SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA
E TELECOMUNICAZIONI PER LO SVILUPPO SOSTENIBILE

Connessione di sistemi a microcontrollore con reti geografiche

Elaborato in

Progetto di Reti di Telecomunicazioni LM

Relatore:
Prof. Franco Callegati
Correlatore
Ing. Marco Ramilli

Presentato da:
Simone Colella

Terza Sessione

Anno Accademico 2010/2011

Sommario

Introduzione

Fra le varie ragioni della crescente pervasività di Internet in molteplici settori di mercato del tutto estranei all'ICT, va senza dubbio evidenziata la possibilità di creare canali di comunicazione attraverso i quali poter comandare un sistema e ricevere da esso informazioni di qualsiasi genere, qualunque distanza separi controllato e controllore.

Nel caso specifico, il contesto applicativo è l'automotive: in collaborazione col Dipartimento di Ingegneria Elettrica dell'Università di Bologna, ci si è occupati del problema di rendere disponibile a distanza la grande quantità di dati che i vari sotto-sistemi componenti una automobile elettrica si scambiano fra loro, sia legati al tipo di propulsione, elettrico appunto, come i livelli di carica delle batterie o la temperatura dell'inverter, sia di natura meccanica, come i giri motore. L'obiettivo è quello di permettere all'utente (sia esso il progettista, il tecnico riparatore o semplicemente il proprietario) il monitoraggio e la supervisione dello stato del mezzo da remoto nelle sue varie fasi di vita: dai test eseguiti su prototipo in laboratorio, alla messa in strada, alla manutenzione ordinaria e straordinaria.

L'approccio individuato è stato quello di collezionare e memorizzare in un archivio centralizzato, raggiungibile via Internet, tutti i dati necessari. L'architettura scelta è rappresentata nella figura sottostante:

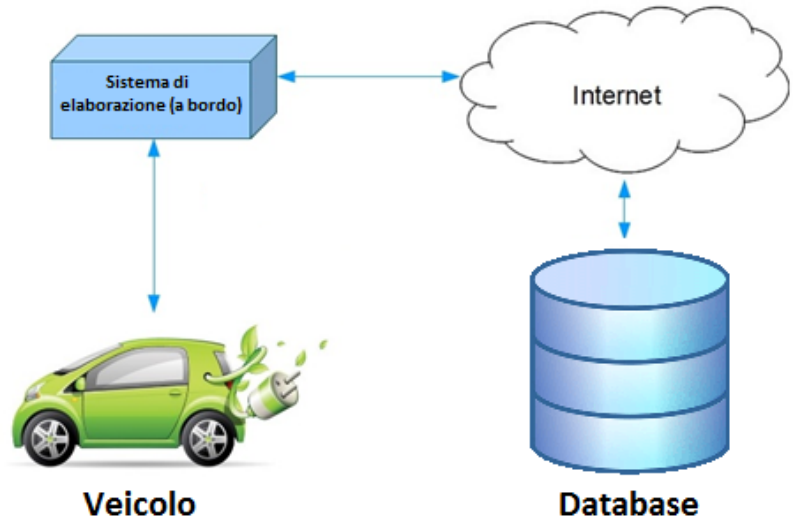


Fig. 1 - Schema generale del progetto

Il sistema di elaborazione a bordo richiede di essere facilmente integrabile, quindi di piccole dimensioni, e a basso costo, dovendo prevedere la produzione di molti veicoli; ha inoltre compiti ben definiti e noti a priori. Data la situazione, si è quindi scelto di usare un sistema embedded, cioè un sistema elettronico di elaborazione progettato per svolgere un limitato numero di funzionalità specifiche sottoposte a vincoli temporali e/o economici. Apparati di questo tipo sono denominati "special purpose", in opposizione ai sistemi di utilità generica detti "general purpose" quali, ad esempio, i personal computer, proprio per la loro capacità di eseguire ripetutamente un'azione a costo contenuto, tramite un giusto compromesso fra hardware dedicato e software, chiamato in questo caso "firmware".

I sistemi embedded hanno subito nel corso del tempo una profonda evoluzione tecnologica, che li ha portati da semplici microcontrollori in grado di svolgere limitate operazioni di calcolo a strutture complesse in grado di interfacciarsi a un gran numero di sensori e attuatori esterni oltre che a molte tecnologie di comunicazione.

Nel caso in esame, si è scelto di affidarsi alla piattaforma open - source Arduino; essa è composta da un circuito stampato che integra un microcontrollore Atmel da programmare attraverso interfaccia seriale, chiamata *Arduino board*, ed offre nativamente numerose funzionalità, quali ingressi e uscite digitali e analogici, supporto per SPI, I2C ed altro; inoltre, per aumentare le possibilità d'utilizzo, può essere posta in comunicazione con schede elettroniche esterne, dette *shield,* progettate per le più disparate applicazioni, quali controllo di motori elettrici, gps, interfacciamento con bus di campo quale ad esempio CAN, tecnologie di rete come Ethernet, Bluetooth, ZigBee, etc. L'hardware è open - source, ovvero gli schemi elettrici sono liberamente disponibili e utilizzabili così come gran parte del software e della documentazione; questo ha permesso una grande diffusione di questo frame work, portando a numerosi vantaggi: abbassamento del costo, ambienti di sviluppo multi-piattaforma, notevole quantità di documentazione e, soprattutto, continua evoluzione ed aggiornamento hardware e software.

È stato quindi possibile interfacciarsi alla centralina del veicolo prelevando i messaggi necessari dal bus CAN e collezionare tutti i valori che dovevano essere archiviati. Data la notevole mole di dati da elaborare, si è scelto di dividere il sistema in due parti separate: un primo nodo, denominato Master, è incaricato di prelevare dall'autovettura i parametri, di associarvi i dati GPS (velocità, tempo e posizione) prelevati al momento della lettura e di inviare il tutto a un secondo nodo, denominato Slave, che si occupa di creare un canale di comunicazione attraverso la rete Internet per raggiungere il database. La denominazione scelta di Master e Slave riflette la scelta fatta per il protocollo di comunicazione fra i due nodi Arduino, ovvero l'I2C, che consente la

comunicazione seriale fra dispositivi attraverso la designazione di un “master” e di un arbitrario numero di “slave”.

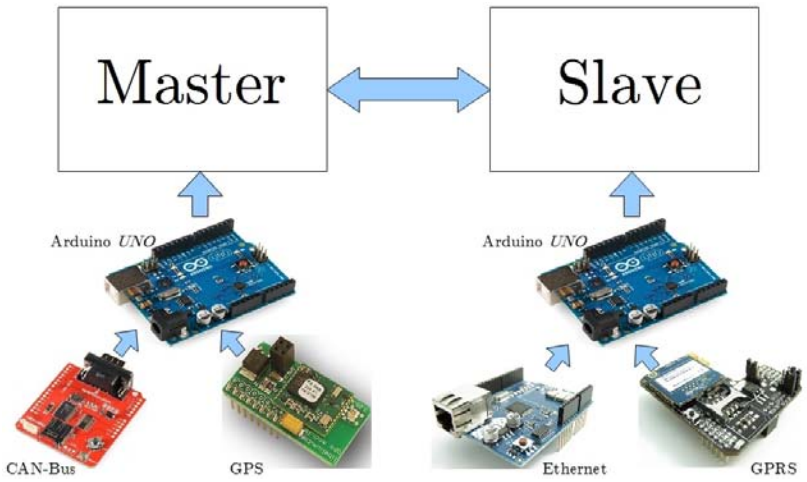


Fig. 2 - Architettura del sistema di controllo

La suddivisione dei compiti fra due nodi permette di distribuire il carico di lavoro con evidenti vantaggi in termini di affidabilità e prestazioni.

Del progetto si sono occupate due Tesi di Laurea Magistrale; la presente si occupa del dispositivo Slave e del database.

Avendo l’obiettivo di accedere al database da ovunque, si è scelto di appoggiarsi alla rete Internet, alla quale si ha oggi facile accesso da gran parte del mondo. Questo ha fatto sì che la scelta della tecnologia da usare per il database ricadesse su un web server che da un lato raccoglie i dati provenienti dall’autovettura e dall’altro ne permette un’agevole consultazione. Anch’esso è stato implementato con software open - source: si tratta, infatti, di una web application in linguaggio php che riceve, sotto forma di richieste HTTP di tipo GET oppure POST, i dati dal dispositivo Slave e provvede a salvarli, opportunamente formattati, in un database MySQL.

Questo impone però che, per dialogare con il web server, il nodo Slave debba implementare tutti i livelli dello stack protocollare di Internet. Due differenti *shield* realizzano quindi il livello di collegamento, disponibile sia via cavo sia wireless, rispettivamente attraverso l’implementazione in un caso del protocollo Ethernet, nell’altro della connessione GPRS. A questo si appoggiano i protocolli TCP/IP che provvedono a trasportare al database i dati ricevuti dal dispositivo Master sotto forma di messaggi HTTP.

Nei capitoli a seguire saranno descritti approfonditamente il sistema veicolare da controllare e il sistema controllore; si passerà quindi alla realizzazione dei firmware utilizzati per realizzare le funzioni dello Slave con tecnologia Ethernet e con tecnologia GPRS; sarà descritta la web application e il database; infine, saranno presentati i risultati delle simulazioni e dei test svolti sul campo nel laboratorio DIE.

1 - Arduino e il Sistema Veicolare

1.1 - La Piattaforma Arduino

1.1.1 - Il progetto Arduino

Arduino è una piattaforma open source concepita per la prototipazione rapida e flessibile di progetti elettronici. Essa si compone di una parte hardware e di una software.

Elemento centrale dell'hardware è la *Arduino board,* ovvero un circuito stampato integrante un microcontrollore Atmel AVR della serie megaAVR. A seconda della versione del microcontrollore, e quindi delle sue prestazioni, si distinguono varie versioni della *Arduino Board*, quali la UNO, Dumilanove, Mega ecc. Oltre al microcontrollore, è incluso un regolatore di tensione a 5 volt, un oscillatore a cristallo a 16MHz e un'interfaccia usb. Sono messi a disposizione dei pin di input/output digitale e dei pin analogici in grado di campionare la tensione rilevata fino a un massimo di 5V e di convertirla in 1024 livelli discreti; è inoltre possibile generare segnali PWM. Sono poi supportati protocolli di comunicazione quali SPI e I2C.

In questo progetto si è scelto di usare la *Arduino UNO* [1], basata su Atmel AtMega328P microcontrollore a 8 bit, con architettura RISC e pin-out a 28 pin. Dispone di 32KB di memoria flash (di cui 0.5KB utilizzati dal bootloader), 2 KB di SRAM e 1 KB di EEPROM. È alimentabile esternamente (range da 7V a 12V) oppure via usb (la selezione è effettuata automaticamente); è presente un dispositivo di protezione da sovracorrente (fusibile ripristinabile) che interviene quando la corrente sulla usb è maggiore di 500 mA. Offre un'interfaccia usb che, grazie al chip ATmega8U2 "USB-to-TTL Serial Chip", permette di comunicare direttamente con il microcontrollore (dotato invece di interfaccia UART). Sono presenti 14 pin di I/O digitale, alcuni dei quali hanno funzionalità aggiuntive, usabili nativamente o con opportune librerie:

- Pin 0 (RX) e 1 (TX) usati per la comunicazione seriale e connessi direttamente al chip ATmega8U2
- Pin 2 e 3 programmabili come interrupt esterni
- Pin 3, 5, 6, 9, 10 e 11 possono generare segnali PWM

- Pin 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCLK) supportano la comunicazione col protocollo SPI
- Pin 13 collegato a un led integrato nella *board*

Sono presenti 6 pin di input analogico, dei quali i pin 4 e 5 implementano la comunicazione I2C.

Un altro aspetto interessante del progetto Arduino è la possibilità di espandere le potenzialità applicative della *board* attraverso i cosiddetti *shields*. Si tratta di schede elettroniche che integrano i più disparati dispositivi, dal controller per Ethernet a quello per il bus CAN, dal controller ZigBee al GPRS, che possono essere utilizzati semplicemente "impilando" tali *shield* sopra la *board*, essendo previsto uno standard per la disposizione dei pin sulle schede. L'utilizzo di tali espansioni è facilitato dalla presenza di librerie software che facilitano la loro integrazione nei programmi, fornendo numerose funzioni pronte per l'utilizzo.

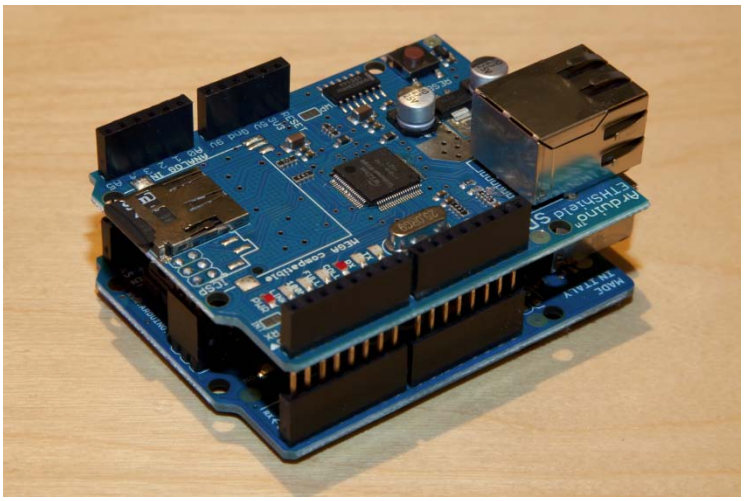


Fig.1 - Arduino UNO ed Ethernet Shield

In questo progetto si sono utilizzati vari *shield*: CAN e GPS per il nodo master, Ethernet e GPRS per il nodo slave; si veda oltre per una più approfondita descrizione degli *shield* pertinenti a questa Tesi.

L'aspetto open source del progetto è dato dal fatto che tutti gli schemi elettrici e l'elenco dei componenti necessari di tutte le *board* e gli *shield* sono disponibili liberamente con licenza Creative Commons Attribution-ShareAlike 2.5, permettendo [2] a chiunque lo desideri, per esempio, di costruire legalmente una propria versione modificata di una delle *board* o degli *shield*.

La parte software è composta da un *boot loader* e da una IDE. Il *boot loader* è precaricato nel microcontrollore ed ha il compito di semplificare il caricamento del codice, evitando di dover ricorrere a un programmatore hardware dedicato e consentendo semplicemente la programmazione via seriale. L'IDE è scritta in linguaggio Java e risulta quindi nativamente multipiattaforma. Tale ambiente è basato sul progetto open source *Processing* e permette di programmare il microcontrollore con un linguaggio *C-like,* anche questo derivato da un secondo progetto open source chiamato *Wiring,* ed offre varie funzionalità utili in fase di stesura del codice, quali evidenziamento della sintassi, indentazione, controllo parentesi e altro. Uno dei pregi di maggior rilievo dell'ambiente di sviluppo è tuttavia la possibilità di scrivere una sola volta il codice e di poterlo poi caricare su tutte le *board* del progetto Arduino, dato che la IDE si preoccupa di generare l'opportuno codice macchina a seconda del diverso microcontrollore usato. Questo permette di progettare il software indipendentemente dal tipo di hardware e dalle sue prestazioni, garantendo vantaggi sia in fase di progettazione, dato che permette un maggior livello di astrazione, sia in fase di test e validazione, dato che risulta semplice sostituire una *board* con un'altra di diverse caratteristiche in termini di capacità computazionale, memoria, costo, consumi o altro ancora a seconda delle necessità.

La struttura base di un programma per Arduino, comunemente chiamato *sketch,* prevede solo due funzioni: un *setup()* incaricato di svolgere una configurazione preliminare del sistema, eseguito all'avvio, ed un *loop()* che viene eseguito ciclicamente fino allo spegnimento. Possono essere inoltre caricate delle librerie per gestire gli *shield* utilizzando funzioni di alto livello.

Anche gran parte del software, incluse le librerie necessarie al funzionamento degli *shield*, viene rilasciato sotto licenza Creative Commons.

La scelta di affidarsi ad Arduino è stata influenzata da due fattori principali: da un lato l'aspetto open source, che garantisce una grande abbondanza di schede di espansione, librerie e documentazione, oltre ad una continua evoluzione, spinta sia dal team di sviluppo ufficiale di Arduino sia dalla vasta community; dall'altro, il basso costo, molto minore di quello dell'intero sistema da controllare, che da quindi la possibilità di aggiungere funzionalità avanzate a bassissimi costi.

