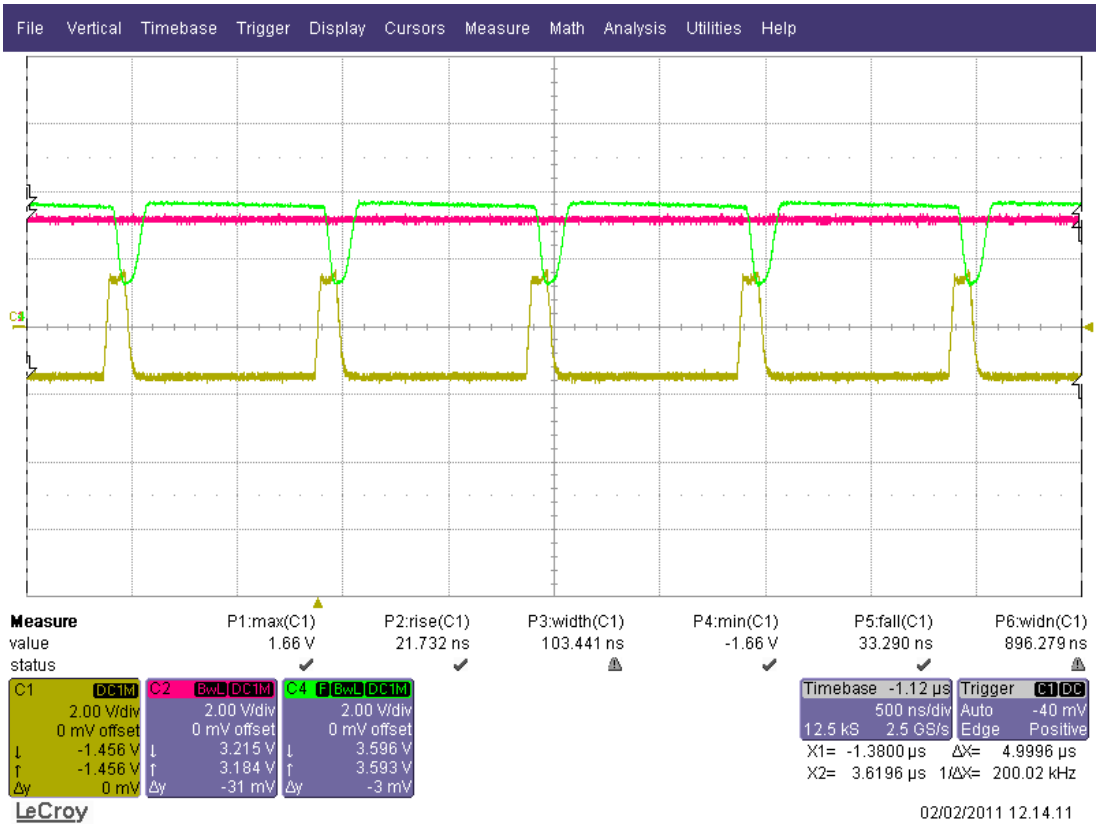


Figura 4.9 Segnali in ingresso al comparatore senza guasti

La figura 4.10 ci mostra il segnale trasmesso e ricevuto quando il bus lavora senza guasti. Sul canale 1 (linea gialla, segnale 10) abbiamo il segnale trasmesso ad una interfaccia dal generatore di segnale, mentre sui canali 2(linea viola, segnale 6) e 4(linea verde, segnale 7) ci sono i segnali che riceviamo sul pin Rx delle altre interfacce.

Figura 4.10 Segnale di prova trasmesso e segnali ricevuti sul pin RX dalle interfacce(segnali 6 e 7)

In seguito sono state fatte varie prove provocando guasti di tipo *c* e *a* cortocircuitando o staccando i segnali +DB (segnale 15) e i segnali –DB (segnale 17). In tutti questi casi si poteva vedere che cambiava il margine di rumore della tensione ma andando sui pin di ricezione RX delle schede il segnale di prova si riceveva ugualmente come possiamo vedere in figura.

La figura 4.11 ci mostra il caso più critico. In questa situazione si è provocato un guasto di tipo *a* cortocircuitando il segnale +DB (segnali 15 e 16) ed il ricevitore non funzionava perché i 2 segnali in ingresso all'amplificatore operazionale non venivano comparati. Per risolvere questo problema sono stati modificati i valori delle resistenze R15 e R2 del partitore di tensione in ingresso all'operazionale per far aumentare il segnale a tensione fissa (segnale verde in figura 4.11). Aumentando questa tensione fissa di riferimento la comparazione avviene con successo ed il bus è in grado di ricevere il segnale come possiamo vedere in figura 4.12

.

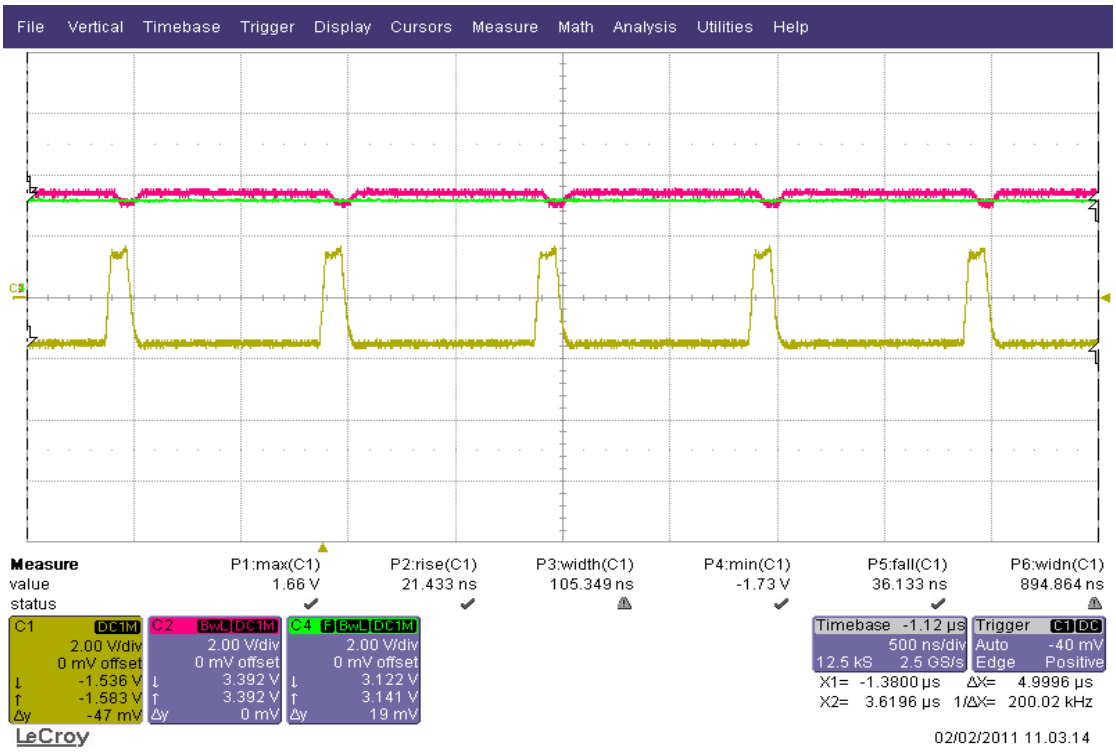


Figura 4.11 Segnali in ingresso al comparatore con guasto di tipo *a* su +DB (segnali 15 e 16)

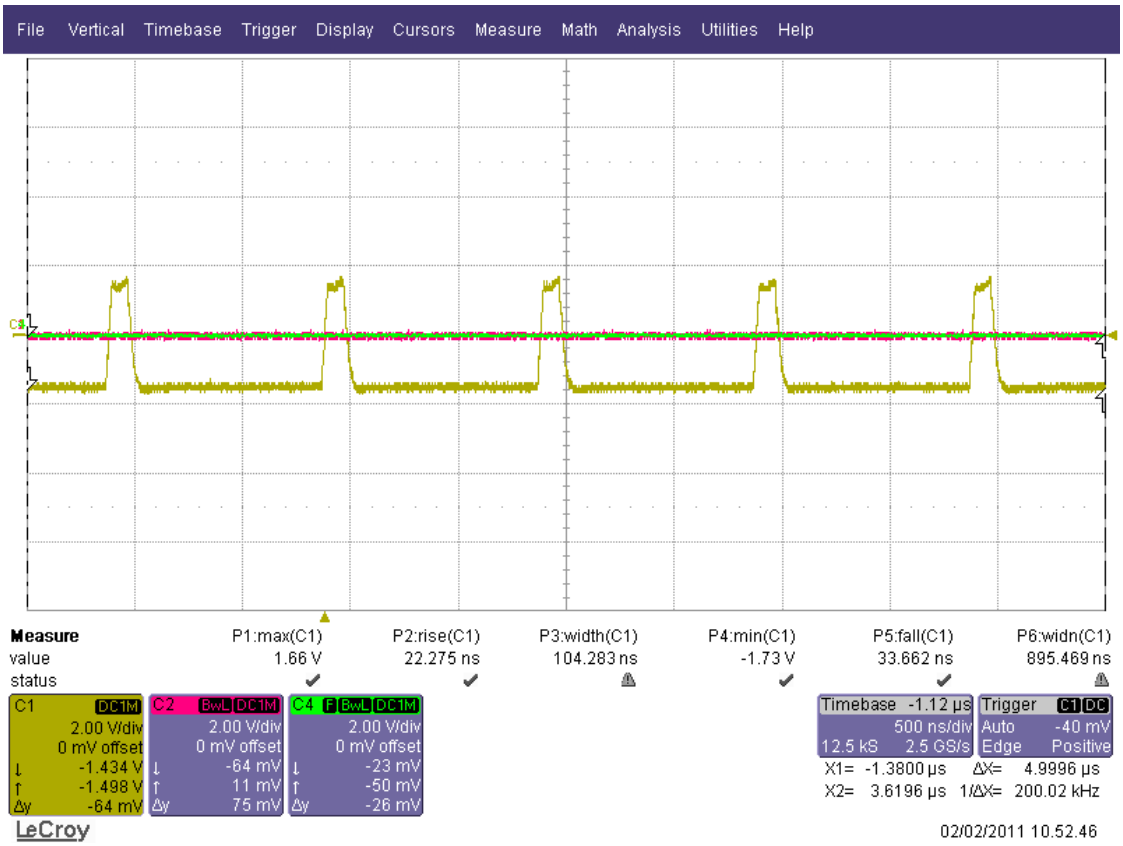


Figura 4.12 Segnali sui pin RX prima della correzione dei partitori (segnali 6 e 7)