1.1.2 - Shield Ethernet

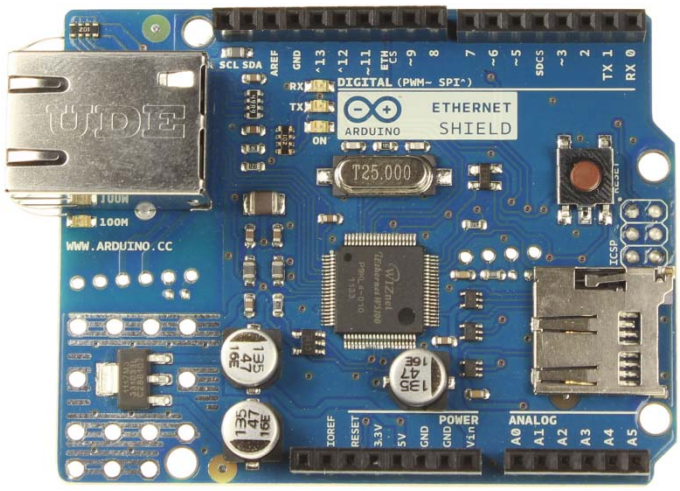


Fig. 2 - Ethernet Shield

L'*Ethernet Shield* integra il controller Ethernet Wiznet W5100 [3]; esso ha un buffer interno di 16KB e mette a disposizione gli stack TCP/IP e UDP/IP, supportando fino a quattro *socket* contemporanee. La connessione alla rete LAN, disponibile in entrambe le velocità 10 e 100 Mb/s, avviene tramite comune cavo di rete Ethernet con connettore RJ-45. Sono presenti numerosi led riguardanti lo stato della scheda e della comunicazione:

- RX,TX,COLLISION: indicano rispettivamente ricezione, trasmissione o collisione di frame
- PWR: indica che lo *shield* è alimentato
- LINK: indica la presenza di un collegamento alla rete; lampeggia in ricezione e trasmissione
- FULLD: indica che la connessione alla rete è full-duplex
- 100M: indica che la connessione è a 100Mb/s

È anche presente un lettore di schede micro-SD.

Arduino comunica con il controller e con il lettore di micro-SD attraverso il bus SPI, il quale è un bus seriale, sincrono (dato che viene distribuito a tutti gli slave il clock del master) e full-duplex. Prevede la presenza di un microcontrollore master (Arduino appunto) che controlla un bus a quattro fili, comunicando con altri circuiti integrati aventi ruolo di slave. Sulla *board* UNO è implementato sui seguenti pin digitali:

- 11: MOSI, master output slave input

- 12: MISO, master input slave output
- 13: SCLK, serial clock
- 10: chip select del W5100
- 4: chip select del card reader.

Questi pin non possono essere quindi usati per altri scopi.

Congiuntamente all'hardware viene fornita una libreria, detta *Ethernet Library*. La struttura della libreria prevede [4] cinque classi, in modo da permettere il funzionamento come client TCP, server TCP e la comunicazione UDP.

Per essere usata negli *sketch* deve essere inclusa con il comando:

```
#include <Ethernet.h>
```

Quindi nel *setup()* deve essere inizializzata la scheda attraverso il comando:

```
Ethernet.begin(mac,ip,gateway,subnet);
```

Si noti che l'ip, il gateway e la subnet possono essere omessi se nella LAN è presente un server DHCP. Quindi, è possibile creare un oggetto di tipo Client o Server cui far effettuare varie operazioni, quali l'apertura di una socket (metodo connect()), l'invio di byte o la ricezione di byte (rispettivamente con i metodi write() e read()) . Oppure si può creare un oggetto di tipo EthernetUDP ed inviare e ricevere dati con il protocollo UDP attraverso metodi analoghi.

È disponibile on line un'ampia documentazione al riguardo, ricca di esempi commentati, oltre ad un forum dove la community aiuta i neofiti, propone migliorie al team di sviluppo, discute di progetti ecc.

1.1.3 - Shield GPRS

Fig. 3 - GPRS Shield

Questo *shield* è composto da un modulo GPRS prodotto dalla Sagem Communications, chiamato HiLo, da un regolatore di tensione a 3.3V e da un alloggiamento per la scheda SIM.

Il modulo HiLo è stato progettato specificamente per applicazioni M2M (machine to machine) ovvero per favorire la comunicazione a distanza fra entità elettroniche, sfruttando la rete GSM/GPRS. Ha piena funzionalità GSM [5] (chiamate, SMS, rubrica, ecc) e GPRS (Classe 10, con velocità in down-link fino 85,6 Kbps e fino a 42,8 Kbps in up-link); è un dispositivo *quad band*, ovvero in grado di operare sia sulle bande di frequenza 900/1800 MHz (in uso principalmente in Europa) e sulle 850/1900 MHz (in uso nel continente americano); garantisce inoltre un corretto funzionamento in un ampissimo range di temperatura, da -40°C a +85°C, aspetto importante dato che se ne prevede il montaggio su veicolo, dove le temperature possono rapidamente salire e scendere a seconda delle condizioni ambientali.

Può essere alimentato esternamente oppure direttamente dalla *Arduino Board*; in quest'ultimo caso deve essere tenuto in conto l'assorbimento di corrente:

- In comunicazione: 220mA, 2200mA di picco
- In stand-by: <2mA
- Off: 50µA

Arduino UNO infatti, se alimentato semplicemente via usb, non è in grado di fornire sufficiente corrente al modulo (si rammenti la presenza sulla linea usb di Arduino UNO del fusibile riprogrammabile tarato a 500 mA), il quale non sarà in grado quindi di accendersi; le soluzioni sono due: o scegliere l'alimentazione esterna per il modulo GPRS oppure alimentare esternamente la *board* Arduino.

Il modulo ha un connettore SMA per il collegamento di un'antenna esterna tramite cavo coassiale di impedenza 50Ω, da utilizzarsi necessariamente pena la mancata connessione alla rete GSM/GPRS.

L'accensione del modulo richiede [6] di fornire per almeno 629ms (a 25°C) un segnale logico alto al pin denominato POK_IN del modulo; ciò può essere fatto manualmente, premendo l'apposito pulsante previsto sullo *Shield* oppure via software, imponendo ad Arduino di portare a livello logico alto il piedino digitale numero 2, col comando:

```
digitalWrite(2,HIGH);
```

Il funzionamento del modulo HiLo è gestito attraverso comandi AT. Tali comandi, nella loro forma base, consistono [7] in stringhe da inviare al dispositivo via seriale; sono formate dai caratteri "AT" seguiti da un opportuno set di caratteri, rappresentanti una determinata operazione da far svolgere al dispositivo, e terminate da un carattere di ritorno a capo (<CR>, carriage return, ascii 13). Ad esempio, per fare una chiamata si dovrà inviare la stringa:

```
ATD <number>;
```

Sono inoltre supportati i comandi estesi, ovvero set di comandi che prevedono la manipolazione di parametri. Possono essere di quattro tipologie:

Tipo di comando	Formato	Descrizione
Comando di Test	AT+<XXX>=?	Il dispositivo ritorna la lista dei parametri utilizzabili e i loro possibili valori.
Comando di Lettura	AT+<XXX>?	Il dispositivo ritorna i valori attuali dei parametri.
Comando di Scrittura	AT+<XXX>=<…>	Il comando imposta i parametri con i valori forniti.
Comando di Esecuzione	AT+<XXX>	Il dispositivo ritorna la lista dei parametri non variabili, modificabili solo dai processi interni del dispositivo stesso.

Di seguito sono riportati esempi di comandi estesi, uno per tipo:

Comando di Test	AT+CMGF=?	Ritorna i possibili formati dei messaggi SMS (dati o testo).
Comando di Lettura	AT+CMGF?	Ritorna la attuale modalità dei messaggi sms (dati o testo).
Comando di Scrittura	AT+CMGF=1	Imposta la modalità dei messaggi sms in testo.
Comando di Esecuzione	AT+GSN	Ritorna l'IMEI.

Quasi sempre è prevista una risposta, [7] inviata dal modulo via seriale, avente forma:

<CR><LF><response><CR><LF>

Dove <LF> è il carattere di nuova linea, line feed, ASCII 12; risposte tipiche sono OK o ERROR. Esistono anche risposte estese, riportanti il codice di errore CME (errore del dispositivo) e CMS (errore di rete avente forma:

+CME ERROR: <n>

Dove il numero <n> indica il tipo di errore occorso; per esempio, CME 10 indica sim non inserita; CMS 42 indica che la rete al momento è in uno stato di congestione ecc.

I set di comandi supportati dal modulo [7] sono quelli standard delle versioni 07.05, 07.07 e v25ter. Inoltre, il modulo mette a disposizione un set di comandi AT proprietari per semplificare lo scambio di dati attraverso l'utilizzo dei protocolli più diffusi di Internet. In particolare, è implementato uno stack protocollare molto ricco, composto da: TCP/IP (comandi AT+KCNX<XXX> e AT+KTCP<XXX>), UDP (comandi AT+KUDP<XXX>), FTP (comandi AT+KFTP<XXX>), SMTP (comandi AT+KSMTP<XXX>) e POP3 (comandi AT+KPOP<XXX>)

La comunicazione col modulo avviene attraverso un'interfaccia UART con controllo di flusso, con velocità fino a 115.2Kbps, la quale è messa in comunicazione con i piedini 0 e 1 digitali di Arduino, dedicati alla comunicazione seriale.

1.2 - Il Sistema Veicolare

1.2.1 - Descrizione

Il sistema veicolare è stato progettato dal DIE, Dipartimento di Ingegneria Elettrica dell'Università di Bologna: si tratta di un prototipo di veicolo completamente elettrico, alimentato da un corposo "pacco batterie" che alimentano un motore elettrico a tre fasi. Nel laboratorio del Dipartimento è disponibile tutta la parte *powertrain* del veicolo, ovvero tutti i componenti dedicati alla generazione e al controllo del moto; in particolare, è stato approntato un set-up sperimentale composto dai componenti mostrati nelle figure sottostanti.

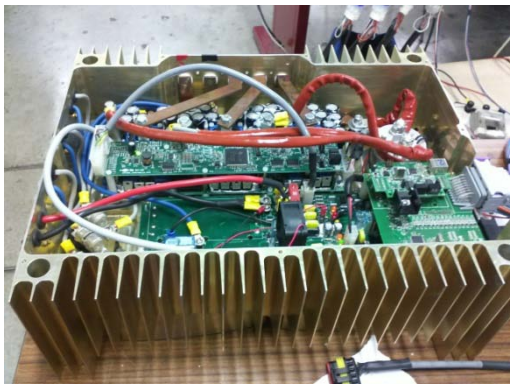


Fig. 4 - Centralina di Controllo

Fig. 5 -Motore elettrico (in primo piano) e batterie (in secondo piano)

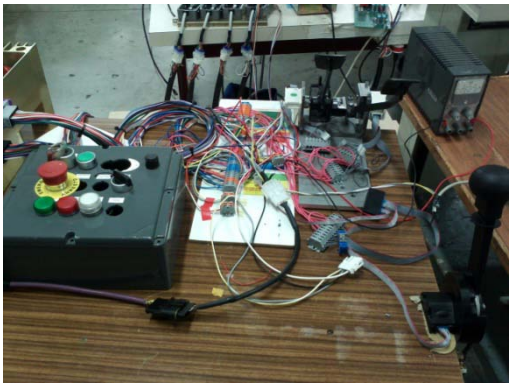


Fig. 6 - Da sinistra, cruscotto, pedaliera e leva del cambio

La centralina comunica con i dispositivi di controllo delle varie parti del sistema (sensori di temperatura, misuratori di carica ecc.) attraverso due reti realizzate con un bus di campo CAN, una non accessibile dall'esterno e dedicata alle comunicazioni *safety critical* mentre l'altra da utilizzare per applicazioni di monitoraggio. Entrambe le reti hanno una *data rate* pari a 250kbit/s.

1.2.2 - Protocollo CAN

Il CAN è un bus seriale multicast, progettato negli anni ottanta dalla Robert Bosch GmbH e standardizzato dalla International Organization for Standardization come ISO 11989 e dalla Society of Automotive Engeneers come SAE J1939. È strutturato [8] su quattro livelli gerarchici, in analogia con il modello ISO/OSI:

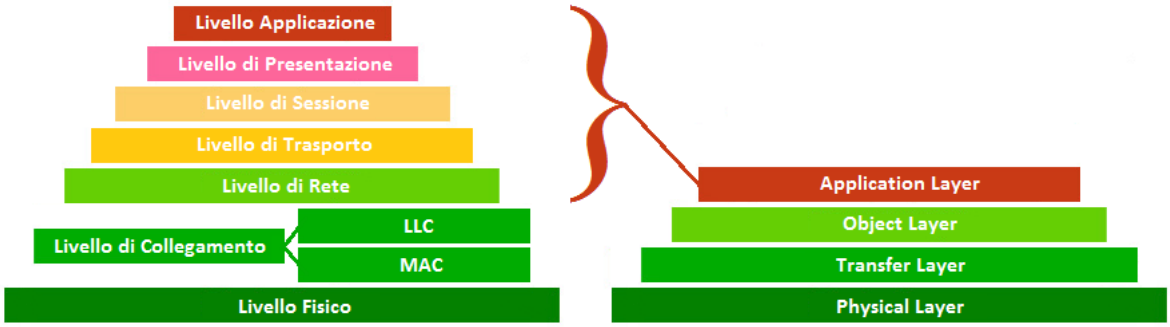


Fig. 7 - Confronto fra pila ISO/OSI e protocollo CAN

Il livello fisico specifica molti dettagli, dato che deve essere garantita un'elevata immunità ai disturbi elettromagnetici. Per esempio, come mezzo di trasmissione si usa un singolo canale bidirezionale realizzato con doppino elettrico intrecciato, schermato o

meno a seconda della quantità di rumore presente nell'ambiente. Al fine di aumentare l'immunità alle interferenze esterne poi, viene usata per i segnali una codifica differenziale. Viene data poi particolare rilevanza alle politiche di gestione del mezzo fisico, in quanto responsabili del valore medio e della variabilità della latenza dei messaggi; sono parametri critici in applicazioni *automotive*, dove la mancata ricezione di un messaggio d'errore entro una precisa *dead line* temporale può essere catastrofica. Invece, i compiti che nel modello OSI sono spettano ai livelli superiori, sono demandati nel CAN al livello applicativo, sul quale, peraltro, nessun standard pone alcuna specifica, dato che si lascia libero il progettista di valutare il rapporto fra costo (in termini di *overhead* introdotto) e beneficio di ogni funzionalità di alto livello che si vuol introdurre.

Il protocollo CAN è di tipo CSMA/CR: quando un nodo connesso al bus intende trasmettere, deve controllare che il canale sia libero, ovvero che nessun altro sia in trasmissione. Qualora si verificasse una collisione, si ha un meccanismo di arbitraggio basato su bit dominanti (0 logico) e recessivi (1 logico): in caso di collisione fra due bit, il canale assumerà valore logico secondo il seguente schema, detto *wired AND*:

	Dominante	Recessivo
Dominante	Dominante	Dominante
Recessivo	Dominante	Recessivo

Quando un nodo impone al canale uno stato recessivo (trasmissione di un 1) ma, nonostante ciò, rileva che lo stato del canale è dominante, capisce che un messaggio più prioritario sta impegnando il canale e sospende la trasmissione del proprio.

In particolare, il canale a riposo risulta essere in stato recessivo; per iniziare la trasmissione, si genera un primo bit dominante, detto di *Start of Frame*, seguito da un campo a 11 bit detto *data type* avente duplice funzione: identificare il tipo di dato trasmesso e fissarne la priorità. All'atto della collisione fra due o più messaggi, il confronto bit a bit del *data type* decreterà quale sarà inviato subito e quale ritardato.

Sono previsti cinque diverse tipologie di frame:

- Data frame: usato per la trasmissione dati fra nodi
- Remote frame: usato per richiedere dati ad un altro nodo
- Error frame: usato per comunicare la presenza di un errore

- Interframe: sequenza di bit recessivi usata per riportare il canale nello stato di riposo, ovvero recessivo
- Overframe: simile all'error frame, ma generabile solo durante l'interframe

1.2.3 - Messaggi CAN

Come già menzionato, il veicolo ha due reti CAN, una privata e una pubblica; all'interno della rete accessibile dall'esterno è presente un nodo, chiamato *Device Info*, il quale provvede ad effettuare un'operazione di raccolta e di sintesi di un certo set di messaggi circolanti sulla prima rete privata, ritenuti rilevanti ai fini del monitoraggio, e a generare messaggi con opportuna temporizzazione su quella pubblica. La struttura del frame è la seguente:

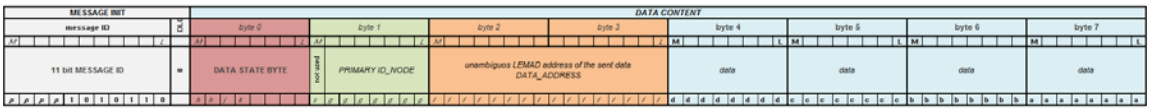


Fig. 8 - Struttura del frame inviato dal nodo Device Info

Il sistema veicolare da trattare utilizza CAN_lemad, un'implementazione del livello applicativo del CAN standard, la cui documentazione è riportata in Appendice B. Questa implementazione utilizza solo data frame e sceglie di dividere gli 11 bit di identificazione in due sottocampi: i primi quattro identificano il tipo di messaggio (e quindi la priorità) mentre i secondi sette identificano il nodo trasmettente, che in questo caso ha valore fisso 1010110 in ogni messaggio, dato che è sempre il nodo *Device Info* a trasmettere.

Il campo dati, composto da otto byte, è poi suddiviso come segue:

- Primo byte: data state, riportante informazioni sullo stato dei dati (per esempio se sono stati generati durante la fase di inizializzazione della periferica, e quindi non validi).
- Secondo byte: primary id node, identifica il nodo che sta trasmettendo il messaggio sulla rete CAN privata.
- Terzo e quarto byte: data address, identificano univocamente il tipo di dato (per esempio "temperatura dell'inverter").
- Successivi byte: dati veri e propri; possono essere inviati dati di tipo int o unsigned int (2 byte) oppure di tipo long (4 byte); i bit non utilizzati sono posti a zero.

I messaggi di interesse sono i seguenti:

Messaggio	Data Address (decimale)	Dimensione Dati (byte)	Tipo Dati	Temporizzazione (ms)
Motor Current	10180	2	Unsigned Int	10
Battery Current	10301	2	Signed Int	10
Motor Winding Temperature	13101	2	Signed Int	1000
DC Bus Voltage	33631	2	Unsigned Int	10
Charge Level	32791	2	Unsigned Int	100
Charge Current	34901	2	Unsigned Int	1000
Number of Equalizing Cells	34231	2	Unsigned Int	1000
Motor Speed	11600	4	Signed Long	10
Inverter Temperature	13051	2	Signed Int	100
Auxiliary Battery Voltage	10031	2	Unsigned Int	100
Maximum Battery Temperature	13151	2	Unsigned Int	100
Lower Cell Voltage	33621	2	Unsigned Int	10
Higher Cell Voltage	33611	2	Unsigned Int	10

1.3 - Il Sistema Controllore

1.3.1 - Scopo

Lo scopo del presente lavoro è stato quello di realizzare un sistema elettronico che prelevasse i messaggi dal bus CAN, vi associasse, attraverso il GPS, le coordinate geografiche e temporali relative al momento della lettura, e salvasse il tutto in un database remoto attraverso tecnologie di rete estremamente diffuse quali Ethernet e GPRS. I risvolti applicativi sono molteplici: per esempio, in fase di prototipazione del veicolo possono essere monitorati a distanza alcuni parametri per verificarne la correttezza, oltre che creare uno storico di dati su cui fare statistiche; durante l'utilizzo da parte dell'acquirente finale poi, il costruttore potrebbe effettuare periodiche valutazioni dello stato di salute del veicolo nonché rapide diagnosi dei guasti, il tutto a distanza. Queste sono solo alcune applicazioni, evidentemente limitate solo dalle esigenze e dall'inventiva dei fruitori del sistema di controllo.

I passi seguiti nel progetto del sistema di controllo sono i seguenti:

- Individuazione dell'architettura di sistema migliore
- Progetto dei componenti
- Prove con simulatore
- Prove sul veicolo

Nel seguito si questo capitolo sarà presentata l'architettura scelta; nei capitoli a seguire, invece, saranno presentati i componenti pertinenti a questa Tesi (nodo Slave, database e web application) ed infine le prove svolte, dapprima attraverso simulazione quindi direttamente nel laboratorio DIE.

1.3.2 - Architettura

Si è scelta un'architettura distribuita, ovvero costituita da due nodi, denominati Master e Slave comunicanti fra loro attraverso protocollo I2C. La motivazione che sta alla base della scelta di dividere il progetto in due unità logiche distinte è duplice: da un lato, la bassa capacità computazionale delle *board* Arduino, unitamente alla grande mole di dati provenienti dal bus CAN, rende impossibile per un singolo nodo la raccolta dati e le

successive operazioni di salvataggio su database; dall'altro, il basso costo economico dell'hardware permette di realizzare funzionalità logiche diverse su dispositivi fisici diversi, garantendo riuso, scalabilità, facilità di identificazione e confinamento dei guasti, oltre che a una più facile progettazione.

Il primo aspetto considerato è stata la grande differenza fra il periodo dei messaggi CAN e il tempo medio necessario all'instaurazione di una connessione TCP fra la *board* Arduino e il server remoto; ciò comporta l'impossibilità di realizzare un nodo Slave che riceva il singolo messaggio CAN, lo elabori e lo salvi nel database, dato che durante il tempo necessario a svolgere queste operazioni sopraggiungerebbe un gran numero di altri messaggi che verrebbero inevitabilmente scartati. La soluzione trovata è la seguente: il nodo Master raccoglie per ogni tipo di dato 10 valori e ne calcola la media matematica; accumula quindi 20 medie e provvede ad inviare allo Slave un vettore contenente queste 20 medie con annesse le coordinate GPS relative.

La comunicazione I2C è seriale, quindi il nodo Slave riceve un byte alla volta; si è quindi definito un protocollo di comunicazione per fissare un formato comune per il frame che viene generato dal Master. Sono stati delineati e quindi provati vari formati, eliminando di volta in volta problemi sia logici che di compatibilità di caratteri da piattaforma a piattaforma. La versione finale prevede per le stringhe ricevute due tipi di formato, a seconda del fatto che il dato sia a 4 byte (parametro Motor Speed) o a 2 byte (i restanti).

La stringa ricevuta per i dati riguardanti il parametro Motor Speed è la seguente:

```
!!,DATA_TYPE_MSB,DATA_TYPE_LSB,val1_MSBH,val1_MSBL,val1_LSBH,val1
_LSBL,[...],$$,152532,4411.9724,N,01202.0435,E,10,125.7,0.25,1409
11,**
```

Dato che il Master invia sequenze di caratteri separati da virgole, si sono usati due caratteri identici consecutivi per segnalare l'inizio e la fine del frame, per evitare ambiguità e rendere univoca l'identificazione del formato del frame; quindi, sono riportati, nell'ordine, il byte più significativo e quello meno significativo del data type; in seguito sono riportati a gruppi di quattro byte (dal più significativo al meno significativo) le 20 medie riguardanti i valori della velocità del motore; vi sono quindi due caratteri separatori ed in seguito i dati GPS, ovvero orario, latitudine, longitudine, il numero di satelliti agganciati, la velocità, l'altitudine e la data; infine sono previsti due caratteri terminatori. I simboli $$ devono essere eliminati dallo Slave quando invia la stringa alla web application, dato che questi caratteri con il linguaggio Php possono creare problemi di compatibilità.

I restanti parametri hanno stringhe analoghe ma vengono usati gruppi di due byte anziché quattro per rappresentare le medie:

```
!!,DATA_TYPE_MSB,DATA_TYPE_LSB,val1_MSB,val1_LSB,val2_MSB,val2_LS
B,[...],$$,152532,4411.9724,N,01202.0435,E,10,125.7,0.25,140911,*
*
```

Il ricevitore è stato implementato con una macchina a stati finiti:

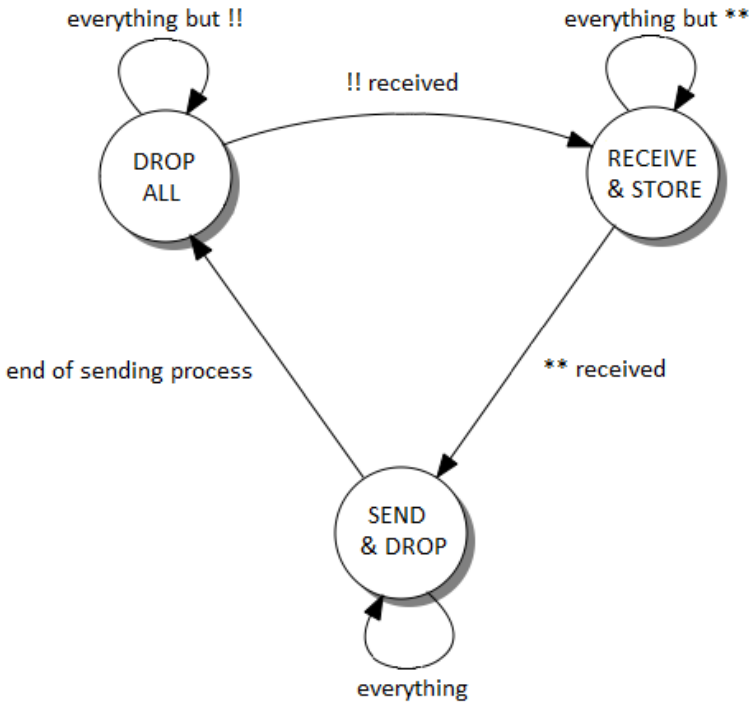


Fig. 9 - Grafo della macchina a stati finiti usata per la ricezione

Il funzionamento è il seguente: il dispositivo scarta tutto ciò che riceve fino alla ricezione dei caratteri di inizio frame; quindi inizia a salvare in un buffer tutti i byte successivi, fino alla ricezione dei caratteri di fine frame; quindi, controllata la correttezza del frame ricevuto, instaura la connessione a Internet (con Ethernet o GPRS) e invia i dati al database remoto; durante questa fase, I dati eventualmente ricevuti vengono necessariamente scartati. L'implementazione sarà accuratamente descritta nel capitolo 3.

Anche il database è stato implementato con tecnologie open source, in particolare su piattaforma LAMP (Linux, Apache, MySQL, Php): attraverso il protocollo HTTP il nodo Slave invia alla web application scritta in linguaggio Php la stringa così come ricevuta dal Master; la web application, quindi, estrae i parametri dalla stringa e li inserisce nelle opportune tabelle del database. È prevista anche una semplice interfaccia grafica per la consultazione dei dati salvati.

2 - Implementazione del nodo in versione Ethernet

Il nodo Slave come già detto ha il compito di prelevare i dati ricevuti via I2C e di inviarli ad un web server tramite Internet; si è scelto di realizzare una versione che utilizzasse Ethernet dato che è una tecnologia di collegamento estremamente diffusa, sia in ambito *consumer* che in ambito industriale (si pensi ai numerosi standard per l'*Industrial Ethernet* nati negli ultimi anni, quali PROFINET o EtherCat).

La comunicazione I2C in Arduino è gestita attraverso la libreria Wire.h. Tale libreria [9] mette a disposizione primitive di alto livello per la gestione della comunicazione, quali ad esempio:

- write(): usata per inviare un byte sul bus
- read(): usata per prelevare un byte dal buffer di ricezione interno
- beginTransmission(address): usata dal master per inviare dati ad uno slave
- beginTransmission(address): usata dal master per inviare dati ad uno slave
- onReceive(handler): usata per registrare una funzione che venga invocata ad ogni ricezione per gestire il byte ricevuto
- onRequest(handler): usata nei dispositivi Slave per registrare una funzione che venga invocata ad ogni richiesta del Master

Il controller Ethernet invece è controllabile con le funzioni della libreria Ethernet.h. Come già detto, questa libreria è molto vasta ed offre numerose funzioni. In questo caso si sono utilizzate solo le funzioni della classe Client. In particolare, deve essere creato l'oggetto Client con l'opportuno costruttore, il quale mette a disposizione numerosi metodi, fra cui:

- connect(): usata per effettuare la connessione
- stop(): usata per terminare la connessione
- print():usata per inviare una stringa di testo sulla socket TCP
- read():usata per leggere dati dalla socket TCP
- flush(): usata per svuotare il buffer di ricezione

La struttura dello sketch è la seguente: dopo la dichiarazione delle variabili necessarie, si ha un *setup()* nel quale vengono avviate le periferiche necessarie (I2C, Ethernet e Seriale). Quindi viene registrata la funzione *handler* che dovrà gestire l'evento "byte ricevuto via I2C", detta *receiveEvent()*, e successivamente descritta. Nel *loop()* viene eseguita la procedura di invio dati solo se il nodo è nello stato di "SEND", altrimenti non viene eseguita alcuna operazione. *L'handler receiveEvent()* viene chiamato ad ogni byte ricevuto; è pertanto necessario mantenere il carico computazionale e quindi il tempo di

esecuzione molto basso, dato che deve essere completato entro l'arrivo del byte successivo. La macchina a stati finiti è realizzata attraverso quattro *flags* booleani, inizialmente tutti posti al valore di *false*:

- StartFrame: *true* se il carattere precedentemente ricevuto era !
- RicState: *true* se il nodo è in stato RECEIVE
- AsteriskReceived: *true* se il carattere precedentemente ricevuto era *
- Sending: *true* se il nodo è in stato SEND

Si veda il paragrafo 2.2.4 per il funzionamento e il loro utilizzo.

2.1 - Implementazione (versione con metodo GET)

2.1.1 - Inclusione librerie e dichiarazione di variabili

Definendo la macro DEBUG viene abilitata la stampa su interfaccia seriale delle stringhe ricevute via I2C e di altre informazioni di controllo sul corretto comportamento del nodo. Per alleggerire il carico computazionale e per avere un comportamento più stabile, è bene disabilitare tale funzione in contesti che non siano di progetto.

```
#define DEBUG
```

La seguente costante è la dimensione del buffer di ricezione, dove viene salvata la stringa ricevuta da I2C e da inviare al web server.

```
#define SIZE 300
```

La seguente costante è l'indirizzo IP da utilizzare nell'header *host* nella richiesta HTTP.

```
#define HTTP_HOST "192.168.1.1"
```

Quindi si includono le librerie necessarie.

```
#include <Wire.h>

#include <SPI.h>

#include <Ethernet.h>
```

Vengono poi dichiarate le variabili necessarie all'interfaccia di rete Ethernet.

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };

byte ip[] = { 192, 168, 1, 2 };

byte gateway[] = { 192, 168, 1, 254 };

byte subnet[] = { 255, 255, 255, 0 };

byte server[] = { 192, 168, 1, 1 };
```

Si noti che l'indirizzo del server di destinazione deve essere specificato sotto forma di indirizzo IP; non è infatti previsto l'utilizzo di DNS in questa libreria. Tale indirizzo è poi specificato nuovamente come stringa; essa deve indicare necessariamente un host, ovvero avere forma *www.site.com* oppure di indirizzo IP; non devono essere quindi usati identificativi di protocolli di livello superiore, quali ad esempio "http://" per il protocollo HTTP. Questo è necessario perché questa stringa verrà usata poi per la creazione dell'*header Host* del pacchetto HTTP, che non prevede appunto forme diverse. Inoltre si dichiara il percorso del file richiesto, nella forma */path/file.ext*; in questo caso si indica il path del file save.php della web application.

```
char address[]="192.168.1.1";

char resource="/arduino/save.php";
```

Successivamente si dichiara l'oggetto client, l'indirizzo che dovrà essere usato nel protocollo I2C, il byte ricevuto via I2C, il buffer di ricezione e due contatori necessari oltre.

```
Client client(server, 80);

const int Slave1 = 1;

byte rx_wire;

byte BufferRX[SIZE];

int Count,i;
```

Infine i *flags* booleani.

```
boolean startFrame=false;

boolean RicState=false;
```

```
boolean AsteriskReceived=false;

boolean Sending=false;
```

2.2.2 - Setup

Come già detto, i due compiti di questa parte sono di inizializzare le periferiche e di registrare l'*handler* di ricezione per I2C.