4.3 Test sulla comunicazione

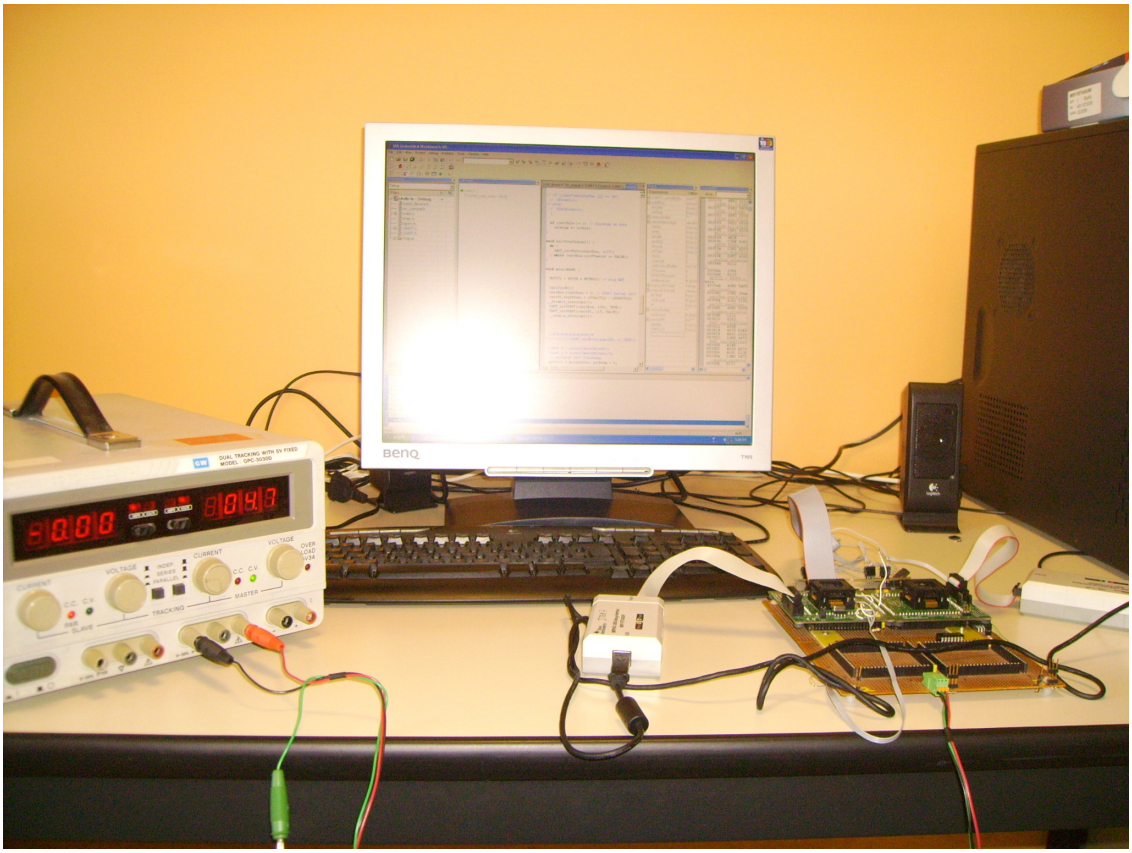


Figura 4.13 Foto test di comunicazione

4.4 Implementazione software

Per l'implementazione software del livello Collegamento è stato inzialmente usato lo stesso codice in linguaggio ANSI C usato nella tesi precedente sui processori MSP430. Per la compilazione del codice è stato utilizzato l'ambiente di sviluppo IAR Embedded Workbench della IAR System.
Il codice, presente in Appendice, è scritto con una tecnica pseudo object-oriented mantenendo la sintassi ANSI C per avere compatibilità con compilatori esistenti.
La classe UART, presente nel codice, si occupa dell'inizializzazione e dell'interazione con le periferiche di comunicazione asincrona nel processore.
La classe contiene le seguenti funzioni:

- **initUART:** inizializza i registri di configurazione della UART del processore;

- **sendByte:** invia un singolo byte;
- **sendString:** invia una stringa di caratteri terminata da una carattere NULL;
- **sendMem:** invia un area di memoria di *length* byte;
- **recvByte:** riceve un singolo byte;
- **recvMem:** riceve un numero *buffLength* di byte nell'area di memoria che parte da *buffPtr*.

Nella funzione di inizializzazione viene controllato il parametro *irdaMode* che, se vero, richiede l'attivazione della codifica/decodifica RZI ad impulsi.
Le funzioni di trasmissione hanno un parametro *blocking* che rende l'esecuzione della funzione sincrona se è vero, cioè se la funzione è chiamata, non ritornerà fino a quando l'invio non sarà completato. Nel caso di parametro blocking falso, l'invio sarà invece asincrono e sarà compito della classe verificare il completamento della trasmissione prima di un nuovo invio. Durante il periodo di attesa dell'invio sincrono, il processore viene posto nella modalità a basso consumo Low Power Mode 1 (LPM1) che disabilita la CPU e diversi clock interni lasciando però attivo il clock SMCLK, necessario al funzionamento della UART.
In questo codice la classe UART viene istanziata due volte ma nel nostro caso utilizziamo solo la prima, cioè l'*uartBus* per la comunicazione sul bus e sarà inizializzata con il parametro irdaMode vero.

Il passo successivo è stata la verifica della funzionalità del canale e della scheda ibrida. Dopo aver alimentato la scheda e connesso le interfacce tramite il connettore comune, sono stati collegati i processori MSP430 al pc tramite due USB-Debug-Interface MSP-FET430UIF per la compilazione del codice. Sono stati utilizzati due processori MSP430F2418 e per la compilazione del codice è stato utilizzato l'ambiente di sviluppo IAR Embedded Workbench della IAR System.
Lo scopo iniziale del test consiste nel far comunicare i due processori tramite il canale. Per far questo sono stati creati 2 progetti chiamati "obdb-tx" e "obdb-rx".
Entrambi i progetti hanno i seguenti file:

- Main.c
- Main.h
- UART.c
- UART.h

Main.c è il programma principale di test e contiene:

- la funzione *initClock* che inizializza i vari registri del clock;

- la funzione *safesand* che controlla se il canale è libero;
- la funzione *sendMessage* che manda i dati secondo il formato del pacchetto (ID, Datalength, Data, Checksum) attraverso una funzione *sendByte*;
- la funzione *rxFunc* che riceve sempre i dati dal canale ed esegue un controllo sul checksum per verificare la corretta ricezione dei dati;
- la funzione *main* nella quale vengono eseguite le funzioni di inizializzazione del clock e della uart e poi tramite la definizione di *MONITOR* si decide se si è in modalità di ricezione o trasmissione. Se si è in modalità di ricezione viene eseguita solo la *rxFunc*, se si è in modalità di trasmissione viene mandato come dato la lettera "H" attraverso la funzione *sendMessage*.

Nel file *UART.c* vengono descritte le seguenti funzioni:

- *UART_sendByte* che ha il compito di inviare un byte;
- *UART_recvByte* che ha il compito di ricevere un byte;
- *UART_initUART* che inizializza i registri di configurazione della UART;
- *UART_isrAB0_tx* e *UART_isrAB1_tx* sono le Interrupt Service Routine di trasmissione delle due UART del processore;
- *UART_isrAB0_rx* e *UART_isrAB1_rx* sono le Interrupt Service Routine di ricezione delle due UART del processore.

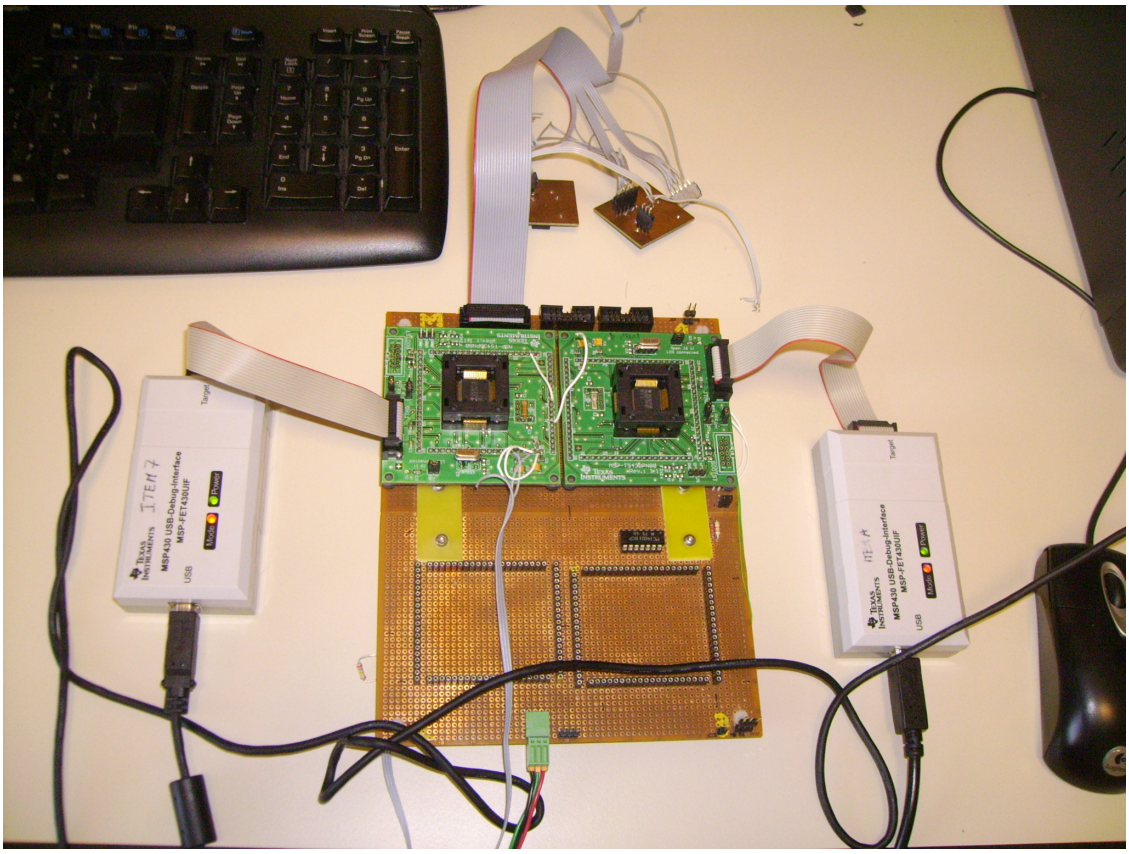


Figura 4.14 Collegamenti per il test di comunicazione

4.4.1 Comunicazione unidirezionale

La prima prova da fare per testare l'interfaccia di comunicazione e la funzionalità del canale è trasmettere un dato da un processore all'altro e verificare se la ricezione è andata a buon fine.
Chiamiamo per semplicità la "obdb_tx" come "unità 1" e la "obdb_rx" come "unità 2". Per rendere l'unità 1 trasmissiva non è stata definita la macro MONITOR quindi viene debuggato solo il codice nel quale c'è la funzione *sendMessage* con la quale viene trasmessa la lettera "H". Per avere l'unità 2 in modalità di ricezione è stata definita la macro MONITOR e quindi è stato debaggato solo il codice nel quale viene eseguita la funzione di ricezione.
Dopo aver debuggato entrambi i progetti ed aver inviato i dati secondo il formato del pacchetto (ID, Datalength, Data, Checksum), si è andati a vedere nell'array *MessBuffer* dell'unità 2 cosa abbiamo ricevuto ed in effetti se si và a vedere in "Data" c'è la lettera "H", quindi la comunicazione funziona correttamente.