```
void setup() {


  Wire.begin(Slave1)


  Ethernet.begin(mac, ip, gateway, subnet);


  Serial.begin(9600);


  #ifdef DEBUG

  Serial.println("Setup Slave Done !");

  #endif


  Count=0;


  Wire.onReceive(receiveEvent);


}
```

2.1.3 - Loop

La funzione *loop()* viene eseguita continuamente dal microcontrollore; all'inizio della sua implementazione, vi è un controllo sul *flag* Sending; se è vero, ovvero il sistema è nello stato di SEND, esegue l'invio del buffer di ricezione al web server, altrimenti non esegue nulla. Essa viene periodicamente interrotta dall'*handler* di ricezione, il quale a sua volta controlla prima di tutto il *flag* Sending ed esegue azioni solo se è falso, garantendo che nello stato di SEND la funzione *loop()* possa agire ininterrotta.

```
void loop(){

   while(Sending){ //solo se ci si trova nello stato di SENDING
```

Nel caso di DEBUG definito, il buffer di ricezione, oltre ad essere inviato alla web application, viene stampato su seriale. Prima di tutto si stampano i due caratteri costituenti lo *Start of Frame*.

```
#ifdef DEBUG

Serial.print("!!,");
```

Il contatore *i* rappresenta i caratteri finora stampati, ovvero la posizione attuale nel buffer di ricezione. Quindi, partendo da 3 (i primi tre caratteri sono già stati stampati), finché non si incontrano due simboli dollaro (simbolo "*$*") consecutivi, ogni byte del buffer (si ricordi che è composto da byte separati da virgole) viene stampato con il suo valore decimale, eccetto che per le virgole (separatori) che vengono stampate come carattere; ossia, se il byte vale 58, verrà stampato "58" e non il carattere ":" (ovvero la rappresentazione ASCII del valore 58), dato che questo numero rappresenta parte di una grandezza riguardante il veicolo.

```
i=3;

while(!(BufferRX[i]=='$' && BufferRX[i+1]=='$')){

   if(BufferRX[i]==','){

      Serial.print(BufferRX[i]);

   }else{

      Serial.print(BufferRX[i], DEC);
```

```
        }
        i++;
    }
```

Si sceglie di non inviare alla web application i due caratteri dollaro (simbolo *"$"*) dato che potrebbero essere causa di incompatibilità. Quindi vengono saltati insieme alla virgola che li separa dal primo byte successivo; in tutto, tre caratteri.

```
    i += 3;
```

Quindi fino ai due asterischi (simbolo *"*"*) ogni byte è stampato come carattere; in altre parole, se il byte vale 58, verrà stampato il carattere : ovvero la sua rappresentazione ASCII. Questo perché il GPS fornisce nativamente i suoi dati come stringhe di caratteri, per cui per esempio il nodo Master invia allo Slave l'orario nella forma di 152532 ovvero sei byte rispettivamente aventi valore 49 (ovvero il carattere "1" in ASCII), 52 (il carattere "5" in ASCII) ecc.

```
    while(BufferRX[i]!='*' && BufferRX[i+1]!='*'){
        Serial.print(BufferRX[i]);
        i++;
    }
    Serial.print(BufferRX[i]);//stampa ,
   Serial.print(BufferRX[i+1]);//stampa *
   Serial.print(BufferRX[i+2]);//stampa *
    Serial.println();//riga vuota
    #endif
```

In seguito si ha la parte di invio dati. Innanzi tutto si effettua la connessione all'indirizzo IP del web server, aprendo una *socket* TCP.

```
    if (client.connect()) {
        #ifdef DEBUG
        Serial.println("connected");
```

```
#endif
```

Quindi attraverso questa viene inviato il pacchetto HTTP, il quale deve avere forma [10]:

```
GET /path/file.php?param1=value1&param2=value2[&…] HTTP/1.1
Host: www.host1.com:80
[blank line]
```

Verrà richiesto lo script Php che dovrà occuparsi del salvataggio dei dati (si veda il capitolo 4) e passando un solo parametro chiamato obj, il cui contenuto coinciderà col buffer di ricezione.

```
client.print("GET ")

client.print(resource);

client.print("?obj=");
```

Quindi, con un procedimento del tutto simile a quello utilizzato per la stampa del buffer su seriale, si inviano i dati.

```
i=3;
   client.print("!,");
   while(!(BufferRX[i]=='$' && BufferRX[i+1]=='$')){
   if(BufferRX[i]==','){
         client.print(BufferRX[i]);
      }else{
         client.print(BufferRX[i], DEC);
      }
      i++;
   }
   i += 3;


   while(BufferRX[i]!='*' && BufferRX[i+1]!='*'){
```

```
            client.print(BufferRX[i]);
            i++;
         }
         client.print(BufferRX[i]);
      client.print(BufferRX[i+1]);
```

Infine si conclude la *request* HTTP.

```
      client.println(" HTTP/1.1");
      client.print("Host: ");
      client.println(HTTP_HOST);
      client.println();
```

In caso di errore, se DEBUG è definito si genera un messaggio di errore.

```
      }else{
         #ifdef DEBUG
         Serial.println("connection failed");
         #endif
      }
```

Si termina poi la connessione e si azzerano sia il buffer di ricezione (con la funzione memset) sia il contatore associato. Quindi si esce dallo stato di SEND.

```
      client.stop();

      Count=0;
      memset(BufferRX,0,SIZE);
      Sending=false;
   }
}
```

2.1.4 - Handler di ricezione

L'*handler* viene chiamato ad ogni byte ricevuto; appena invocato, preleva il byte dal buffer interno del microcontrollore per evitare di saturarlo.

```
void receiveEvent(int howMany){

  while(Wire.available()){

      rx_wire = Wire.receive();
```

Se il nodo è in SEND, non esegue altre operazioni; ovvero, il byte ricevuto non viene salvato nel buffer di ricezione, quindi viene scartato. Altrimenti, si analizza il byte ricevuto per determinare se e quale transizione di stato deve essere fatta. In particolare, il meccanismo è il seguente: all'avvio, il sistema si trova nello stato DROP ALL; alla ricezione di un byte, se è un punto esclamativo (ovvero il primo carattere di un frame) lo si salva come primo carattere del buffer di ricezione e si pone a *true* il *flag* StartFrame. Se il successivo byte è un secondo punto esclamativo, allora significa che si è ricevuto lo S*tart of Frame* e quindi si passa nello stato RECEIVE, dove ogni byte viene salvato nel buffer di ricezione. Durante la ricezione analogamente, si controlla ogni byte ricevuto (che viene sempre salvato nel buffer) e quando si rilevano due asterischi consecutivi (*End of Frame*) si passa nello stato SEND.

```
      if(!Sending){

         if(!RicState && rx_wire=='!' &&  !startFrame ){

            startFrame=true;

            BufferRX[Count]=rx_wire;

            Count++;

         }else if(!RicState && rx_wire=='!' && startFrame){

             BufferRX[Count]=rx_wire;

            Count++;

            RicState=true;

            startFrame=false;
```

```
          }else if(!RicState && rx_wire!='!'){
             Count=0;
             startFrame=false;
          }else if(RicState){
             BufferRX[Count]=rx_wire;
             Count++;
          }

          if(RicState && rx_wire=='*' && !AsteriskReceived){
             AsteriskReceived=true;
          }else if(RicState && rx_wire!='*' && AsteriskReceived){
          AsteriskReceived=false;
          }else if(RicState && rx_wire=='*' && AsteriskReceived){
             AsteriskReceived=false;
             RicState=false;
             Sending=true;
          }
       }
    }
}
```

2.2 - Implementazione (versione con metodo POST)

Il codice è essenzialmente lo stesso. Cambia solo il formato del pacchetto HTTP, che ora dovrà essere [10]:

```
POST /path/file.php HTTP/1.1
Host: www.host1.com:80
Content-Type: application/x-www-form-urlencoded
Content-Length: [payload dimension]
[blank line]
param1=value1&param2=value2[&…]
```

Si nota che in questo caso è necessaria la dimensione esatta del *payload*. È quindi stata aggiunta una funzione apposita, che conta i caratteri tenendo conto che i byte prima dei simboli dollaro (*$$*) saranno inviati come stringhe, ossia il byte di valore 128 sarà inviato come sequenza di caratteri 1 - 2 - 8, ovvero tre byte. Quindi per i caratteri diversi dalle virgole prima dei simboli dollaro, se hanno valore compreso fra [0;9] la dimensione dello stampato sarà 1; se appartengono all'intervallo [10;99] la dimensione sarà 2; se appartengono infine all'intervallo [100;255] la dimensione sarà 3. Il valore iniziale della variabile *dimension* è pari a 7 per tenere conto della parte iniziale (obj=, quattro caratteri) e i simboli punto esclamativo iniziali (!!, tre caratteri).

```
int bufDim(){
   int dimension=7;
   int i=3;

   while(!(BufferRX[i]=='$' && BufferRX[i+1]=='$')){
      if(BufferRX[i]==','){
         dimension++; //virgola singola
       }else{
          if(BufferRX[i]<10)
             dimension++;
          else if(BufferRX[i]<100)
             dimension+=2;
          else
             dimension+=3;
```

```
        }
        i++;
    }
```

I caratteri successivi ai simboli dollaro riguardano i dati GPS e ogni byte rappresenta un carattere, quindi conta come uno nel computo della dimensione.

```
    i+=3;
    while(BufferRX[i]!='*' && BufferRX[i+1]!='*'){
        dimension++;
        i++;
    }
    dimension+=3;
    return dimension;
}
```

Con questa funzione, è possibile generare la *request* HTTP con metodo POST semplicemente modificando la sezione apposita.

```
            client.print("POST ");
            client.print(resource);
            client.print(" HTTP/1.1");
            client.print(13,BYTE);//\r
            client.print(10,BYTE);//\n
            client.print("Host: ");
           client.print(address);
            client.print(13,BYTE);
            client.print(10,BYTE);
            client.println("Content-Type: application/x-www-form-
urlencoded");
```

```
client.print("Content-Length: ");

client.print(bufDim());

client.print(13,BYTE);

client.print(10,BYTE);

client.print(13,BYTE);

client.print(10,BYTE);

client.print("obj=!!,");
```

E a seguire si stampano i byte che compongono il buffer di ricezione in modo identico a quanto fatto per il caso GET.

3 - Implementazione del nodo in versione GPRS

La comunicazione con il modulo GPRS avviene tramite l'invio di comandi AT, ovvero di caratteri ASCII, attraverso la linea seriale. Ad ogni comando, il modulo risponde inviando una risposta, sempre sotto forma di caratteri via seriale, in un certo tempo variabile e dipendente sia dal tipo di comando che da altri fattori casuali quali tempi di risposta della rete GSM ecc. Il primo approccio seguito è stato quindi di strutturare la comunicazione Arduino / modulo GPRS secondo un paradigma del tipo:

```
Serial.println("COMANDO AT");

wait(TEMPO_EMPIRICO);
```

Come suggerito [11] dal venditore del modulo stesso. Tuttavia, ben presto sono emersi i limiti e i problemi li di questo approccio, principalmente identificabili in:

- Si è costretti a sovradimensionare di molto il tempo di attesa, penalizzando le prestazioni del nodo
- Ciò nonostante, non si ha garanzia che il comando sia andato a buon fine, dato che non controlla la risposta fornita

L'alternativa proposta è la creazione di una libreria che gestisca in modo opportuno la comunicazione seriale a carattere nelle due direzioni e fornisca all'utente funzioni di alto livello che nascondano i dettagli implementativi e i comandi AT necessari. Tale libreria offre numerose funzionalità, molte delle quali non strettamente necessarie all'implementazione del nodo Slave; tuttavia, si è pensato di sviluppare questo progetto nell'ottica di sviluppi futuri.

3.1 - Libreria GPRS

Grazie al fatto che l'ambiente Arduino è open - source, è stato possibile riusare parte del codice appartenente ad altri progetti sviluppati da terze parti e rilasciate nel pubblico dominio. In particolare, si è avuto accesso ad una libreria progettata per un modulo GPRS di un diverso produttore, facente parte del progetto denominato gsm-playground [13]. Essa è strutturata in tre parti:

- AT.cpp: si occupa della comunicazione seriale a carattere fra la *board* Arduino e il modulo GPRS e dell'invio di comandi AT e della lettura delle risposte
- GSM.cpp: fornisce funzioni relative alla rete GSM (chiamata, invio di sms, gestione rubrica ecc.)
- GPRS.cpp: fornisce funzioni relative alla rete GPRS (apertura di connessione, richiesta di una pagina web ecc.)

Essendo progettata per un diverso modulo, i moduli GSM e GPRS non potevano essere usati; tuttavia, la comunicazione seriale segue gli stessi principi, per cui è stato possibile usare una parte del codice del modulo AT.cpp. Il concetto su cui si basa la gestione della comunicazione è la definizione di un processo di ricezione dei caratteri inviati dal modulo alla *board*; tale processo riempie un buffer di ricezione e si considera terminato in due situazioni: quando non si riceve alcun carattere entro un predefinito timeout o quando, dopo aver ricevuto alcuni caratteri, non ne ricevo altri per un certo tempo, anch'esso predefinito. In particolare, sono state usate le seguenti funzioni (se ne riporterà in questa sede solo una breve descrizione; per una più precisa descrizione si rimanda alla relativa documentazione):

- `void RxInit()`: da inizio al processo di ricezione.

- `byte HasRxFinished(unsigned long start_reception_tmout)`: controlla se il processo di ricezione è finito o è ancora in corso.

- `byte WaitResp(unsigned long start_reception_timeout)`: attende per un certo tempo che il modulo inizi la trasmissione di caratteri quale risposta ad un comando impartito.

- `byte WaitResp(unsigned long start_reception_timeout, char const *expected_resp_string)`: attende per un certo tempo che il modulo inizi la trasmissione di caratteri quale risposta ad un comando impartito e controlla che la risposta corrisponda a quella prevista.

- `byte IsStringReceived(char const *compare_string)`: controlla che nel buffer di ricezione sia presente la stringa passata come parametro.

- `byte SendATCmdWaitResp(char const *AT_cmd_string, char const *response_string, unsigned long start_reception_timeout)`: invia al modulo un comando AT e attende una risposta; controlla inoltre che la risposta fornita sia quella prevista.

Inoltre, essendo necessario codificare le stringhe da inserire nei pacchetti HTTP secondo lo standard url-encode, si è fatto uso del codice facente parte del progetto AVR-netino [13], in particolare delle seguenti funzioni (anche di queste se ne riporterà in questa

sede solo una breve descrizione, rimandando alla relativa documentazione per approfondimenti):

- `void int2h(char c, char *hstr):` converte un singolo carattere ASCII in una stringa di tre caratteri, due rappresentanti il suo valore esadecimale più il terminatore.

- `unsigned char h2int(char c):` converte una singola cifra esadecimale nel suo valore intero.

- `void Urlencode(char *str,char *urlbuf):` codifica secondo lo standard url-encode la stringa passata come primo parametro e salva il risultato nell secondo parametro.

- `void Urldecode(char *urlbuf):` decodifica la stringa passata come parametro.

A partire da questo codice, è stata strutturata la libreria Colella_GPRS, che si compone di due file: Colella_GPRS.h e Colella_GPRS.cpp.

3.1.2 - File Header

Il file Colella_GPRS.h riporta alcune definizioni e le dichiarazioni dei metodi. La struttura è quella imposta da Arduino [12]:

```
#ifndef Colella_GPRS_h

#define Colella_GPRS_h

#include "WProgram.h"
```

Viene quindi inclusa la libreria string.h dato che si fa uso di funzioni operanti su stringhe.

```
#include "string.h"
```

È data poi la possibilità di definire o meno una constante, detta DEGUG, la quale, se definita, abilita le stampe su linea seriale di informazioni di debug e statistiche; deve essere usata con cautela e solo per fini di test, dato che la linea seriale è una sola e tali informazioni pertanto vengono ricevute anche dal modulo GPRS e potrebbero causare malfunzionamenti.

```
#define DEBUG
```

Per l'accensione del modulo, si ricorda, si deve portare per almeno 629ms il piedino digitale numero 2 della *board* ad un livello logico alto; la successiva definizione riguarda questo.

```
#define ON_PIN 2
```

Si passa poi alla definizione dei vari timer usati per la gestione della comunicazione, rispettivamente riguardanti l'attesa da fare prima di inviare un comando AT, il massimo tempo che intercorre fra la ricezione di due caratteri (scaduto il quale la ricezione è considerata finita) e il tempo di attesa per l'inizio della ricezione di una pagina web dopo che ne è stata fatta richiesta.