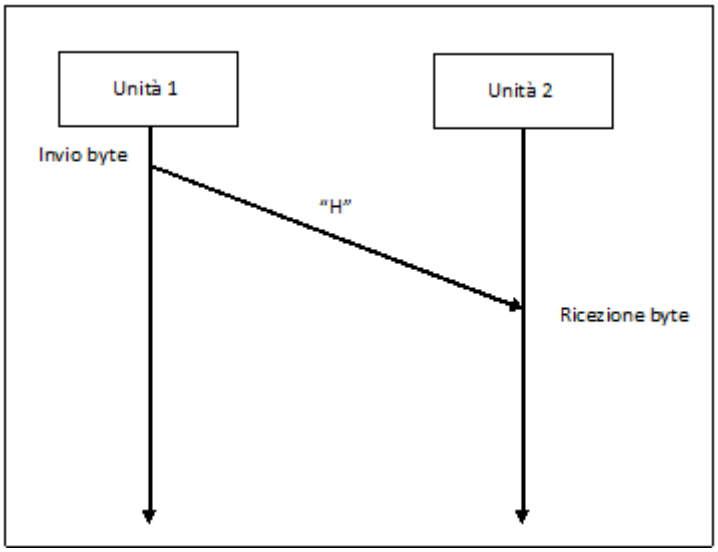


Figura 4.15 Esempio trasmissione unidirezionale

4.4.2 Comunicazione bidirezionale

Il secondo test consisteva nel trasmettere dall'unità 1 un byte (nel nostro caso sempre il carattere "H"), successivamente l'unità 2 riceve e ritrasmette ciò che ha ricevuto e andiamo a leggere nell'unità 1 se è lo stesso byte trasmesso.

Per far ciò si è dovuto modificare il codice per rendere la comunicazione bidirezionale e quindi per avere la *rxFunc* eseguita su entrambe le unità.
Per far ritrasmettere dall'unità 2 ciò che ha ricevuto è stata inserita una nuova *sendMessage* che ritrasmette il contenuto di Data del proprio *MessBuffer*.

Dopo aver debuggato i due progetti, per verificare se la comunicazione tra le due unità è andata a buon fine si è andato a vedere nella funzione di ricezione dell'unità 1, in particolar modo nel campo Data dell'array *MessBuffer*, se si è ricevuto il carattere "H". Una volta verificato ciò, si può dire che il test è risultato positivo.

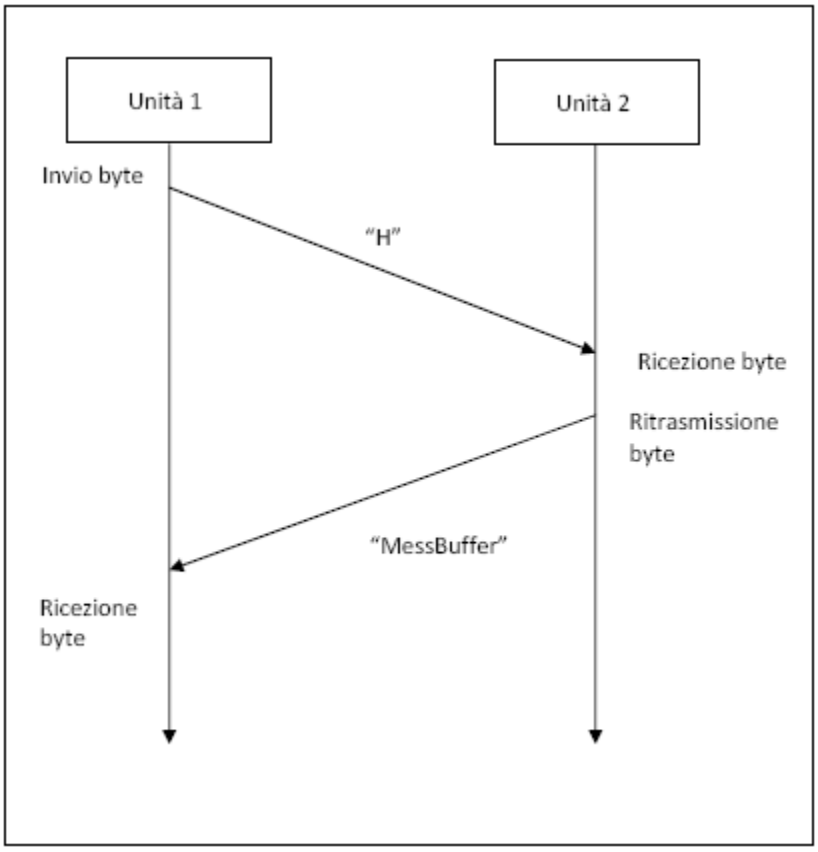


Figura 4.16 Esempio comunicazione bidirezionale

4.4.3 Comunicazione in Loop

L'ultimo test consisteva nel creare un loop e fare numerose misure. Per fare ciò bisogna integrare il codice del test precedente facendo trasmettere continuamente l'unità 1. Nella funzione di ricezione dell'unità 1 viene verificato se è stato ricevuto lo stesso byte che era stato tramesso in precedenza come nel test di comunicazione bidirezionale.
La probabilità di errore viene calcolata nell'unità 2 controllando il checksum per ogni invio di dato.
.

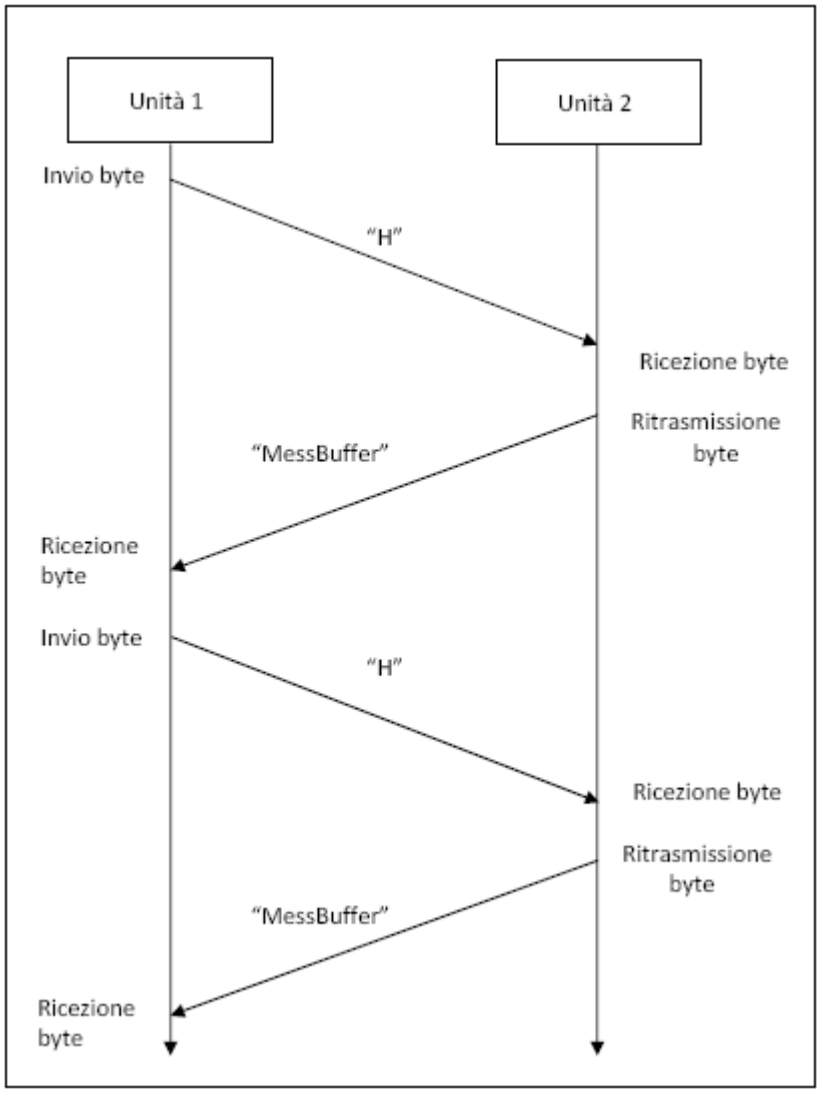


Figura 4.17 Esempio comunicazione in loop

Dopo aver fatto un ciclo di misure in condizioni normali si è dovuta verificare la tolleranza ai guasti del sistema, provocandoli sui conduttori del bus.
La prima prova è stata fatta togliendo dai connettori i fili del bus differenziale e debuggando il sistema si è verificato il non funzionamento.
Successivamente sono state effettuate le altre misure provocando i guasti di tipo *c* e *a* sul bus differenziale staccando o cortocircuitando i fili +DB e –DB del bus. Le misure con i guasti provocati sono state effettuate in modalita loop e sono state inserite nella tabella di figura 4.7.
Dalla tabella si può notare come il sistema funzioni correttamente anche con la presenza di singoli guasti su un filo del bus differenziale. Questo rende la comunicazione molto affidabile soprattutto in ambienti come quelli spaziali dove la telemetria e lo scambio di dati è fondamentale.

	NUMERO CICLI NEL LOOP	NUMERO DATI CORRETTI	NUMERO ERRORI
CONDIZIONE NORMALE	57446	1.009,79 Kbyte	0
ASSENZA BUS	57446	0	57446
GUASTO TIPO C SU +DB	57787	1.015,78 Kbyte	0
GUASTO TIPO C SU -DB	56691	996,52 Kbyte	0
GUASTO TIPO A SU +DB	57894	1.017,66 Kbyte	0
GUASTO TIPO A SU -DB	58211	1.023,24 Kbyte	0

Figura 4.18 Tabella misure in loop con e senza guasti

Capitolo 5

Conclusioni

L'argomento trattato in questo lavoro di tesi riguarda il sottosistema di comunicazione bus dati del satellite Aramis. Quest'ultimo è un sistema modulare che ha bisogno di far comunicare i vari moduli attraverso un sotto-sistema di comunicazione.

Si è iniziato studiando le tesi che precedentemente avevano trattato questo tema e quindi il primo passo è stato analizzare i vari standard esistenti di comunicazione e verificare quali vadano incontro alle specifiche richieste da Aramis. I requisiti principali richiesti sono alta affidabilità e tolleranza ai guasti usando componenti semplici ed economici.

La scelta è ricaduta su un bus di comunicazione differenziale che garantisca l'isolamento galvanico tra le mattonelle per mezzo di trasformatori ed utilizzi una codifica RZI.

Il passo successivo è stato sviluppare ed integrare un prototipo su una scheda millefori con 4 unità interconnesse fra di loro. Per fare ciò è stato aggiunto un connettore in grado di collegare le periferiche UART delle 4 unità (MSP430FG2418) alle interfacce con il canale di comunicazione.

Una volta realizzato lo schematico ed il relativo PCB sono state sviluppate e collaudate le schede di interfaccia modificando alcuni componenti rispetto allo schema precedente.

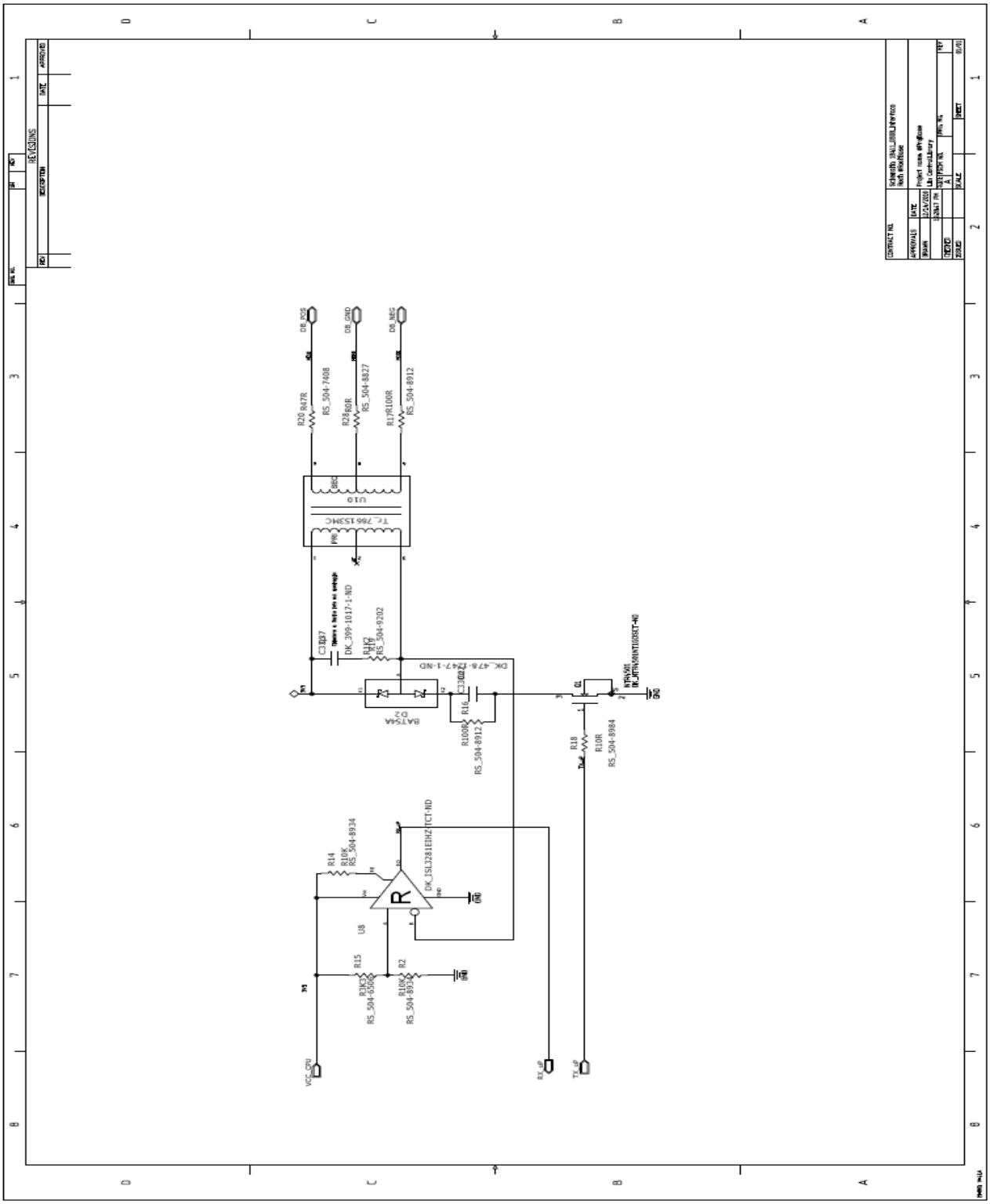
Verificato il funzionamento delle interfacce si è connesso il tutto per implementare il livello collegamento. La comunicazione tra le unità avviene attraverso la trasmissione e ricezione di byte e modificando il codice si è arrivati ad avere un comunicazione in loop che avveniva con successo.

Il lavoro svolto in questo tempo mi ha permesso di apprendere nuove conoscenze in diversi settori, sia in campo elettronico che aerospaziale.

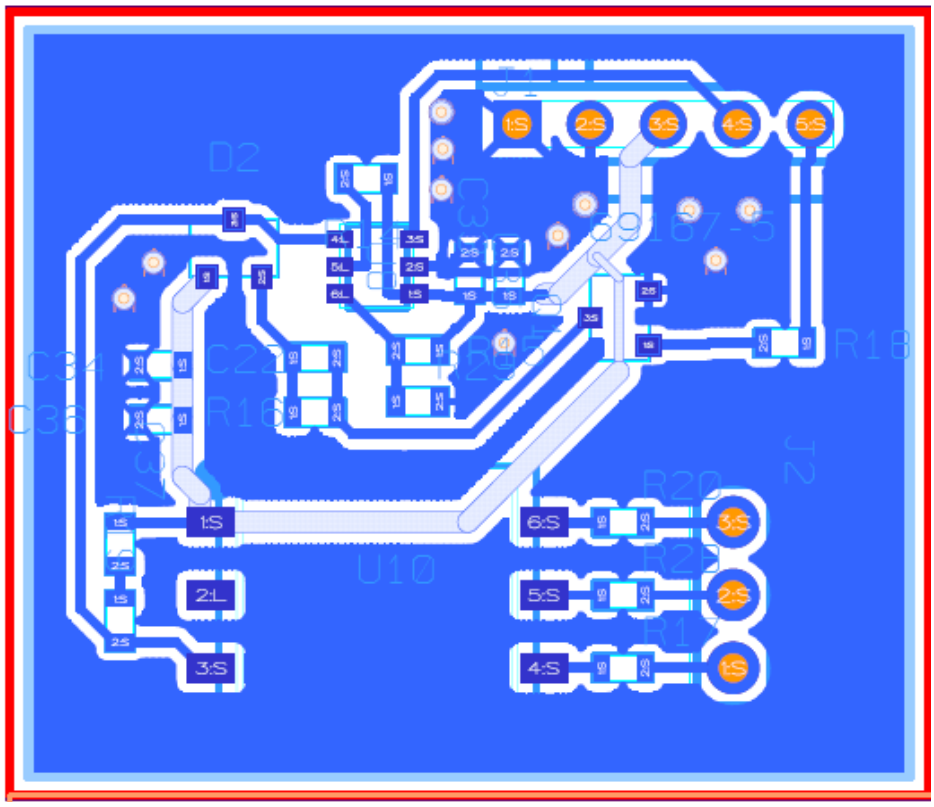
La programmazione di microcontrollori e la realizzazione e collaudo di schede elettroniche sono attività che non vengono affrontate in modo principale nella formazione accademica ma che sono importanti in un progetto complesso e grande come il satellite Aramis.