```
#define AT_DELAY 500

#define INTERCHAR_TMOUT 20

#define WEB_TMOUT 500
```

Il processo di ricezione è implementato come una semplice macchina a stati finiti a due stati, sotto definiti:

```
#define RX_NOT_STARTED 0

#define RX_ALREADY_STARTED 1
```

Vi sono poi i valori ritornati da alcune funzioni, definiti come costanti per avere una maggiore leggibilità del codice.

```
#define RX_NOT_FINISHED 2

#define RX_FINISHED 3

#define RX_TMOUT_ERR 4

#define RX_FINISHED_STR_RECV 5

#define RX_FINISHED_STR_NOT_RECV 6

#define AT_RESP_ERR_NO_RESP 7

#define AT_RESP_OK 8

#define AT_RESP_ERR_DIF_RESP 9
```

Si definisce infine la lunghezza del buffer di comunicazione fra la *board* e il modulo GPRS e di quello destinato ad accogliere la pagina web richiesta.

```
#define COMM_BUF_LEN 50

#define WEB_PAGE_LEN 350
```

Si passa quindi alla definizione dei metodi dell'unica classe presente nella libreria; per l'implementazione e la descrizione di ognuno si veda oltre.

```
class Colella_GPRS{

   public:

      //general utility methods

      Colella_GPRS(void);

       void TurnOn();

       void TurnOff();

       void SetupGPRS(char const *apn, char const *usr, char
const *pwd);

       byte TCPConnect(char const *addr, int port);

       void TCPDisconnect();

       //gsm methods

       void Call(char const *number);

       void Hang();

       byte SendSMS(char const *number, char const *text);

       //web methods

       char* HTTPGet(char const *resource);

       char* HTTPPost(char const *resource, char const *data);

       void Prepare_Post_Data(char *data, int howMany, ...);

       void Urlencode(char *str,char *urlbuf);

       void Urldecode(char *urlbuf);
```

```
        //debug methods

        void PrintComBuf();

        void SignalStrength();

        //low level methods

       void RxInit();

        byte HasRxFinished(unsigned long start_reception_tmout);

        byte WaitResp(unsigned long start_reception_timeout);

        byte WaitResp(unsigned long start_reception_timeout, char
const *expected_resp_string);

        byte IsStringReceived(char const *compare_string);

        byte SendATCmdWaitResp(char const *AT_cmd_string, char
const *response_string, unsigned long start_reception_timeout);

        void int2h(char c, char *hstr);

        unsigned char h2int(char c);
```

Infine si dichiarano le variabili globali necessarie.

```
        //internal variables

        byte *p_comm_buf;  // pointer to the communication buffer

        byte rx_state;     // internal state of rx state machine

        unsigned long prev_time;   // previous time in msec.

        int comm_buf_len;    // num. of characters in the buffer

        byte comm_buf[COMM_BUF_LEN+1]; // communication buffer +1

                                            for 0x00 termination

        char *web_page;         // web page

        const char *address;    // web site address

};
```

```
#endif
```

3.1.2 - File Sorgente

Nel file Colella_GPRS.cpp è contenuta l'implementazione di tutti i metodi; l'implementazione di quelli riutilizzati dai progetti gsm-playground e AVR-netino è riportata in Appendice A. Nel seguito, si descriveranno invece i metodi originali, frutto del lavoro della presente tesi.

Il primo metodo è il costruttore, che crea l'oggetto Colella_GPRS dotato di tutti i metodi successivi. Semplicemente, imposta il pin di accensione come pin di output e la velocità della porta seriale.

```
Colella_GPRS::Colella_GPRS(){
   pinMode(ON_PIN, OUTPUT);
   Serial.begin(19200);
}
```

Il metodo successivo si occupa dell'accensione del modulo e di connettersi alla rete GSM.

```
void Colella_GPRS::TurnOn(){
   byte status;
   boolean isReg=false;
```

Per accendere il modulo si porta ad un livello di tensione alto il piedino opportuno per due secondi, quindi lo si riporta al livello basso.

```
   digitalWrite(ON_PIN,HIGH);
   delay(2000);
   digitalWrite(ON_PIN,LOW);
```

Nel caso di stampe di debug abilitate, si da comunicazione dell'avvenuta accensione.

```
   #ifdef DEBUG
```

```
   Serial.println("Started, waiting for network connection...");

   #endif
```

Quindi si attende che il modulo sia registrato presso la rete GSM, verificando ogni secondo lo stato attraverso il comando AT+CREG, che restituisce +CREG: 0,1 nel caso di registrazione avvenuta nella rete del proprio operatore oppure +CREG: 0,5 se in roaming.

```
   do{

      Serial.println("AT+CREG?");

      status=WaitResp(1000);

      if(status == RX_FINISHED)

         if(IsStringReceived("+CREG: 0,1") ||
IsStringReceived("+CREG: 0,5"))

            isReg=true;

      delay(1000);

   }while(!isReg);
```

Quindi si impostano gli sms nella modalità testo e si attendono 30 secondi, per essere certi che finiscano tutti i processi di agganciamento della portante, registrazione alla rete ecc. Infine, si da comunicazione della fine dell'esecuzione del metodo.

```
   Serial.println("AT+CMGF=1");

   delay(30000);

   #ifdef DEBUG

   Serial.println("Connected!");

   #endif

}
```

Il seguente metodo spegne il modulo, semplicemente impartendo il comando AT+PSCPOF.

```
void Colella_GPRS::TurnOff(){
```

```
   Serial.println("AT*PSCPOF");

}
```

Il seguente metodo imposta i parametri necessari alla connessione GPRS. In questo caso, dato che i comandi AT usati impostano parametri e non prevedono fallimento, si è scelto di usare l'approccio comando-attesa piuttosto che usare le più pesanti (dal punto di vista computazionale) funzioni implementate. Innanzi tutto, attiva il controllo di flusso svolto in hardware:

```
void Colella_GPRS::SetupGPRS(char const *apn, char const *usr,
char const *pwd){

   #ifdef DEBUG

   Serial.println("Starting GPRS setup...");

   #endif

   Serial.println("AT&k3");

   delay(1000);
```

Si imposta poi apn, username e password richiesti per la connessione:

```
   Serial.print("AT+KCNXCFG=0,");

   Serial.print(34,BYTE); //ascii 34 = "

   Serial.print("GPRS");

   Serial.print(34,BYTE);

   Serial.print(",");

   Serial.print(34,BYTE);

   Serial.print(apn);

   Serial.print(34,BYTE);

   Serial.print(",");

   Serial.print(34,BYTE);

   Serial.print(usr);
```

```
Serial.print(34,BYTE);

Serial.print(",");

Serial.print(34,BYTE);

Serial.print(pwd);

Serial.print(34,BYTE);

Serial.println();

delay(1000);
```

Si impostano poi i timer necessari; nell'ordine, essi sono [7]:

- <cnx cnf>: di tipo intero, è l'indice dell'insieme dei parametri necessari alla configurazione ella connessione; valore massimo 200.
- <tim1>: timeout della connessione in secondi. Deve avere valori compresi fra 15s e 120s (30s per default).
- <nbtrial>: numero di tentativi di connessione alla rete. Deve avere valore compreso fra 2 e 4 (2 per default).
- <tim2>: tempo di persistenza in secondi. Deve avere valori fra 60s e 300s (60s per default, 0 disattivato ovvero la connessione non si chiude da sola).

```
Serial.println("AT+KCNXTIMER=0,60,2,70");

delay(1000);
```

Si imposta poi il profilo di servizio.

```
Serial.println("AT+KCNXPROFILE=0");

delay(1000);
```

Si effettua quindi la connessione alla rete GPRS e si da infine comunicazione del successo dell'operazione.

```
Serial.println("AT+CGATT=1");

delay(1000);

#ifdef DEBUG

Serial.println("GPRS setup done!");
```

```
   #endif

}
```

I metodi successivi si occupano di avviare e terminare una chiamata vocale rispettivamente con i comandi ATD<numero> e ATH.

```
void Colella_GPRS::Call(char const *number){

   Serial.print("ATD");

   Serial.print(number);
```

Il punto e virgola seguente impone una chiamata vocale; senza, il modulo effettua una chiamata dati.

```
   Serial.println(";");

   WaitResp(5000);

}

void Colella_GPRS::Hang(){

   Serial.println("ATH");

}
```

Il metodo seguente invia un messaggio di testo al numero specificato; il funzionamento è il seguente [7]: si imposta il numero col comando AT+CMGS=”<numero>”, quindi il modulo entra in modalità di ricezione testo, ovvero risponde con un simbolo > e inizia a accettare il testo del messaggio; per terminare l’immissione si deve inviare il carattere esadecimale 0x1A.

```
byte Colella_GPRS::SendSMS(char const *number, char const
*message){

  byte ret_val = 0;

  Serial.print("AT+CMGS=\"");

  Serial.print(number);

  Serial.println("\"");
```

```
  if(RX_FINISHED_STR_RECV == WaitResp(2000, ">")){
     Serial.print(message);
     delay(500);
     Serial.print(0x1a, BYTE);
```

Se ricevo la risposta +CMGS significa che l'sms è stato correttamente inviato.

```
     if(RX_FINISHED_STR_RECV == WaitResp(2000, "+CMGS"))
        ret_val = 1;
}
```

Il metodo ritorna 1 in caso di successo e 0 in caso di fallimento.

```
  return (ret_val);
}
```

Il metodo successivo apre una socket TCP con il server passato come parametro attraverso i comandi AT proprietari Sagem.

```
byte Colella_GPRS::TCPConnect(char const *addr, int port){
   address=addr;
```

Innanzi tutto si impostano indirizzo IP e porta del server.

```
   Serial.print("AT+KTCPCFG=0,0,");
   Serial.print(34,BYTE);
   Serial.print(address); //ip address
   Serial.print(34,BYTE);
   Serial.print(",");
   Serial.println(port); //tcp port
```

Quindi si apre la connessione, per un massimo di tre tentativi.

```
   byte status;
```

```
   for(int i=0; i<3; i++){
      status = SendATCmdWaitResp("AT+KTCPCNX=1","OK",7000);
      if(status == AT_RESP_OK)
         return 1;
       delay(500);
   }

   #ifdef DEGUG
   Serial.println("Network error");
   #endif

   return 0;
}
```

Il metodo ritorna 1 in caso di avvenuta connessione oppure 0 in caso di errore.

Il metodo successivo chiude la socket precedentemente aperta, semplicemente impartendo il comando AT+KTCPCLOSE.

```
void Colella_GPRS::TCPDisconnect(){
   SendATCmdWaitResp("AT+KTCPCLOSE=1,1","OK",2000);
}
```

Il metodo successivo richiede una pagina web attraverso la socket TCP aperta attraverso il metodo HTTP GET.

```
char* Colella_GPRS::HTTPGet(char const *resource){
```

Si calcola il totale dei byte che saranno inviati nella socket, dato che il comando di invio dati AT+KTCPSND lo richiede; 27 sono i byte richiesti per la costruzione del pacchetto HTTP, a cui vanno aggiunti le lunghezze del nome dell'host e del percorso del file

richiesto. Si crea quindi la stringa contenente il comando di invio dati con il numero giusto di byte da inviare.

```
int bytesToSend = 27 + strlen(resource) + strlen(address);

char sndCommand[25]="";

sprintf(sndCommand, "AT+KTCPSND=1,%d", bytesToSend);
```

Quindi si alloca in memoria uno spazio sufficiente a contenere la pagina web richiesta con la funzione malloc; con la funzione memset si azzera tale spazio.

```
if((web_page = (char *)malloc(WEB_PAGE_LEN))== NULL){
   #ifdef DEBUG
   Serial.println("Unable to allocate memory for web page");
   #endif
   return NULL;
}
memset(web_page,0,WEB_PAGE_LEN);
delay(1000);
```

Quindi si invia il comando AT di invio dati. Tale comando può rispondere positivamente (CONNECT, e quindi accettare dati) o con un messaggio di errore (e in tal caso si libera lo spazio allocato per la pagina web e si restituisce NULL).

```
byte status = SendATCmdWaitResp(sndCommand, "CONNECT", 5000);
if(status != AT_RESP_OK){
   #ifdef DEBUG
   Serial.println("KTCPSND failed.");
   #endif
   delay(500);
    free(web_page);
   return NULL;
```

```
}
```

In caso di esito positivo, si procede a inviare il pacchetto HTTP opportunamente formato:

```
Serial.print("GET ");

Serial.print(resource);

Serial.println(" HTTP/1.1");

Serial.print(13,BYTE);

Serial.print("Host: ");

Serial.println(address);

Serial.print(13,BYTE);

Serial.print(10,BYTE);

Serial.print(13,BYTE);
```

L'invio dati termina quando si invia la seguente stringa di terminazione:

```
Serial.print("--EOF--Pattern--");
```

Quindi si attendono le due risposte previste, OK e +KTCPDATA (indica il numero di byte ricevuti).

```
WaitResp(2000);//waits for OK

WaitResp(2000);//waits for +KTCPDATA
```

Quindi si inizia a salvare i caratteri ricevuti componenti la pagina web nello spazio precedentemente riservato. Il comando di ricezione dati è AT+KTCRCV=1,<byte da ricevere>; impongo di ricevere al massimo un numero di caratteri pari alla dimensione del buffer allocato per la pagina web, per evitare overflow.

```
Serial.flush();

Serial.print("AT+KTCPRCV=1,");

Serial.println(WEB_PAGE_LEN);
```

La ricezione prevede di salvare ogni carattere disponibile finché, ricevuto un carattere, non se ne ricevono altri fino allo scadere di un predefinito timeout.

```
int k=0;

unsigned long prev_time=millis();

while((millis()-prev_time)<=WEB_TMOUT){

   if(Serial.available()){

      web_page[k]=(byte(Serial.read()));

        k++;

        prev_time=millis();

   }

}
```

Si inserisce in coda il terminatore di stringa e si restituisce il puntatore al primo byte del buffer.

```
   web_page[k]=0x00;//string terminator

   return web_page;

}
```

Del precedente metodo è prevista anche una versione che utilizza il metodo POST. La differenza principale risiede nella diversa formazione del pacchetto HTTP. La dichiarazione prevede che siano passati come parametro i dati da passare alla web application indicata dal parametro *resource*. Tali dati devono essere codificati secondo lo standard url-encode. Per far questo si può usare il metodo Prepare_Post_Data successivamente presentato.

```
char* Colella_GPRS::HTTPPost(char const *resource, char const
*data){
```

Il metodo è esattamente uguale a quello che utlizza GET a meno della parte che forma il pacchetto HTTP.

```
   int bytesToSend = 95 + strlen(resource) + strlen(address) +

   strlen(data);
```

```
char sndCommand[25]="";
sprintf(sndCommand, "AT+KTCPSND=1,%d", bytesToSend);

if((web_page = (char *)malloc(WEB_PAGE_LEN))== NULL){
   #ifdef DEBUG
    Serial.println("Unable to allocate memory for web page");
    #endif
    return NULL;
}

memset(web_page,0,WEB_PAGE_LEN);
delay(1000);

byte status = SendATCmdWaitResp(sndCommand, "CONNECT", 5000);
if(status != AT_RESP_OK){
   #ifdef DEBUG
   Serial.println("KTCPSND failed.");
   #endif
   delay(500);
    free(web_page);
   return NULL;
}
```

La seguente parte differisce da quella in versione GET.

```
Serial.print("POST ");
```

```
   Serial.print(resource);

   Serial.print(" HTTP/1.1");

   Serial.print(13,BYTE);//\r

   Serial.print(10,BYTE);//\n

   Serial.print("Host: ");

   Serial.print(address);

   Serial.print(13,BYTE);

   Serial.print(10,BYTE);

   Serial.print("Content-Type: application/x-www-form-
urlencoded");

   Serial.print(13,BYTE);

   Serial.print(10,BYTE);

   Serial.print("Content-Length: ");

   Serial.print(strlen(data));

   Serial.print(13,BYTE);

   Serial.print(10,BYTE);

   Serial.print(13,BYTE);

   Serial.print(10,BYTE);

   Serial.print(data);

   Serial.print("--EOF--Pattern--");



   WaitResp(2000);//waits for OK

   WaitResp(2000);//waits for +KTCPDATA



   Serial.flush();
```

```
   Serial.print("AT+KTCPRCV=1,");
   Serial.println(WEB_PAGE_LEN);

   int k=0;
   unsigned long prev_time=millis();
   while((millis()-prev_time)<=WEB_TMOUT){
      if(Serial.available()){
         web_page[k]=(byte(Serial.read()));
          k++;
          prev_time=millis();
      }
   }
   web_page[k]=0x00;//string terminator

   return web_page;
}
```

Il metodo successivo prende in ingresso una certo numero di coppie parametro/valore da inviare con metodo POST e crea una stringa url-encoded correttamente formattata e pronta per l'invio. Per esempio, se si invoca Prepare_Post_Data(data, 4, "param1", "20", "param2", "è?a") alla fine dell'esecuzione la stringa data avrà valore "param1=20&param2=%c3%a8%3fo". Questa stinga, inviata con il metodo POST a una web application, permette ad essa di risalire correttamente ai valori dei due parametri param1 e param2 passati. Si noti che 4 è il numero di parametri/valori passati.