Appendice

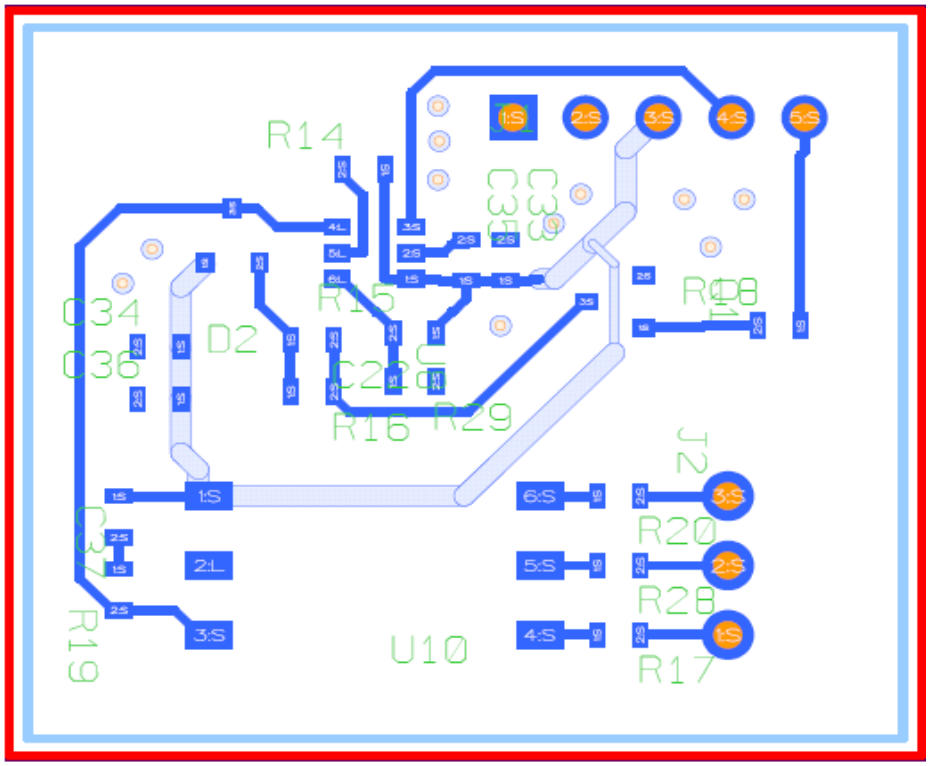
A Schema elettrico e PCB



Schema elettrico interfaccia



Top del PCB



Bottom del PCB

B. Codice sorgente

B.1 Codice comunicazione unidirezionale

CODICE OBDB_TX

Main.c

```
#include "main.h"
//#define COLLIDER
//#define MONITOR
UART uartBus, uartPC;
volatile char hexstr[] = "0123456789ABCDEF";
volatile char messBuffer[50];
volatile char *messPtr;
volatile char rxCksum;
volatile char rxMessLen;
volatile uint messOkCount[8], messErrCount[8];

/*void WaitTime(){
  unsigned int delay = 0xFF;
  do{
  __no_operation();
  __no_operation();
  __no_operation();
  __no_operation();
  } while (delay-- > 0);
}*/

void initClock() {
  BCSCTL1_bit.XT2OFF = 0; // XT2 on
  BCSCTL3_bit.XT2S1 = 1; // 4 MHz xtal
  BCSCTL3_bit.XT2S0 = 0;
  do { // wait for OFIFG staying cleared
    IFG1_bit.OFIFG = 0;
    unsigned int delay = 0xFF;
    while (delay-- > 0);
  } while (BCSCTL3_bit.XT2OF == 1);
  BCSCTL2 = SELM1 + SELS; // MCLK = XT2, SMCLK = XT2
}

boolean safeSend(char value) {
  char ret;
  while (uartBus.sendPtr < uartBus.stopPtr); // flush buffer
  // reset receive buffer
  uartBus.recvPtr = uartBus.recvBuffer;
  uartBus.readPtr = uartBus.recvPtr;
  UART_sendByte(&uartBus, value, FALSE);
//  do {
    ret = UART_recvByte(&uartBus, 0xFF);
//  } while (uartBus.recvTimeout == TRUE);
  if ((ret == value) && (uartBus.recvTimeout == FALSE))
    return TRUE;
  else
    return FALSE;
```

```
}

void sendMessage(char *data, uint length) {
  uint i;
  unsigned char cksum;
  //waitFreeChannel();TianShang
//#ifdef COLLIDER TianShang
  //if (safeSend(0x0F) == FALSE) { // someone else got the channel
//#else
  if (safeSend(0x1F) == FALSE) { // someone else got the channel
//#endif
   return; // TODO: proper id & listen
  }
  else {
    cksum = 0;
    //WaitTime();
    UART_sendByte(&uartBus, (length & 0xFF), TRUE);
   // WaitTime();
    for (i = 0; i < length; i++) {
       UART_sendByte(&uartBus, data[i], TRUE);
      cksum += data[i];
    }
   // WaitTime();
    UART_sendByte(&uartBus, cksum, TRUE);

//    uint del = 0xFF; // TODO: proper timeout
//    while (del-- > 0);
  }
}

volatile void rxFunc(char rxChar) {
  uint currByte;
  __disable_interrupt();
  currByte = (messPtr - messBuffer);
  *(messPtr++) = rxChar;
  if (currByte == 0) // id
    ;// TODO: id check
  else if (currByte == 1) // length byte
    rxMessLen = rxChar;
  else if (currByte >= rxMessLen+2) { // checksum
    // uartbus off!
    INST1_TXIFG = 0;
   IE2 &= ~(UCA0TXIE + UCA0RXIE);
    //IE2 = 3;
    __enable_interrupt();

    uint i = 0;
    while (messBuffer[0] >> i) { i++; }
    if (rxChar == rxCksum)
      messOkCount[i]++;
    else
      messErrCount[i]++;
    for (uint c = 0; c < 8; c++) {
      if ((messOkCount[c] != 0) || (messErrCount[c] != 0))
      {
        UART_sendString(&uartPC, "ID ", TRUE);
        UART_sendByte(&uartPC, c+0x30, TRUE);
        UART_sendString(&uartPC, ": ok ", TRUE);
```

```
        int s = sizeof(uint)*8-4;
        char nibble;
        do {
          nibble = (messOkCount[c] >> s) & 0x0F;
         UART_sendByte(&uartPC, hexstr[nibble], TRUE);
         //UART_sendByte(&uartPC, nibble+0x30, TRUE);
          s -= 4;
        } while (s >= 0);
//        UART_sendByte(&uartPC, (messOkCount[c] && 0xFF), TRUE);
//        UART_sendByte(&uartPC, ((messOkCount[c] >> 8 ) && 0xFF), TRUE);
        UART_sendString(&uartPC, ", err ", TRUE);
        s = sizeof(uint)*8-4;
        do {
          nibble = (messErrCount[c] >> s) & 0x0F;
          UART_sendByte(&uartPC, hexstr[nibble], TRUE);
          s -= 4;
        } while (s >= 0);
        UART_sendString(&uartPC, " | ", TRUE);
      }
    }
   // UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\n', TRUE);
    if (rxChar == rxCksum)
   OKcount++;
 else
   ERRORcount++;
    messPtr = messBuffer; // reset
    rxCksum = 0;
    currByte = 0;
    // uartbus on!
    uartBus.callback = 0;
    IE2 |= UCA0TXIE + UCA0RXIE;
    //IE2 = 0;
    //waitFreeChannel();  TianShang
    __disable_interrupt();
    uartBus.callback = rxFunc; // TODO: better init
  }
  if (currByte >= 2) // checksum on data
    rxCksum += rxChar;
}

void waitFreeChannel() {
  do {
    UART_recvByte(&uartBus, 0x3F);
  } while (uartBus.recvTimeout == FALSE);
}

void main(void) {

 WDTCTL = WDTPW + WDTHOLD; // stop WDT

initClock();

uartBus.regOffset = 0; // TODO: better init
  uartPC.regOffset = &UCA1CTL0 - &UCA0CTL0;
 __disable_interrupt();
  UART_initUART(&uartBus, 1000, TRUE);
```

```
  UART_initUART(&uartPC, 115, FALSE);
  __enable_interrupt();
//uint b = sizeof(messOkCount);
 //uint a = sizeof(messOkCount)/4;
 // callback init TianShang
  messPtr = messBuffer; rxCksum = 0;
 for (uint c = 0; c < 8; messOkCount[c++] = 0);
 for (uint c = 0; c < 8; messErrCount[c++] = 0);
 P1REN = 4; // P1_2 resistor enabled    TianShang
 P1SEL = 0; // io function
 P1DIR = 3; // P1_0 and P1_1 outputs
 P1OUT = 0; // P1_2 pulldown
  //waitFreeChannel(); // before callback enable

#ifdef MONITOR
 uartBus.callback = rxFunc; // TODO: better init
// UC1IE &= ~(UCA1TXIE);
 //UC1IE = 3;
#endif

  while (1) {
//    while (P1IN_bit.P1IN_2 == 0);
//#ifndef COLLIDER
//    __no_operation();
//#endif

#ifndef MONITOR
    sendMessage("Hello World!", 1);
  // WaitTime();
//    sendMessage("\x1F", 1);
#endif

#ifdef COLLIDER
    uint del = 0x3FF;
    while (del-- > 0);
#endif
  }
}
```

CODICE OBDB_RX

Main.c

```
#include "main.h"
//#define COLLIDER
#define MONITOR
UART uartBus, uartPC;
volatile char hexstr[] = "0123456789ABCDEF";
volatile char messBuffer[50];
volatile char *messPtr;
volatile char rxCksum;
volatile char rxMessLen;
volatile uint messOkCount[8], messErrCount[8];

/*void WaitTime(){
  unsigned int delay = 0xFF;
```

```
  do{
  __no_operation();
  __no_operation();

  } while (delay-- > 0);
}*/
void initClock() {
  BCSCTL1_bit.XT2OFF = 0; // XT2 on
  BCSCTL3_bit.XT2S1 = 1; // 4 MHz xtal
  BCSCTL3_bit.XT2S0 = 0;
  do { // wait for OFIFG staying cleared
    IFG1_bit.OFIFG = 0;
    unsigned int delay = 0xFF;
    while (delay-- > 0);
  } while (BCSCTL3_bit.XT2OF == 1);
  BCSCTL2 = SELM1 + SELS; // MCLK = XT2, SMCLK = XT2
}

boolean safeSend(char value) {
  char ret;
  while (uartBus.sendPtr < uartBus.stopPtr); // flush buffer
  // reset receive buffer
  uartBus.recvPtr = uartBus.recvBuffer;
  uartBus.readPtr = uartBus.recvPtr;
  UART_sendByte(&uartBus, value, FALSE);
//  do {
    ret = UART_recvByte(&uartBus, 0xFF);
//  } while (uartBus.recvTimeout == TRUE);
  if ((ret == value) && (uartBus.recvTimeout == FALSE))
    return TRUE;
  else
    return FALSE;
}

void sendMessage(char *data, uint length) {
  uint i;
  unsigned char cksum;
  //waitFreeChannel();TianShang
//#ifdef COLLIDER TianShang
  //if (safeSend(0x0F) == FALSE) { // someone else got the channel
//#else
  if (safeSend(0x1F) == FALSE) { // someone else got the channel
//#endif
   return; // TODO: proper id & listen
  }
  else {
    cksum = 0;
    //WaitTime();
    UART_sendByte(&uartBus, (length & 0xFF), TRUE);
   // WaitTime();
    for (i = 0; i < length; i++) {
       UART_sendByte(&uartBus, data[i], TRUE);
      cksum += data[i];
    }
   // WaitTime();
    UART_sendByte(&uartBus, cksum, TRUE);
    uint del = 0xFF; // TODO: proper timeout
    while (del-- > 0);
```

```
  }
}

volatile void rxFunc(char rxChar) {
  uint currByte;
  __disable_interrupt();
  currByte = (messPtr - messBuffer);
  *(messPtr++) = rxChar;
  if (currByte >= sizeof(messBuffer))
    messPtr = messBuffer;
  if (currByte == 0) // id
    ;// TODO: id check
  else if (currByte == 1) // length byte
    rxMessLen = rxChar;
  else if (currByte >= rxMessLen+2) { // checksum
    // uartbus off!
    INST1_TXIFG = 0;
   IE2 &= ~(UCA0TXIE + UCA0RXIE);
    //IE2 = 3;
    __enable_interrupt();
    uint i = 0;
    while (messBuffer[0] >> i) { i++; }
    if (rxChar == rxCksum)
      messOkCount[i]++;
    else
      messErrCount[i]++;
    for (uint c = 0; c < 8; c++) {
      if ((messOkCount[c] != 0) || (messErrCount[c] != 0))
      {
        UART_sendString(&uartPC, "ID ", TRUE);
        UART_sendByte(&uartPC, c+0x30, TRUE);
        UART_sendString(&uartPC, ": ok ", TRUE);
        int s = sizeof(uint)*8-4;
        char nibble;
        do {
          nibble = (messOkCount[c] >> s) & 0x0F;
         UART_sendByte(&uartPC, hexstr[nibble], TRUE);
         //UART_sendByte(&uartPC, nibble+0x30, TRUE);
          s -= 4;
        } while (s >= 0);
//        UART_sendByte(&uartPC, (messOkCount[c] && 0xFF), TRUE);
//        UART_sendByte(&uartPC, ((messOkCount[c] >> 8 ) && 0xFF), TRUE);
        UART_sendString(&uartPC, ", err ", TRUE);
        s = sizeof(uint)*8-4;
        do {
          nibble = (messErrCount[c] >> s) & 0x0F;
          UART_sendByte(&uartPC, hexstr[nibble], TRUE);
          s -= 4;
        } while (s >= 0);
        UART_sendString(&uartPC, " | ", TRUE);
      }
    }
   // UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\n', TRUE);
    messPtr = messBuffer; // reset
    rxCksum = 0;
    currByte = 0;
```

```
    // uartbus on!
    uartBus.callback = 0;
        IE2 |= UCA0TXIE + UCA0RXIE;     //IE2 = 0;
    //waitFreeChannel();  TianShang
    __disable_interrupt();
    uartBus.callback = rxFunc; // TODO: better init
  }
  if (currByte >= 2) // checksum on data
    rxCksum += rxChar;
}

void waitFreeChannel() {
  do {
    UART_recvByte(&uartBus, 0x3F);
  } while (uartBus.recvTimeout == FALSE);
}

void main(void) {

WDTCTL = WDTPW + WDTHOLD; // stop WDT

initClock();

uartBus.regOffset = 0; // TODO: better init
  uartPC.regOffset = &UCA1CTL0 - &UCA0CTL0;
 __disable_interrupt();
  UART_initUART(&uartBus, 1000, TRUE);
  UART_initUART(&uartPC, 115, FALSE);
  __enable_interrupt();
/////////////UART_sendByte(&uartPC, 1, TRUE);
 //uint b = sizeof(messOkCount);
 //uint a = sizeof(messOkCount)/4;
  // callback init TianShang
  messPtr = messBuffer; rxCksum = 0;
 for (uint c = 0; c < 8; messOkCount[c++] = 0);
 for (uint c = 0; c < 8; messErrCount[c++] = 0);

 P1REN = 4; // P1_2 resistor enabled    TianShang
 P1SEL = 0; // io function
 P1DIR = 3; // P1_0 and P1_1 outputs
 P1OUT = 0; // P1_2 pulldown
  //waitFreeChannel(); // before callback enable

#ifdef MONITOR
 uartBus.callback = rxFunc; // TODO: better init
 // UC1IE &= ~(UCA1TXIE);
 //UC1IE = 3;
#endif
  while (1) {
//    while (P1IN_bit.P1IN_2 == 0);
//#ifndef COLLIDER
//    __no_operation();
//    __no_operation();
//#endif

#ifndef MONITOR
    P1OUT_bit.P1OUT_0 = 0;
   P1OUT_bit.P1OUT_0 = 1;
```

```
   P1OUT_bit.P1OUT_0 = 0;
    sendMessage("Hello World!", 1);
  // WaitTime();
//    sendMessage("\x1F", 1);
#endif

#ifdef COLLIDER
    uint del = 0x3FF;
    while (del-- > 0);
#endif
  }
}
```

B.2 Codice comunicazione in loop

CODICE UNITA' 1

Main.c

```
#include "main.h"

//#define COLLIDER
#define MONITOR

UART uartBus, uartPC;

volatile char hexstr[] = "0123456789ABCDEF";
volatile char messBuffer[50];
volatile char *messPtr;
volatile char rxCksum;
volatile char rxMessLen;
volatile uint messOkCount[8], messErrCount[8];
volatile uint finito = 0;
volatile uint ricevuti = 0;
volatile uint OKcount = 0;
volatile uint ERRORcount = 0;

/*void WaitTime(){
  unsigned int delay = 0xFF;
  do{
  __no_operation();
  __no_operation();
  __no_operation();

  } while (delay-- > 0);
}*/
void initClock() {
  BCSCTL1_bit.XT2OFF = 0; // XT2 on
  BCSCTL3_bit.XT2S1 = 1; // 4 MHz xtal
  BCSCTL3_bit.XT2S0 = 0;
  do { // wait for OFIFG staying cleared
    IFG1_bit.OFIFG = 0;
    unsigned int delay = 0xFF;
    while (delay-- > 0);
  } while (BCSCTL3_bit.XT2OF == 1);
  BCSCTL2 = SELM1 + SELS; // MCLK = XT2, SMCLK = XT2
}

boolean safeSend(char value) {
  char ret;
  while (uartBus.sendPtr < uartBus.stopPtr); // flush buffer
  // reset receive buffer
  uartBus.recvPtr = uartBus.recvBuffer;
  uartBus.readPtr = uartBus.recvPtr;
  UART_sendByte(&uartBus, value, FALSE);
//  do {
    ret = UART_recvByte(&uartBus, 0xFF);
//  } while (uartBus.recvTimeout == TRUE);
  if ((ret == value) && (uartBus.recvTimeout == FALSE))
    return TRUE;
  else
    return FALSE;
}

void sendMessage(char *data, uint length) {
  uint i;
```

```
  unsigned char cksum;
  //waitFreeChannel();TianShang
//#ifdef COLLIDER TianShang
  //if (safeSend(0x0F) == FALSE) { // someone else got the channel
//#else
  if (safeSend(0x1F) == FALSE) { // someone else got the channel
//#endif
   return; // TODO: proper id & listen
  }
  else {
    cksum = 0;
    //WaitTime();
    UART_sendByte(&uartBus, (length & 0xFF), TRUE);
   // WaitTime();
    for (i = 0; i < length; i++) {
       UART_sendByte(&uartBus, data[i], TRUE);
      cksum += data[i];
    }
   // WaitTime();
    UART_sendByte(&uartBus, cksum, TRUE);
//    uint del = 0xFF; // TODO: proper timeout
//    while (del-- > 0);
  }
}

volatile void rxFunc(char rxChar) {
  uint currByte;
  __disable_interrupt();
  currByte = (messPtr - messBuffer);
  *(messPtr++) = rxChar;
  if (currByte == 0) // id
    ;// TODO: id check
  else if (currByte == 1) // length byte
    rxMessLen = rxChar;
  else if (currByte >= rxMessLen+2) { // checksum

    // uartbus off!
    INST1_TXIFG = 0;
   IE2 &= ~(UCA0TXIE + UCA0RXIE);
    //IE2 = 3;
    __enable_interrupt();
    uint i = 0;
    while (messBuffer[0] >> i) { i++; }
    if (rxChar == rxCksum)
      messOkCount[i]++;
    else
      messErrCount[i]++;
    for (uint c = 0; c < 8; c++) {
      if ((messOkCount[c] != 0) || (messErrCount[c] != 0))
      {
        UART_sendString(&uartPC, "ID ", TRUE);
        UART_sendByte(&uartPC, c+0x30, TRUE);
        UART_sendString(&uartPC, ": ok ", TRUE);
        int s = sizeof(uint)*8-4;
        char nibble;
        do {
          nibble = (messOkCount[c] >> s) & 0x0F;
         UART_sendByte(&uartPC, hexstr[nibble], TRUE);
         //UART_sendByte(&uartPC, nibble+0x30, TRUE);
          s -= 4;
        } while (s >= 0);
//        UART_sendByte(&uartPC, (messOkCount[c] && 0xFF), TRUE);
//        UART_sendByte(&uartPC, ((messOkCount[c] >> 8 ) && 0xFF), TRUE);
        UART_sendString(&uartPC, ", err ", TRUE);
        s = sizeof(uint)*8-4;
        do {
```

```
          nibble = (messErrCount[c] >> s) & 0x0F;
          UART_sendByte(&uartPC, hexstr[nibble], TRUE);
          s -= 4;
        } while (s >= 0);
        UART_sendString(&uartPC, " | ", TRUE);
      }
    }

   // UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\n', TRUE);
   ricevuti++;
    if (rxChar == rxCksum)
   OKcount++;
 else
   ERRORcount++;
    messPtr = messBuffer; // reset
    rxCksum = 0;
    currByte = 0;
    // uartbus on!
    uartBus.callback = 0;
    IE2 |= UCA0TXIE + UCA0RXIE;
    //IE2 = 0;
    //waitFreeChannel();  TianShang
    __disable_interrupt();
    uartBus.callback = rxFunc; // TODO: better init
 finito = 1;
  }
  if (currByte >= 2) // checksum on data
    rxCksum += rxChar;
}

void waitFreeChannel() {
  do {
    UART_recvByte(&uartBus, 0x3F);
  } while (uartBus.recvTimeout == FALSE);
}

void main(void) {

  WDTCTL = WDTPW + WDTHOLD; // stop WDT

initClock();

uartBus.regOffset = 0; // TODO: better init
uartPC.regOffset = &UCA1CTL0 - &UCA0CTL0;
 __disable_interrupt();
  UART_initUART(&uartBus, 1000, TRUE);
  UART_initUART(&uartPC, 115, FALSE);
  __enable_interrupt();
//////////////UART_sendByte(&uartPC, 1, TRUE);
 //uint b = sizeof(messOkCount);
 //uint a = sizeof(messOkCount)/4;
  // callback init TianShang
  messPtr = messBuffer; rxCksum = 0;
 for (uint c = 0; c < 8; messOkCount[c++] = 0);
 for (uint c = 0; c < 8; messErrCount[c++] = 0);
  P1REN = 4; // P1_2 resistor enabled    TianShang
 P1SEL = 0; // io function
 P1DIR = 3; // P1_0 and P1_1 outputs
 P1OUT = 0; // P1_2 pulldown
  //waitFreeChannel(); // before callback enable

#ifdef MONITOR
 uartBus.callback = rxFunc; // TODO: better init
// UC1IE &= ~(UCA1TXIE);
```

```
 //UC1IE = 3;
#endif
  while (1) {
   finito = 0;
uartBus.callback = 0;
   sendMessage ("Hello World!", 1);
 uartBus.callback = rxFunc;
    uint del = 0xFFF;
    while (del-- > 0);
 while (finito == 0);
//    while (P1IN_bit.P1IN_2 == 0);
//#ifndef COLLIDER
//    __no_operation();
//    __no_operation();
//    __no_operation();
//#endif

#ifndef MONITOR
    sendMessage("Hello World!", 1);
  // WaitTime();

//    sendMessage("\x1F", 1);
#endif

#ifdef COLLIDER
    uint del = 0x3FF;
    while (del-- > 0);
#endif
  }
}
```