```
void Colella_GPRS::Prepare_Post_Data(char *data, int howMany,
...){
```

```
if(howMany % 2 != 0){
   #ifdef DEBUG
    Serial.println("Error, howMany must be even");
   #endif
    return;
}
```

Questo metodo fa uso di una struttura avanzata del linguaggio C, ovvero le va_list [15].

```
va_list marker;
va_start(marker,howMany); //initialize variable arguments
char *i, buf[100];
int j=0;
```

Per ogni parametro passato, si ha che quelli di posto pari rappresentano i nomi dei parametri mentre quelli di posto dispari sono i valori (da codificare). Nel caso dei nomi dei parametri, si appendono alla stringa finale così come sono, seguiti dal simbolo "=".

```
while(howMany!=0){
   i = va_arg(marker, char*);
   if(j % 2 == 0){ //it's a parameter
      strcat(data,i);
      strcat(data,"=");
```

Nel caso dei valori, vengono codificati con il metodo Urlencode. Fuori eccetto che per l'ultimo valore, ognuno è seguito dal simbolo &, necessario per accodare più parametri (si ricordi che la stringa deve avere formato param1=20&param2=%c3%a8%3fo&param3=… ecc).

```
      }else{ //it's a value
         Urlencode(i, buf);
         strcat(data,buf);
```

```
         if(howMany!=1)

            strcat(data,"&");

      }

      j++;

      howMany--;

   }


   va_end(marker); //reset variable arguments

}
```

Sono infine previsti due metodi di debug. Il primo stampa su seriale informazioni riguardo l'intensità del segnale. Ciò viene fatto attraverso il comando AT+CSQ, il quale [7] restituisce l'intensità del segnale in una scala a 32 valori aventi il seguente significato:

- 0: -113dBm o meno
- 1: -111dBm
- 2 … 30: -109dBm … -53dBm
- 31: -51dBm o più

```
void Colella_GPRS::SignalStrength(){

   #ifdef DEBUG

   Serial.println("Signal strength. Scale: 0=-113dBm or less, 1=-
111dBm");

   Serial.println("2...30=-109...-53dBm, 31=-51dBm or greater");

   #endif

   delay(200);

   Serial.println("AT+CSQ");

   byte status=WaitResp(1000); //one second is enough to receive
the response (it's almost immediate!)

   #ifdef DEBUG
```

```
   if(status == RX_FINISHED)
      PrintComBuf();
   #endif
}
```

Il secondo funziona solo se è stata definita la costante DEBUG e stampa l'intero contenuto del buffer di comunicazione utilizzato per la ricezione delle risposte del modulo GPRS.

```
void Colella_GPRS::PrintComBuf(){
   #ifdef DEBUG
   Serial.println();
   Serial.println("Communication Buffer Content");
   for(int i=0; i<COMM_BUF_LEN; i++)
      Serial.print(comm_buf[i]);
   Serial.println();
   #endif
}
```

A corredo della libreria sono forniti cinque esempi che illustrano l'uso dei vari metodi, in particolare:

- Call
- HTTPGet
- HTTPPost
- SendSMS
- Urlencode

In questo modo, aggiungendo la libreria alla cartella *libraries* del software Arduino, si ottiene il risultato mostrato in figura 10.

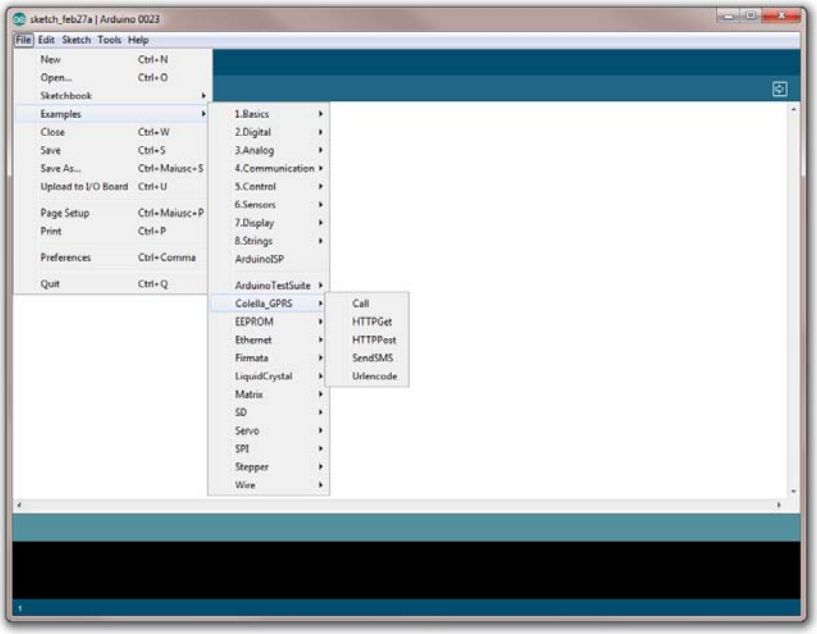


Fig. 10 - Menù della GUI Arduino relativo alla libreria creata

3.2 - Implementazione (versione con metodo GET)

Verrà descritto in seguito l'implementazione del nodo Slave che utilizza il canale GPRS per l'invio dei dati ricevuti. Per la creazione del software si è fatto uso di alcune funzioni della libreria Colella_GPRS, fondamentali per la corretta comunicazione col modulo.

Lo schema di base dello sketch è lo stesso di quello visto per la versione Ethernet; viene implementata la macchina a stati finiti rappresentata in figura 9, con la differenza che lo stato di SEND ora deve realizzare la connessione GPRS.

3.2.1 - Inclusione librerie e dichiarazione di variabili

Innanzi tutto si includono le librerie necessarie alla gestione del modulo GPRS e della comunicazione I2C.

```
#include <Colella_GPRS.h>

#include <Wire.h>
```

Quindi si dimensiona il buffer di ricezione I2C.

```
#define SIZE 300
```

Si definisce l'Access Point Name (dipendente dall'operatore telefonico in uso).

```
#define APN "apn.operator.it"
```

Si dichiara l'oggetto Colella_GPRS, di cui si useranno i metodi successivamente.

```
Colella_GPRS module;
```

Si dichiarano i parametri necessari alla connessione al web server.

```
char address[]="www.web_site.it";

char resource[]="/folder/save.php";

int port=80;
```

Si dichiarano alcune variabili necessarie successivamente.

```
char sndCommand[25]=""; //commando di invio dati

const int Slave1 = 1; //indirizzo I2C dello Slave

byte rx_wire; //byte ricevuto da I2C

byte BufferRX[SIZE]; //buffer di ricezione I2C

int Count,i; //contatori

byte sent=0; //contatore messaggi inviati
```

Infine si dichiarano e inizializzano ai valori opportuni i flag necessari al funzionamento a stati.

```
boolean startFrame=false;

boolean RicState=false;

boolean AsteriskReceived=false;

boolean Sending=false;
```

Vi è poi la funzione bufDim() che calcola la dimensione in byte della stringa che deve essere inviata al web server, già descritta al paragrafo 2.2.

```
int bufDim(){

[…]

}
```

3.2.2 - Setup

Questo blocco di codice si occupa di svolgere alcune operazioni preliminari atte a portare il sistema in condizioni operative. Innanzi tutto si inizializza la comunicazione I2C e si crea l'oggetto Colella_GPRS con il suo costruttore.

```
void setup() {

   Wire.begin(Slave1);

   pinMode(13, OUTPUT);

   module = Colella_GPRS();
```

Quindi si accende il modulo (si ricordi che il metodo TurnOn() si occupa anche di agganciare la portante) e si configura la connessione GPRS.

```
   module.TurnOn();

   module.SetupGPRS(APN, "", "");
```

Il contatore Count rappresenta i byte ricevuti dal bus I2C e inizialmente vale chiaramente zero.

```
   Count=0;
```

Il passaggio successivo consiste nel formare la stinga contente il comando AT per l'invio di dati attraverso la socket TCP. Esso consiste nel comando AT+KTCPSND=1,<numero byte da inviare>. Come si nota, si richiede di sapere in anticipo il numero di byte da inviare; si procede quindi al loro calcolo tenendo conto che per l'ossatura del pacchetto HTTP (ovvero per le parti fisse, quali "GET", "HTTP/1.1", "Host: " ecc.) sono richiesti 25B,

a cui devono essere aggiunte le lunghezze del file richiesto (variabile *resource*), dell'host (variabile *address*) e della stringa inviata (calcolata con la funzione bufDim).

```
int bytesToSend = 25 + strlen(resource) + strlen(address) + bufDim();

bytesToSend=abs(bytesToSend);

sprintf(sndCommand, "AT+KTCPSND=1,%d", bytesToSend);
```

Infine si apre la socket TCP e si imposta quale funzione dovrà fare da handler di ricezione per i byte ricevuti da I2C

```
module.TCPConnect(address, port);


Wire.onReceive(receiveEvent);

digitalWrite(13,HIGH);
}
```

3.2.3 - Loop

Il *loop()* esegue attività solo nel caso in cui ci si trovi nello stato di SEND, altrimenti non viene svolta alcuna operazione. Ricevuta tutta la stringa, il nodo Slave inizia con l'invio dei dati; si ricorda che durante questa fase ogni ulteriore byte ricevuto via I2C verrà scartato, fino al completamento dell'invio dati al database remoto.

```
void loop(){

   while(Sending){
```

Nel caso di stampe di DEBUG attivate, è previsto l'output di un timestamp che indica il tempo di inizio della fase di invio e della dimensione della stringa da inviare.

```
      #ifdef DEBUG

      Serial.print("\nOK received\t");

      Serial.print(millis());
```

```
      Serial.print("\t");

      Serial.println(bufDim());

      #endif
```

Si inizia quindi con l'invio dati. Si da al modem GPRS il comando AT necessario a comunicare che i successivi byte inviati dovranno essere trasmessi attraverso la socket TCP precedentemente aperta.

```
      byte status = module.SendATCmdWaitResp(sndCommand,
"CONNECT", 5000);
```

Questo commando fallisce solo se la socket è chiusa. In tal caso, si deve riaprirla, effettuando nuovamente le operazioni si configurazione e apertura della connessione con gli opportuni metodi.

```
      if(status != AT_RESP_OK){

         #ifdef DEBUG

         Serial.println("KTCPSND failed.");

         #endif

         delay(2000);

         module.SetupGPRS(APN, "", "");

         module.TCPConnect(address, port);
```

In caso di successo invece, si procede a inviare un pacchetto HTTP di tipo GET con lo stesso codice visto per il caso Ethernet; l'unica differenza risiede nel fatto che mentre nel caso Ethernet si usava il metodo client.print() ora è sufficiente inviare i byte direttamente su seriale, ovvero usando il Serial.print().

```
      }else{

         Serial.print("GET ");

         Serial.print(resource);

         Serial.print("?obj=");



         int i=3;
```

```
while(!(BufferRX[i]=='$' && BufferRX[i+1]=='$')){
   if(BufferRX[i]==','){
      Serial.print(BufferRX[i]);
   }else{
      Serial.print(BufferRX[i], DEC);
   }
   i++;
}
i += 3;
while( !(BufferRX[i]=='*' && BufferRX[i+1]=='*') ){
   Serial.print(BufferRX[i]);
   i++;
}
Serial.print(BufferRX[i]);
Serial.print(BufferRX[i+1]);
Serial.print(BufferRX[i+2]);

Serial.println(" HTTP/1.1");
Serial.print("Host: ");
Serial.println(address);
Serial.println();
```

Per indicare la fine dei dati da trasmettere, il modulo prevede l'invio della seguente stringa:

```
Serial.print("--EOF--Pattern--");
```

Si attendono le due risposte del modulo GPRS, che saranno nell'ordine prima "OK" per comunicare il successo dell'operazione, ed in seguito una notifica "+KTCPDATA: <byte ricevuti>" che indica che il server remoto ha risposto al nostro messaggio HTTP e la risposta, di dimensione <byte ricevuti>, è disponibile per essere letta [7].

```
            module.WaitResp(2000);

            module.WaitResp(2000);

        }
```

Finito l'invio dei dati, azzero il buffer di ricezione I2C e il contatore dei byte ricevuti. Quindi esco dallo stato di SEND ponendo a *false* il rispettivo *flag*.

```
        Count=0;

        memset(BufferRX,0,SIZE);

        Sending=false;
```

Durante le fasi di test, è emerso che il modulo GPRS a disposizione chiude automaticamente la socket TCP dopo averla utilizzata, attraverso il comando KTCPSND, per sedici volte. Tale comportamento del modulo, non menzionato nella relativa documentazione, viene corretto attraverso il conteggio degli invii effettuati; quando si raggiunge la soglia di 16 messaggi, la connessione viene chiusa e quindi riaperta.

```
        sent++;

        if(sent>15){

            sent=0;

            module.TCPDisconnect();

            module.SetupGPRS(APN, "", "");

            module.TCPConnect(address, port);

          }

    }

}
```

3.2.4 - Handler di Ricezione

Questa funzione viene chiamata ad ogni byte ricevuto via I2C ed è la stessa del caso Ethernet, per cui si rimanda al capitolo 2.1.4.

3.3 - Implementazione (versione con metodo POST)

Analogamente al caso Ethernet, l'unica differenza fra l'invio dei dati con metodo POST oppure GET risiede nella struttura del pacchetto HTTP che il nodo Slave deve creare. In particolare, il pacchetto inviato viene costituito nel seguente modo:

```
Serial.print("POST ");

Serial.print(resource);

Serial.print(" HTTP/1.1");

Serial.print(13,BYTE);//\r

Serial.print(10,BYTE);//\n

Serial.print("Host: ");

Serial.print(address);

Serial.print(13,BYTE);

Serial.print(10,BYTE);

Serial.print("Content-Type: application/x-www-form-

             urlencoded");

Serial.print(13,BYTE);

Serial.print(10,BYTE);

Serial.print("Content-Length: ");

Serial.print(bufDim());
```

```
        Serial.print(13,BYTE);

        Serial.print(10,BYTE);

        Serial.print(13,BYTE);

        Serial.print(10,BYTE);

        Serial.print("obj=!!,");
```

Ed in seguito i dati ricevuti via I2C stampati con l'algoritmo già più volte mostrato.

4 - Web Application

Si è scelto di lavorare in ambiente LAMP, ovvero:

- **L**inux come sistema operativo, date le sue caratteristiche di sicurezza maggiormente sviluppate
- **A**pache [16] come web server, in assoluto il più diffuso su Internet [17]
- **M**ySQL [18] come DBMS (database management system)
- **P**hp [19] come linguaggio di programmazione per la creazione di pagine dinamiche e di applicazioni *server-side*

Tutte queste tecnologie sono estremamente diffuse e sono inoltre open source, il che garantisce i più alti livelli di prestazioni, affidabilità, stabilità e facilità di risoluzione dei problemi.

Il primo passo è stato la progettazione del database. La prima versione prevedeva un database con una tabella per ogni parametro, la quale avrebbe contenuto in ogni riga i venti valori più i dati GPS. Questa soluzione è stata quindi abbandonata in quanto poco scalabile (aggiungere un parametro comportava modificare pesantemente la struttura del database) e poco performante in fase di esecuzione delle *query*. Si è quindi optato per un database avente le seguenti tabelle:

- data: ogni riga rappresenta i 20 valori provenienti dal CAN del veicolo; vi è associata una chiave primaria ID e due chiavi esterne ID_GPS e ID_PARAM che indicano rispettivamente i relativi dati GPS e il tipo di parametro cui sono riferiti.
- gps: ogni riga rappresenta un set di dati GPS, composti da ora, data, latitudine, longitudine, altitudine, velocità e numero di satelliti agganciati identificati da una chiave primaria ID e riferiti ad una opportuna *entry* della tabella data
- parameters: ogni riga rappresenta una tipologia di parametro, è identificata da una chiave primaria ID e viene associata ad ogni *entry* della tabella data

Nella successiva figura, è riportata una rappresentazione grafica oltre che ai tipi di dato di ogni record.

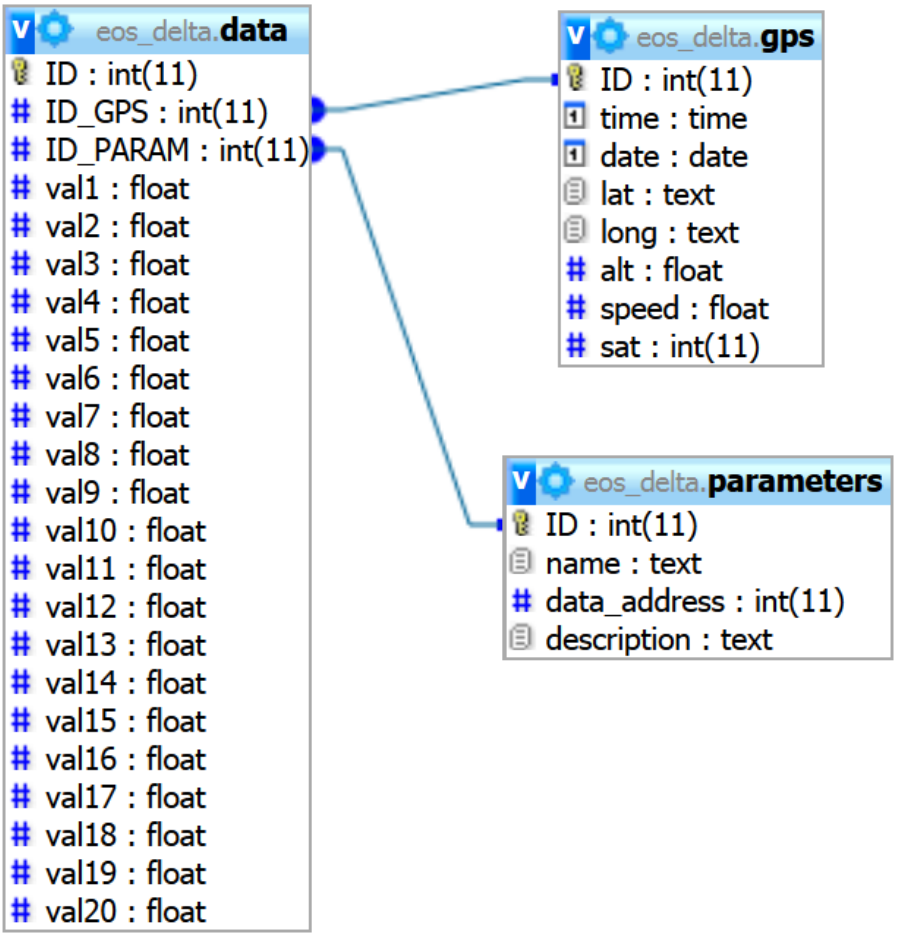


Fig. 11 - Rappresentazione grafica del database

Si è quindi passati alla parte in Php. La web application ha due compiti principali: la ricezione e salvataggio delle stringhe inviate dal nodo "Slave" e la consultazione dei dati. Essa è costituita da otto file che verranno descritti in seguito (saranno descritte riga per riga solo le parti ritenute più rilevanti; il restante sarà riportato in Appendice A):

- config.php
- save.php
- index.php
- createDB.php
- view.php
- viewAll.php
- saveTxt.php
- delete.php

Il funzionamento è il seguente: il nodo Slave avvia lo script save.php passando, alternativamente con il metodo GET o con il metodo POST, la stringa ricevuta dal nodo

Master. Lo script quindi provvede ad estrarre da tale stringa i 20 valori e i dati GPS e a immagazzinarli nel database. La consultazione di tali dati avviene dalla *home page* costituita dal file index.php, che permette di visualizzare in formato tabellare tutti dati ricevuti o relativamente ad un solo parametro o a tutti i parametri. Dalla pagina di visualizzazione è possibile effettuare due operazioni: salvare i dati in formato testo oppure cancellare i dati.

4.1 - Implementazione

Come detto quindi, la *web application* si compone di una parte per il salvataggio dei dati e di una parte per la consultazione. Il file config.php contiene alcuni parametri generali necessari all'interfacciamento con il database:

```
<?php

// file di configurazione con i necessari parametri MySQL

$ip="127.0.0.1";

$sql_user="root";

$sql_password="";

$database_name="eos_delta";

?>
```

I restanti file realizzano le funzionalità richieste.

4.1.1 - Salvataggio dei dati

Lo script Php che si occupa del salvataggio dati è il file save.php; esso riceve in ingresso la stringa e a seconda del risultato dell'operazione restituisce un solo byte (per minimizzare i dati da manipolare da parte del nodo Arduino) avente valore:

- 0: salvataggio correttamente avvenuto
- 1: errore database

- 2: ricevuta stringa errata
- 3: bad data address

La prima parte dello script è occupata da una serie di definizioni di costanti, utilizzate in seguito:

```
<?php

require("config.php");
```

Prima si definiscono i *data* address:

```
define("MOTORCURRENT",10180);

define("BATTERYCURRENT",10301);

define("MOTORWINDINGTEMPERATURE",13101);

define("DCBUSVOLTAGE",33631);

define("CHARGELEVEL",32791);

define("CHARGECURRENT",34901);

define("NEQUALIZINGCELLS",34231);

define("MOTORSPEED",11600);

define("INVERTERTEMPERATURE",13051);

define("AUXBATTERYVOLTAGE",10031);

define("MAXBATTERYTEMPERATURE",13151);

define("LOWERCELLVOLTAGE",33621);

define("HIGHERCELLVOLTAGE",33611);
```

Quindi i fattori di scala in accordo con l'unità di misura:

```
define("MOTORCURRENT_SF",0.1);                //[A_RMS]

define("BATTERYCURRENT_SF",0.1);              //[A]

define("MOTORWINDINGTEMPERATURE_SF",0.01);    //[°C]

define("DCBUSVOLTAGE_SF",0.01);               //[V]
```

```
define("CHARGELEVEL_SF",0.01);                //[%]

define("CHARGECURRENT_SF",0.1);               //[A]

define("NEQUALIZINGCELLS_SF",1);

define("MOTORSPEED_SF",0.001);                //[rpm]

define("INVERTERTEMPERATURE_SF",0.01);        //[°C]

define("AUXBATTERYVOLTAGE_SF",0.1);           //[°V]

define("MAXBATTERYTEMPERATURE_SF",0.01);      //[°C]

define("LOWERCELLVOLTAGE_SF",0.001);          //[°V]

define("HIGHERCELLVOLTAGE_SF",0.001);         //[°V]
```

Vengono quindi definite tre funzioni aventi il compito di ricreare il dato nel formato opportuno (int, unsigned int o long) a partire dai singoli byte che lo compongono. Si noti che gli interi con segno su Arduino vengono rappresentati con 16 bit mediante complemento a due, mentre su PC vengono usati 32 bit. Quindi è necessario estendere il segno per i numeri negativi (quelli il cui bit più significativo è uno, per i quali è necessario aggiungere in testa altri 16 uni):

```
function toInt($msb, $lsb){

   $result=($msb << 8)|$lsb;

   $sign = $result >> 15;

   if($sign == 1)

      $result = $result | 0xffff0000; //estensione del segno

   return $result;

}


function toUInt($msb, $lsb){

   return (($msb << 8)|$lsb);

}
```

```
function toLong($msbh, $msbl, $lsbh, $lsbl){
   return (($msbh << 24)|($msbl << 16)|($lsbh << 8)|$lsbl);
}
```

Il nodo Slave, sia nel caso di utilizzo di GET che di POST, passa allo script save.php un parametro solo, chiamato obj. La prima operazione che svolge quindi lo script è quella di recuperare la stringa e di salvarla in una variabile locale:

```
if(isset($_GET["obj"]))
   $obj=$_GET["obj"];
elseif(isset($_POST["obj"]))
   $obj=$_POST["obj"];
else
   die("2");
```

La stringa ricevuta è composta da byte separati da virgole; viene quindi creato un array i cui elementi corrispondono ai byte della stringa, ovvero, ricordando la struttura della stringa relativa ai dati da due byte:

```
stringa                   !!,DATA_TYPE_MSB,DATA_TYPE_LSB,
formato                start,                          ,
indice array              0,<-----1----->,<-----2----->,

stringa (continua)        val1_MSB,val1_LSB,val2_MSB,val2_LSB,[...],
formato (continua)        <-----dati da canbus, 20 x 2 = 40B------>,
indice array (continua)   <--3--->,<--4--->,<--5--->,<--6--->,     ,

stringa (continua)        152532,4411.9724,N,01202.0435,E,10,125.7,0.25,140911,**
formato (continua)        HHMMSS,<lat......>,<long......>,s#,speed,alti,DDMMYY,end
indice array (continua)   <-43->,<-44 e 45->,<-46 e 47-->,48,<49->,<50>,<-51->,<52>
```

Nel caso del parametro Motor Speed (4 byte) si avranno 80 byte relativi ai valori misurati, di conseguenza gli indici dei dati GPS partiranno da 83 per finire a 92 (incrementati tutti di un *offset* di 40 posizioni).

Nel caso di due virgole consecutive (ovvero di elemento nullo) l'elemento corrispondente dell'array avrà valore NULL.

```
$param = explode(",", $obj);
```

Dato che il numero di elementi delle stringhe è fisso (53 elementi nel caso di dati a 2B, 93 ovvero 53+40 nel caso di dati a 4B), posso controllare la correttezza della stringa ricevuta conteggiando gli elementi dell'array:

```
if((sizeof($param)!=53)&&(sizeof($param)!=93))

   die("2");
```

Quindi si salvano in variabili locali i dati GPS correttamente formattati prelevati dall'array secondo lo schema degli indici sopra riportato; se si sta trattando la stringa del parametro Motor Speed, gli indici devono essere aumentati di un *offset* di 40 come già evidenziato:

```
$offset=0;

if(toUInt($param[1],$param[2])==MOTORSPEED)

   $offset=40;



$time="'".$param[43+$offset]."'";
```

Il GPS usa DDMMYY come formato per la data, mentre MySQL richiede YYMMDD per cui la data deve essere riordinata:

```
$date="'".substr($param[51+$offset],4,2).substr($param[51+$offset
],2,2).substr($param[51+$offset],0,2)."'";
```

Quindi si passa alle coordinate GPS, se presenti; se non presenti, la funzione explode() restituisce NULL. In questo caso si è scelto di sostituire il valore vuoto con una stringa "NA", *not available.*

```
if($param[44+$offset]!=NULL)

   $lat="'".substr($param[44+$offset],0,2)."
```

```
        Deg.".substr($param[44+$offset],2,6)."
        Pr.".$param[45+$offset]."'";
else
   $lat="'NA'";

if($param[46+$offset]!=NULL)
   $long="'".substr($param[46+$offset],0,3)." Deg.
         ".substr($param[46+$offset],3,6)." Pr.
         ".$param[47+$offset]."'";
else
   $long="'NA'";

$alt="'".$param[50+$offset]."'";
$speed="'".($param[49+$offset])."'";
$sat="'".$param[48+$offset]."'";
```

Effettuata la connessione al database, si inseriscono i dati GPS ricevuti nella tabella gps.

```
$connessione = mysql_connect($ip,$sql_user,$sql_password) or
die("1");
mysql_select_db($database_name, $connessione) or die("1");

$query="INSERT INTO `gps` VALUES ( NULL , ".$time." , ".$date." ,
       ".$lat." , ".$long." , ".$alt." , ".$speed." ,
       ".$sat.");";
mysql_query($query,$connessione) or die("1");
```

Si inizia poi a costruire la query per l'inserimento dei dati CAN: innanzi tutto, si recupera l'ID del record GPS appena inserito perché possa essere usato come chiave esterna (ID_GPS):

```
$res=mysql_query("SELECT `ID` FROM `gps` ORDER BY
                 `ID`",$connessione) or die("1");
$last_row = mysql_num_rows($res) - 1;
mysql_data_seek($res, $last_row);
$row = mysql_fetch_row($res);
$id_gps=$row[0];


mysql_free_result($res);
```

Viene poi controllato il data type (ricomponendo i bue byte che lo compongono nell0'intero senza segno corrispondente, da confrontarsi con i valori definiti con la direttiva *define* a inizio script) e recuperato il corrispondente ID, per usarlo poi come chiave esterna (ID_PARAM):

```
switch (toUInt($param[1],$param[2])){


   case MOTORCURRENT:
      $res=mysql_query("SELECT `ID` FROM `parameters` WHERE
                       `data_address` = ".MOTORCURRENT,
                       $connessione) or die("1");
      $id_param=mysql_result($res,0);
      mysql_free_result($res);
      $sf=MOTORCURRENT_SF;
      break;
```

```
   […analogamente per tutti gli altri parametri…]


   default:

      die("3");


}
```

Ora è possibile inserire i dati nel database; si scorre l'array ricomponendo i byte relativi ai dati CAN in un intero con o senza segno oppure un long, a seconda del tipo di dato, e appendendoli in coda alla query.

```
$query="INSERT INTO `data` VALUES ( NULL , '".$id_gps."' ,

       '".$id_param."'";


if(toUInt($param[1],$param[2])==MOTORSPEED){

   for($i=3;$i<83;$i+=4)

      $query=$query." ,

             '".((toLong($param[$i],$param[$i+1],$param[$i+2],

             $param[$i+3]))*$sf)."'";

}elseif((toUInt($param[1],$param[2])==BATTERYCURRENT) ||
(toUInt($param[1],$param[2])==MOTORWINDINGTEMPERATURE) ||
(toUInt($param[1],$param[2])==INVERTERTEMPERATURE)){

   for($i=3;$i<43;$i+=2)

      $query=$query." ,

             '".(toInt($param[$i],$param[$i+1])*$sf)."'";

}else{

   for($i=3;$i<43;$i+=2)

      $query=$query." ,
```

```
            '".(toUInt($param[$i],$param[$i+1])*$sf)."'";
}
$query=$query." );";

mysql_query($query,$connessione) or die("1");

mysql_close($connessione);
```

Infine si restituisce un byte a indicare il corretto svolgimento dell'operazione richiesta.