CODICE UNITA' 2

Main.c

```
#include "main.h"

//#define COLLIDER
#define MONITOR

UART uartBus, uartPC;

volatile char hexstr[] = "0123456789ABCDEF";
volatile char messBuffer[50];
volatile char *messPtr;
volatile char rxCksum;
volatile char rxMessLen;
volatile uint messOkCount[8], messErrCount[8];
volatile uint ritrasmetti = 0;
volatile uint ritrasmessi = 0;
/*void WaitTime(){
  unsigned int delay = 0xFF;
  do{
  __no_operation();
  __no_operation();
  __no_operation();
  __no_operation();
  } while (delay-- > 0);
}*/
void initClock() {
  BCSCTL1_bit.XT2OFF = 0; // XT2 on
  BCSCTL3_bit.XT2S1 = 1; // 4 MHz xtal
  BCSCTL3_bit.XT2S0 = 0;
```

```
  do { // wait for OFIFG staying cleared
    IFG1_bit.OFIFG = 0;
    unsigned int delay = 0xFF;
    while (delay-- > 0);
  } while (BCSCTL3_bit.XT2OF == 1);
  BCSCTL2 = SELM1 + SELS; // MCLK = XT2, SMCLK = XT2
}

boolean safeSend(char value) {
  char ret;
  while (uartBus.sendPtr < uartBus.stopPtr); // flush buffer
  // reset receive buffer
  uartBus.recvPtr = uartBus.recvBuffer;
  uartBus.readPtr = uartBus.recvPtr;
  UART_sendByte(&uartBus, value, FALSE);
//  do {
    ret = UART_recvByte(&uartBus, 0xFF);
//  } while (uartBus.recvTimeout == TRUE);
  if ((ret == value) && (uartBus.recvTimeout == FALSE))
    return TRUE;
  else
    return FALSE;
}

void sendMessage(char *data, uint length) {
  uint i;
  unsigned char cksum;
  //waitFreeChannel();TianShang
//#ifdef COLLIDER TianShang
  //if (safeSend(0x0F) == FALSE) { // someone else got the channel
//#else
  if (safeSend(0x1F) == FALSE) { // someone else got the channel
//#endif
   return; // TODO: proper id & listen
  }
  else {
    cksum = 0;
    //WaitTime();
    UART_sendByte(&uartBus, (length & 0xFF), TRUE);
   // WaitTime();
    for (i = 0; i < length; i++) {
       UART_sendByte(&uartBus, data[i], TRUE);
      cksum += data[i];
    }
   // WaitTime();
    UART_sendByte(&uartBus, cksum, TRUE);

    uint del = 0xFF; // TODO: proper timeout
    while (del-- > 0);
  }
}

volatile void rxFunc(char rxChar) {
  uint currByte;
  __disable_interrupt();
  currByte = (messPtr - messBuffer);
  *(messPtr++) = rxChar;
  if (currByte >= sizeof(messBuffer))
    messPtr = messBuffer;
  if (currByte == 0) // id
    ;// TODO: id check
  else if (currByte == 1) // length byte
    rxMessLen = rxChar;
  else if (currByte >= rxMessLen+2) { // checksum
    // uartbus off!
    INST1_TXIFG = 0;
```

```
   IE2 &= ~(UCA0TXIE + UCA0RXIE);
    //IE2 = 3;
    __enable_interrupt();
      uint i = 0;
    while (messBuffer[0] >> i) { i++; }
    if (rxChar == rxCksum)
      messOkCount[i]++;
    else
      messErrCount[i]++;
    for (uint c = 0; c < 8; c++) {
      if ((messOkCount[c] != 0) || (messErrCount[c] != 0))
      {
        UART_sendString(&uartPC, "ID ", TRUE);
        UART_sendByte(&uartPC, c+0x30, TRUE);
        UART_sendString(&uartPC, ": ok ", TRUE);
        int s = sizeof(uint)*8-4;
        char nibble;
        do {
          nibble = (messOkCount[c] >> s) & 0x0F;
         UART_sendByte(&uartPC, hexstr[nibble], TRUE);
         //UART_sendByte(&uartPC, nibble+0x30, TRUE);
          s -= 4;
        } while (s >= 0);
//        UART_sendByte(&uartPC, (messOkCount[c] && 0xFF), TRUE);
//        UART_sendByte(&uartPC, ((messOkCount[c] >> 8 ) && 0xFF), TRUE);
        UART_sendString(&uartPC, ", err ", TRUE);
        s = sizeof(uint)*8-4;
        do {
          nibble = (messErrCount[c] >> s) & 0x0F;
          UART_sendByte(&uartPC, hexstr[nibble], TRUE);
          s -= 4;
        } while (s >= 0);
        UART_sendString(&uartPC, " | ", TRUE);
      }
    }
   // UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\r', TRUE);
    UART_sendByte(&uartPC, '\n', TRUE);
    messPtr = messBuffer; // reset
    rxCksum = 0;
    currByte = 0;
    // uartbus on!
    uartBus.callback = 0;
        IE2 |= UCA0TXIE + UCA0RXIE;
        ritrasmessi++;
      ritrasmetti = 1;
    //IE2 = 0;
    //waitFreeChannel();  TianShang
    __disable_interrupt();
    uartBus.callback = rxFunc; // TODO: better init
     }
  if (currByte >= 2) // checksum on data
    rxCksum += rxChar;
}

void waitFreeChannel() {
  do {
    UART_recvByte(&uartBus, 0x3F);
  } while (uartBus.recvTimeout == FALSE);
}

void main(void) {

 WDTCTL = WDTPW + WDTHOLD; // stop WDT

initClock();
```

```
uartBus.regOffset = 0; // TODO: better init
  uartPC.regOffset = &UCA1CTL0 - &UCA0CTL0;
 __disable_interrupt();
  UART_initUART(&uartBus, 1000, TRUE);
  UART_initUART(&uartPC, 115, FALSE);
  __enable_interrupt();
//////////////UART_sendByte(&uartPC, 1, TRUE);

 //uint b = sizeof(messOkCount);
 //uint a = sizeof(messOkCount)/4;
  // callback init TianShang
  messPtr = messBuffer; rxCksum = 0;
 for (uint c = 0; c < 8; messOkCount[c++] = 0);
 for (uint c = 0; c < 8; messErrCount[c++] = 0);
 P1REN = 4; // P1_2 resistor enabled    TianShang
 P1SEL = 0; // io function
 P1DIR = 3; // P1_0 and P1_1 outputs
 P1OUT = 0; // P1_2 pulldown
  //waitFreeChannel(); // before callback enable

#ifdef MONITOR
 uartBus.callback = rxFunc; // TODO: better init
 // UC1IE &= ~(UCA1TXIE);
 //UC1IE = 3;
#endif
  while (1) {
      while (ritrasmetti == 0);
    uartBus.callback = 0;
    sendMessage((char *) messBuffer + 2, rxMessLen);
    uartBus.callback = rxFunc;
    ritrasmetti = 0;
    //    while (P1IN_bit.P1IN_2 == 0);
//#ifndef COLLIDER
//    __no_operation();
//    __no_operation();
//    __no_operation();
//#endif

#ifndef MONITOR
    P1OUT_bit.P1OUT_0 = 0;
   P1OUT_bit.P1OUT_0 = 1;
   P1OUT_bit.P1OUT_0 = 0;
    sendMessage("Hello World!", 1);
  // WaitTime();
//    sendMessage("\x1F", 1);
#endif

#ifdef COLLIDER
    uint del = 0x3FF;
    while (del-- > 0);
#endif
  }
}
```

CODICE UART PER ENTRAMBE LE UNITA’

```
#include "UART.h"

UART *iInstance[2];

void UART_sendByte(UART *this, char value, boolean blocking)
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--6edc429e:11860caa7eb:-8000:00000000000007EC begin
{
  if (this->sendPtr < this->stopPtr) { // queuing needed?
    *(this->stopPtr++) = value; // just append the byte, tx params are given by
                                // ongoing transmission
                                // TODO: check end of buffer
  }
  else {
    this->sendBuffer[0] = value;
    this->stopPtr = this->sendBuffer + 1;
    this->sendPtr = this->sendBuffer;
    this->blockingSend = blocking;
    if (this->regOffset == (&UCA1CTL0 - &UCA0CTL0)) // see UART_sendString
      INST1_TXIFG = 1;
    else if (this->regOffset == 0)
      INST0_TXIFG = 1;
    if ((this->blockingSend) && (this->sendPtr < this->stopPtr))
      __low_power_mode_1(); // see UART_sendString
  }
}
// section -126--64--91-84--6edc429e:11860caa7eb:-8000:00000000000007EC begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element
void UART_sendString(UART *this, char *stringPtr, boolean blocking)
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--6edc429e:11860caa7eb:-8000:00000000000007F1 begin
{
  // 1.1mA rms with interrupts, 2.8mA rms without
  uint i = 0; // copying str to sendBuffer
  while (stringPtr[i] != '\0') {
    this->sendBuffer[i] = stringPtr[i]; // TODO: queuing? (stopPtr)
    i++;
  }
  this->stopPtr = this->sendBuffer + i; // pointers initialization
  this->sendPtr = this->sendBuffer;
  this->blockingSend = blocking;
  if (this->regOffset == (&UCA1CTL0 - &UCA0CTL0)) // executing the interrupt
    INST1_TXIFG = 1;                              // manually the first time
  else if (this->regOffset == 0)
    INST0_TXIFG = 1;
  if (this->blockingSend)
    __low_power_mode_1(); // shut down leaving uart running (SMCLK active)
  // when blocking, execution will continue here after __lpm_off_on_exit
}
// section -126--64--91-84--6edc429e:11860caa7eb:-8000:00000000000007F1 begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

void UART_sendMem(UART *this, char *startPtr, uint length, boolean blocking)
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2db04367:1186108a0b9:-8000:00000000000007FF begin
{
  uint i = 0; // copying str to sendBuffer
  while (i < length) {
```

```
    this->sendBuffer[i] = startPtr[i]; // TODO: queuing? (stopPtr)
    i++;
  }
  this->stopPtr = this->sendBuffer + length; // pointers initialization
  this->sendPtr = this->sendBuffer;
  this->blockingSend = blocking;
  if (this->regOffset == (&UCA1CTL0 - &UCA0CTL0)) // see UART_sendString
    INST1_TXIFG = 1;
  else if (this->regOffset == 0)
    INST0_TXIFG = 1;
  if (this->blockingSend)
    __low_power_mode_1(); // see UART_sendString
}
// section -126--64--91-84--2db04367:1186108a0b9:-8000:00000000000007FF begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

char UART_recvByte(UART *this, uint timeout)
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2db04367:1186108a0b9:-8000:000000000000081A begin
{
  if (this->readPtr >= (this->recvBuffer + sizeof(this->recvBuffer)))
    this->readPtr = this->recvBuffer;
  uint lto = timeout; // TODO: low power timeout
  while ((lto > 0) && (this->readPtr == this->recvPtr))
    lto--;
  if (this->readPtr != this->recvPtr) {  // TODO: safe to check for lto > 0?
    this->recvTimeout = FALSE;
    return *(this->readPtr++);
  }
  else {
    this->recvTimeout = TRUE;
    return 0;
  }
}
// section -126--64--91-84--2db04367:1186108a0b9:-8000:000000000000081A begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

boolean UART_recvString(UART *this, char *buffPtr, uint buffLength, uint timeout)
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2db04367:1186108a0b9:-8000:0000000000000814 begin
{
  return FALSE;
}
// section -126--64--91-84--2db04367:1186108a0b9:-8000:0000000000000814 begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

boolean UART_recvMem(UART *this, char *buffPtr, uint buffLength, uint timeout)
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2db04367:1186108a0b9:-8000:000000000000080D begin
{
  return FALSE;
}
// section -126--64--91-84--2db04367:1186108a0b9:-8000:000000000000080D begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

void UART_initUART(UART *this, int bitrate, boolean irdaMode)
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:0000000000000816 begin
```

```
{
  RELOC(UCA0CTL1) |= UCSWRST; // reset UART
  this->sendPtr = this->sendBuffer; // send buffer
  this->stopPtr = this->sendPtr;
  this->sendOverflow = FALSE;
  this->recvPtr = this->recvBuffer; // receive buffer
  this->readPtr = this->recvPtr;
  this->recvOverflow = FALSE;
  this->blockingSend = FALSE; // flags
  this->recvTimeout = FALSE;
  this->callback = 0; // TODO: better init (with initial wait!)
  unsigned long n = FBRCLK/bitrate;
  unsigned long mod = (FBRCLK*((long)8))/bitrate-n*8;

  RELOC(UCA0CTL0) = 0; // LSB first
  RELOC(UCA0CTL1) = UCSSEL1 + UCSWRST; // SMCLK
  RELOC(UCA0BR1) = 0;
//  RELOC(UCA0BR0) = 34; // 115200 bps @ 4MHz
//  RELOC(UCA0BR0) = 15;
  ////////////////RELOC(UCA0BR0) = 7;
  RELOC(UCA0BR0) = n;
//  RELOC(UCA0MCTL) = (6 << 1); // UCBRS = 6
//  RELOC(UCA0MCTL) = (5 << 1); // UCBRS = 5
  RELOC(UCA0MCTL) = (mod << 1); // UCBRS
  /////RELOC(UCA0STAT) = UCLISTEN; // clear flags and loopback
  RELOC(UCA0STAT) = 0;
//  UCA1MCTL = (6 << 1); // TODO: REMOVE!!!
//  UCA1BR0 = 16; // TODO: REMOVE!!!
  if (irdaMode == TRUE) {
//    RELOC(UCA0IRTCTL) = (35 << 2) + UCIREN; // UCIRTXPLx = 35 = 4.5us*2*4MHz-1
//    RELOC(UCA0IRRCTL) = (24 << 2) + UCIRRXFE + UCIRRXPL; // UCIRRXFLx = 24 =
(4.5us-1us)*2*4MHz-4
//    RELOC(UCA0IRTCTL) = (17 << 2) + UCIREN; // UCIRTXPLx = 35 = 4.5us*2*4MHz-1
    //RELOC(UCA0IRTCTL) = (3 << 2) + UCIREN; // UCIRTXPLx = 35 = 4.5us*2*4MHz-1
    ////RELOC(UCA0IRTCTL) = (2 << 2) + UCIREN;
    RELOC(UCA0IRTCTL) = 9;
//    RELOC(UCA0IRRCTL) = (5 << 2) + UCIRRXFE; // UCIRRXFLx = 24 = (4.5us-
1us)*2*4MHz-4
//                                      // and detect high pulses (UCIRRXPL = 0)
   ///////// RELOC(UCA0IRRCTL) = (1 << 2) + UCIRRXFE; // UCIRRXFLx = 24 = (4.5us-
1us)*2*4MHz-4
                                        // and detect high pulses (UCIRRXPL = 0)
    RELOC(UCA0IRRCTL) = 0;
  }
  else {
    RELOC(UCA0IRTCTL) &= ~UCIREN;
    RELOC(UCA0IRRCTL) &= ~UCIRRXFE;
  }
  if (this->regOffset == (&UCA1CTL0 - &UCA0CTL0)) {
    iInstance[1] = this;
    P3REN &= ~(P3REN_6 + P3REN_7); // resistor off
    P3SEL |= P3SEL_6 + P3SEL_7; // primary function
    P3DIR_bit.P3DIR_6 = 1; // txd
    P3DIR_bit.P3DIR_7 = 0; // rxd
    RELOC(UCA0CTL1) &= ~UCSWRST; // enable UART (TODO: after port init?)
    INST1_TXIFG = 0; // to avoid an isr trigger on global int enable
    UC1IE |= UCA1TXIE + UCA1RXIE; // tx/rx int enable
  }
  else if (this->regOffset == 0) {
    iInstance[0] = this;
    P3REN &= ~(P3REN_4 + P3REN_5); // resistor off
    P3SEL |= P3SEL_4 + P3SEL_5; // primary function
    P3DIR_bit.P3DIR_4 = 1; // txd
    P3DIR_bit.P3DIR_5 = 0; // rxd
    RELOC(UCA0CTL1) &= ~UCSWRST; // enable UART (TODO: after port init?)
    INST0_TXIFG = 0; // to avoid an isr trigger on global int enable
```

```
    IE2 |= UCA0TXIE + UCA0RXIE; // tx/rx int enable
  }
  __enable_interrupt(); // global int enable
}
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:0000000000000816 begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

Interrupt(USCIAB0TX) void UART_isrAB0_tx()
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:0000000000000823 begin
{
  INST0_TXIFG = 0; // TODO: check int source (A/B)
  if (iInstance[0]->sendPtr < iInstance[0]->stopPtr)
    UCA0TXBUF = *(iInstance[0]->sendPtr++);
  else {
    iInstance[0]->stopPtr = iInstance[0]->sendBuffer; // this first, so stopPtr
always < sendPtr
//    sendPtr = stopPtr;
    if (iInstance[0]->blockingSend == TRUE)
      __low_power_mode_off_on_exit(); // TODO: what if sendX called from an isr?
  }
}
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:0000000000000823 begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

Interrupt(USCIAB0RX) void UART_isrAB0_rx()
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:000000000000081F begin
{
  IFG2_bit.UCA0RXIFG = 0; // TODO: check int source (A/B)
  if (iInstance[0]->callback != 0)
    iInstance[0]->callback(UCA0RXBUF);
  else {
    *(iInstance[0]->recvPtr++) = UCA0RXBUF;
    if (iInstance[0]->recvPtr >= (iInstance[0]->recvBuffer + RECVBUFFERSIZE)) {
      iInstance[0]->recvPtr = iInstance[0]->recvBuffer;
      iInstance[0]->recvOverflow = TRUE;
    }
  }
}
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:000000000000081F begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element

Interrupt(USCIAB1TX) void UART_isrAB1_tx()
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:0000000000000823 begin
{
  INST1_TXIFG = 0; // TODO: check int source (A/B)
   if (iInstance[1]->sendPtr < iInstance[1]->stopPtr)
    UCA1TXBUF = *iInstance[1]->sendPtr++;
  else {
    iInstance[1]->stopPtr = iInstance[1]->sendBuffer; // this first, so stopPtr
always < sendPtr
//    sendPtr = stopPtr;
    if (iInstance[1]->blockingSend == TRUE)
      __low_power_mode_off_on_exit(); // TODO: what if sendX called from an isr?
  }
}
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:0000000000000823 begin
// don't delete the previous line as it's needed to preserve source code of this
```

```
autogenerated element

Interrupt(USCIAB1RX) void UART_isrAB1_rx()
// don't delete the following line as it's needed to preserve source code of this
autogenerated element
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:000000000000081F begin
{
  UC1IFG_bit.UCA1RXIFG = 0; // TODO: check int source (A/B)
  if (iInstance[1]->callback != 0)
    iInstance[1]->callback(UCA0RXBUF);
  else {
    *(iInstance[1]->recvPtr++) = UCA1RXBUF;
    if (iInstance[1]->recvPtr >= (iInstance[1]->recvBuffer + RECVBUFFERSIZE)) {
      iInstance[1]->recvPtr = iInstance[1]->recvBuffer;
      iInstance[1]->recvOverflow = TRUE;
    }
  }
}
// section -126--64--91-84--2f17d71b:1187420e48e:-8000:000000000000081F begin
// don't delete the previous line as it's needed to preserve source code of this
autogenerated element
```

Bibliografia

[1] Stefano Speretta, Leonardo M. Reyneri, C. Sansoè, Maurizio Tranchero, Claudio Passerone, Dante Del Corso, "*MODULAR ARCHITECTURE FOR SATELLITES*" in Proceedings of the 58th international Astronautical Congress, Hyderabad, India, September 2007. IAC-07-B4.7.09

[2] Claudio Passerone, Maurizio Tranchero, Stefano Speretta,Leonardo Reyneri, Claudio Sansoè, Dante Del Corso, "*Design Solutions for a University Nano-satellite*" Politecnico di Torino, Dipartimento di Elettronica

[3] D. Del Corso, C. Passerone, L. M. Reyneri, C. Sansoè, M. Borri, S. Speretta, M.Tranchero, "*Architecture of a small low-cost satellite*" in Proceedings of the 10*th*Euromicro Conference on Digital System Design, Lubeck, Germany, August 2007

[4] Danilo Roascio. *Sottosistema di comunicazione tollerante ai guasti per satellite modulare AraMiS*. Master's thesis, Politecnico di Torino, 2008.

[5] Wei Zinan, Development of a fault-tolerant on-board bus for modular satellites, Master's thesis, Politecnico di Torino,2009

[6] PiCPoT. Online:http://www.delen.polito.it/en/content/view/full/353/

[7] Daniel Mannisto, Mark Dawson, *An Overview of Controller Area Network (CAN) Technology*, Machine Bus Corporation,2003

[8] ECSS-E-50-12 Space Engineering *SpaceWire:* SERIAL POINT-TO-POINT LINKS,2000

[9] Stephane Rey. *Introduction to LIN (Local Interconnect Network)*, 2003

[10] Rhombus Industries Inc. Pulse Transformers, Transformer equivalent circuit, 2008. Online: <http://www.rhombus-ind.com/app-note/circuit.pdf>

[11] C&D Technologies-Murata Power Solutions. Pulse Transformers, 786 Series Datasheet, 2001

[12] ON Semiconductor. NTR4501N, Power MOSFET, datasheet, 2008

[13] Infrared Data Association. IrDA Serial Infrared Physical Layer Specification,Version 1.4, 2001

[14] Texas Instruments. Mixed Signal Products, MSP430x2xx Family, User's Guide,2007

[15] Datasheet microcontrollore Texas Instruments MSP430F2418

[16] MSP430 USB-Debug-Interface MSP-FET430UIF

[17] Enciclopedia in rete. *http://it.wikipedia.org/wiki/Pagina_principale*