```
echo "0";
```

4.1.2 - Consultazione dei dati

La *home page* è costituita da un semplice script contenuto nel file index.php (si veda il codice in Appendice A); esso da la possibilità di creare il database con le tre tabelle, se già non ne esiste uno, e di visualizzare i dati relativi ad ogni singolo parametro oppure tutti i dati disponibili.

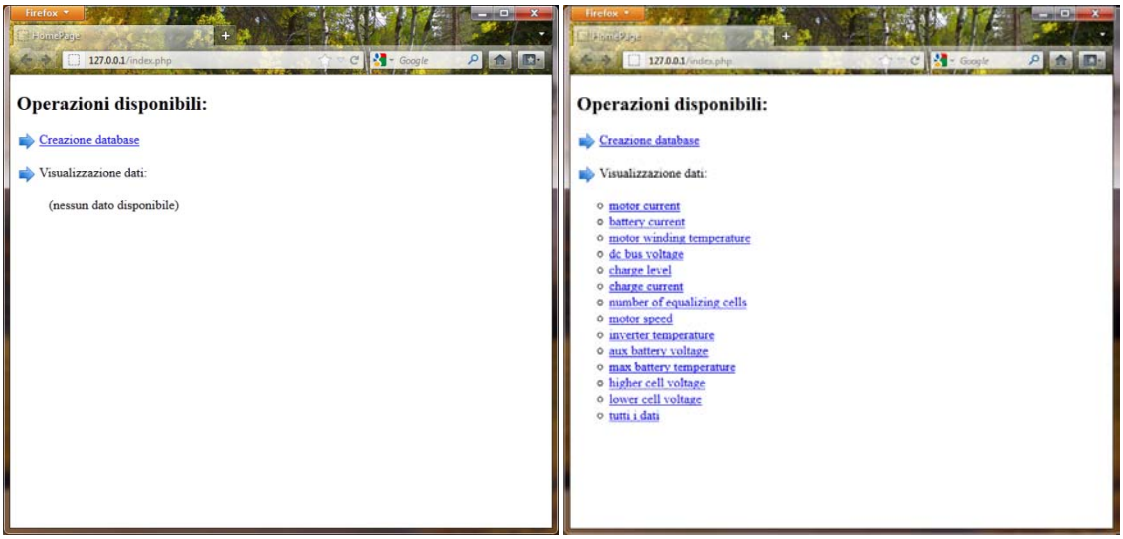
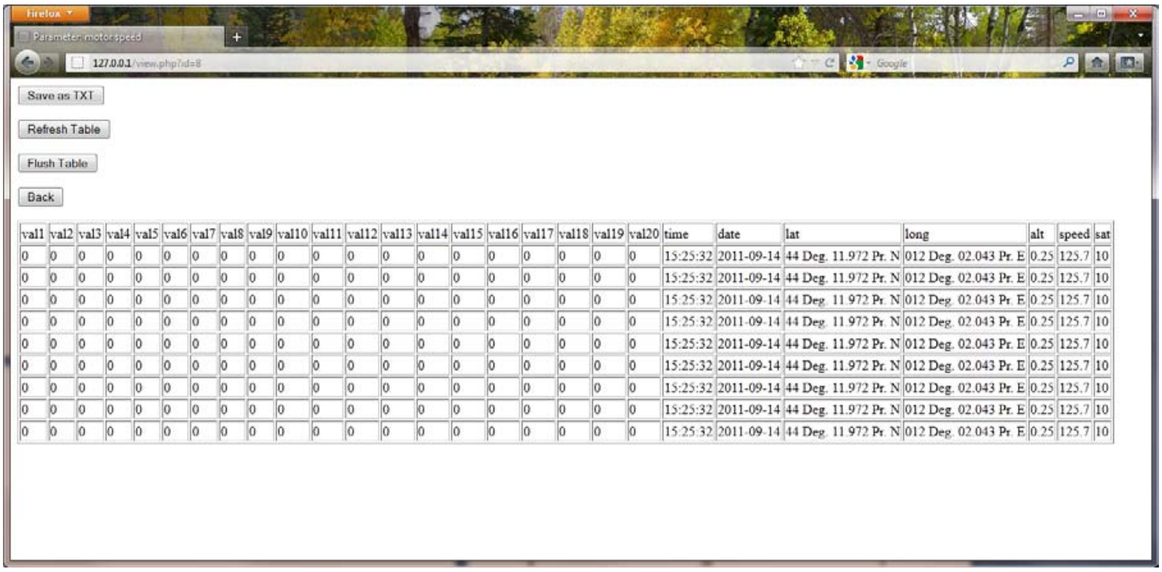


Fig. 12 - Home page prima e dopo la creazione del database

La creazione del database avviene avviando il file createDB.php il quale contiene semplicemente le istruzioni SQL necessarie a implementare la struttura progettata (si veda il codice riportato in Appendice A).

Cliccando sul nome del parametro da visualizzare, viene caricato il file view.php passando come parametro l'ID del parametro da visualizzare (attraverso il metodo GET, quindi con view.php?id=…).



val1	val2	val3	val4	val5	val6	val7	val8	val9	val10	val11	val12	val13	val14	val15	val16	val17	val18	val19	val20	time	date	lat	long	alt	speed	sat
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15:25:32	2011-09-14	44 Deg. 11.972 Pr. N	012 Deg. 02.043 Pr. E	0.25	125.7	10

Fig. 13 - Pagina di visualizzazione di un solo parametro (in questo caso Motor Speed)

Lo script quindi procede a salvare su una variabile locale l'identificativo del parametro che si vuol visualizzare.

```
<html>

<?php

require("config.php");

$id_param = $_GET["id"];
```

Quindi, crea un pulsante per il salvataggio dei dati in formato testo, eseguendo il file saveTxt.php (descritto oltre).

```
echo "<form action=\"saveTxt.php?id=".$id_param."\"

      name=\"frm1\" method=\"post\">\n";

echo "<input type=\"submit\" name=\"save\" value=\"Save as
```

```
     TXT\">\n";

echo "</form>\n";
```

Si ha poi un pulsante per la cancellazione di tutti i dati relativi al parametro in esame, realizzata attraverso lo script delete.php (descritto in seguito).

```
echo "<form action=\"delete.php?id_param=".$id_param."\"

      name=\"frm3\" method=\"post\">\n";

echo "<input type=\"submit\" name=\"flush\" value=\"Flush

      Table\">\n";

echo "</form>\n";
```

È inoltre previsto un pulsante per il refresh della pagina, per visualizzare i nuovi dati pervenuti, e uno per tornare alla *home page*:

```
echo "<form name=\"frm2\" method=\"post\">\n";

echo "<input type=\"button\" value=\"Refresh Table\"

      onClick=\"window.location.reload()\">";

echo "</form>\n";


echo "<form action=\"index.php\" name=\"frm4\"

      method=\"post\">\n";

echo "<input type=\"submit\" name=\"back\" value=\"Back\">\n";

echo "</form>\n";
```

Effettuata la connessione al database, si imposta il titolo della pagina in “Parameter: <nome_parametro>”; per far ciò si recupera dalla tabella data il nome del parametro corrispondente all’identificativo che era stato precedentemente salvato:

```
$connessione = mysql_connect($ip, $sql_user, $sql_password) or
die("connessione non riuscita: ".mysql_error());

mysql_select_db($database_name, $connessione) or die("errore
nella selezione del database");
```

```
$result = mysql_query("SELECT `description` FROM `parameters`

WHERE `ID` = ".$id_param) or die("errore query");

$name=mysql_fetch_row($result);

mysql_free_result($result);

echo "<title>Parameter: ".$name[0]."</title></head><body>";
```

Quindi vengono selezionati tutti i record relativi al parametro richiesto, nonché tutti i dati GPS:

```
$data_result = mysql_query("SELECT * FROM `data` WHERE `ID_PARAM`
= ".$id_param) or die("errore query");

gps_result = mysql_query("SELECT * FROM `gps`") or die("errore
query");
```

Quindi si scorre l'array dei risultati per stampare a video una tabella; la prima riga riporta i nomi dei campi (si noti che per la tabella data l'indice parte da 2 per non stampare gli ID, ID_GPS e ID_PARAM, ritenuti non rilevanti per l'utente; analogamente per la tabella gps, dove l'indice parte da 1 per evitare ID):

```
echo "<table border='1'><tr>";



for($i=0; $i<mysql_num_fields($data_result); $i++)

{

    $field = mysql_fetch_field($data_result);

    if($i>2) echo "<td>{$field->name}</td>";

}



for($i=0; $i<mysql_num_fields($gps_result); $i++)

{

    $field = mysql_fetch_field($gps_result);
```

```
    if($i>0) echo "<td>{$field->name}</td>";
}

mysql_free_result($gps_result);

echo "</tr>\n";
```

Quindi si passa alla stampa dei dati veri e propri:

```
while($data_row = mysql_fetch_row($data_result))
{
   echo "<tr>";
   for($i=3; $i<sizeof($data_row); $i++)
      echo "<td>".$data_row[$i]."</td>";

   $gps_result = mysql_query("SELECT * FROM `gps` WHERE `ID` =
               ".$data_row[1]) or die("errore query");
   $gps_row=mysql_fetch_row($gps_result);

   for($i=1; $i<sizeof($gps_row); $i++)
      echo "<td>".$gps_row[$i]."</td>";

   mysql_free_result($gps_result);

   echo "</tr>\n";
```

```
}

echo "</table>";

?>

</body>

</html>
```

La stampa dell'insieme di tutti i dati è effettuata dal file viewAll.php (riportato in Appendice A) ed avviene in modo del tutto analogo a quanto visto per la visualizzazione del singolo parametro; semplicemente, vengono selezionati tutti i record della tabella data invece che solo quelli relativi ad uno specifico ID.

Le funzionalità prima menzionate di salvataggio in formato testo dei dati e di cancellazione vengono svolte rispettivamente dai file saveTxt.php e delete.php. Il primo, riportato in Appendice A, è strutturato in modo identico a view.txt; ciò che cambia è l'aggiunta delle due funzioni header, usate per inviare direttamente al client degli header HTTP: in questo caso, si specifica il *content-type* come testo e il *content-disposition* come allegato, da scaricare.

```
header('Content-type: text/plain');

header('Content-Disposition: attachment; filename=' .

        $name[0].'.txt');
```

Quindi invece di creare la tabella con gli opportuni tag html (<td>, <tr>) vengono usati i caratteri \t (tabulazione) e \n (nuova riga).

La cancellazione è effettuata dal file delete.txt. Se eseguito da view.php, esso riceverà come parametro l'ID_PARAM relativo al tipo di dati da cancellare; saranno quindi prelevati tutti gli ID_GPS dei record aventi ID_PARAM pari a quello voluto; si procederà poi alla cancellazione dei record dalla tabella data e in seguito dalla tabella gps. Se lo script viene eseguito da viewAll invece, saranno semplicemente svuotate entrambe le tabelle con il comando SQL TRUNCATE <table>. Quindi si ritorna alla pagina di partenza (view o viewAll) inviando al client un header HTTP di tipo *Location*.

5 - Risultati delle prove

La fase finale di questo lavoro di tesi ha visto i test del sistema sia in un caso simulato che sul veicolo vero e proprio.

Il nodo Master realizzato invia al nodo Slave una stringa dati ogni 2s; si è quindi voluto verificare innanzi tutto che il nodo Slave avesse tempi di risposta sufficientemente rapidi da processare tutte le stringhe ricevute; inoltre, si è anche messo alla prova l'hardware per caratterizzarne le prestazioni.

La prova simulata aveva quindi un duplice scopo: da un lato, quello di verificare il corretto comportamento del sistema al variare dei valori dei dati in ingresso (e quindi si è cercato di esplorare il maggior numero di ingressi possibili); dall'altro, quello di analizzare le prestazioni del sistema, in termini di massima frequenza di arrivo da I2C delle stringhe senza che vengano scartate (e si è quindi provveduto a far lavorare il sistema con diverse frequenze di arrivo).

Lo scopo della prova sul veicolo, infine, è stato quello di verificare la correttezza del lavoro svolto e di riuscire a visualizzare i dati riguardanti il veicolo durante il suo normale funzionamento, per esempio facendo variare la velocità del motore e osservando il variare dei numeri salvati sul database. In questo caso l'esattezza dei valori acquisiti è stata constatata dal confronto con quelli forniti da un analizzatore apposito connesso anch'esso alla rete CAN, progettato e costruito dai ricercatori del laboratorio DIE.

5.1 - Simulatore

Si è realizzato un simulatore di messaggi CAN. In altre parole, si è creato un nodo Master che simuli la ricezione di messaggi CAN e provveda ad inviarli via I2C al nodo Slave con temporizzazione regolabile, pur non essendo connesso alla rete CAN del veicolo.

Lo *sketch* seguente invia ogni WAIT millisecondi (dove WAIT è una costante definita all'inizio) una stringa riferita al parametro Motor Speed dove tutti i dati provenienti dal CAN sono nulli eccetto il primo che parte, nel primo invio, da un valore pari a 0 per crescere di una unità per ogni invio fino a 255, dopo il quale torna a zero, il tutto fino ad un massimo di LIMIT stringhe (anche questa è definita come costante). Così facendo si testano tutti i possibili valori assumibili da un singolo byte.

```
#include <Wire.h>

#define WAIT 1000

#define LIMIT 1000

int sent=0;

byte dummy=0;


void setup(){

  Wire.begin();

  Serial.begin(9600);

}


void loop(){

  byte buf[]=

{'!','!',',',45,',',80,',',0,',',0,',',0,',',dummy,',',0,',',0,',
',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,'
,',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,
',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0
,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',
0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',
,0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',
',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',',0,',','$','$',
,',1','5','2','5','3','2',',',',4','4','1','1','.','9','7','2','4
',',',',N',',',0','1','2','0','2','.',0','4','3','5',',',',E',',',
,1','0',',',',1','2','5','.','7',',',0','.','2','5',',',1','4',
0','9','1','1',',',',*','*'};

   for(int k=0;k<sizeof(buf);k++){

    Wire.beginTransmission(1);

    Wire.send(buf[k]);

    Wire.endTransmission();

  }
```

```
  if(dummy<256) dummy++; else dummy=0;



  //il simulatore invia LIMIT pacchetti, quindi si ferma e
accende il led

  sent++;

  if(sent>=LIMIT){

     digitalWrite(13,HIGH);

     while(true);

  }


 delay(WAIT);


}
```

5.2 - Prove del nodo in versione Ethernet

5.2.1 - Prove mediante simulatore

Si è collegato il simulatore al nodo Slave, il quale è stato connesso tramite cavo di rete ad un modem ADSL dotato di connessione a 7Mbps. Attraverso questa connessione, il nodo doveva raggiungere un web server remoto. Si sono quindi provati vari valori per il tempo che il simulatore fa intercorrere fra l'invio di due stringhe. In particolare, si sono usati i seguenti valori: 10s, 8s, 6s, 4s, 2s, 1s, 0.8s, 0.6s, 0.4s, 0.2s. Il simulatore è stato configurato per inviare messaggi per un'ora di tempo o mille stringhe al massimo (vale la condizione più restrittiva). Obiettivo della simulazione era indagare la dipendenza dalla frequenza di arrivo delle stringhe dei seguenti parametri, ritenuti indicativi delle prestazioni del nodo:

- Successo percentuale: è il rapporto fra le stringhe inviate dal simulatore e quelle che effettivamente sono giunte a destinazione, ovvero che sono state salvate nel database remoto. Dà un'indicazione di quante stringhe vengono perse a causa dell'eccessiva frequenza di arrivo.
- Stringhe processate al secondo: è il rapporto fra stringhe ricevute dal nodo Slave (che può differire da quelle ricevute dal web server, per esempio nel caso di errori di rete come perdita dei pacchetti o errori di elaborazione della web application) e tempo di simulazione. Dà un'indicazione della velocità di elaborazione del nodo e come questa possa dipendere dalla frequenza di arrivo di nuove stringhe.

I valori ottenuti sono i seguenti:

Tempo [s]	Stringhe Inviate	Tempo di Simulazione [s]	Stringhe Elaborate	Stringhe Ricevute	Successo [%]	Stringhe/s
0,2	1000	200	332	331	33,10	1,66
0,4	1000	400	496	494	49,40	1,24
0,6	1000	600	943	938	93,80	1,57
0,8	1000	800	990	985	98,50	1,24
1	1000	1000	980	975	97,50	0,98
2	1000	2000	999	994	99,40	0,50
4	900	3600	899	894	99,33	0,25
6	600	3600	600	597	99,50	0,17
8	450	3600	450	448	99,56	0,13
10	360	3600	360	358	99,44	0,10

Si sono quindi graficati i valori ottenuti.

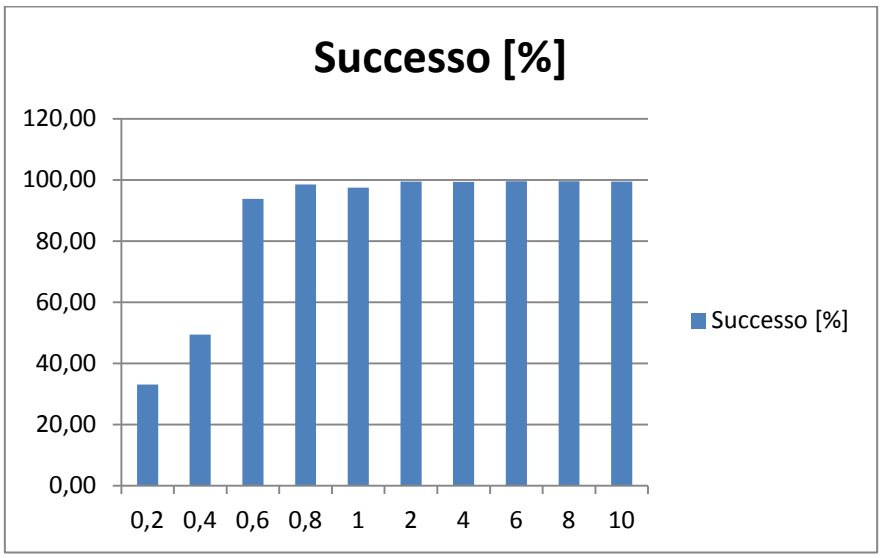


Fig. 14 - Successo percentuale nel caso Ethernet

Come si vede in figura 14, il successo percentuale è praticamente 100% finché il tempo di arrivo è maggiore di 0,8s quindi da 0,6s in poi cala. Questo è legato all'andamento dell'altro parametro analizzato. Infatti dalla figura 15 si vede che la velocità di elaborazione delle stringhe, intesa come stringhe processate al secondo, cresce al crescere della frequenza di arrivo fino a saturare ad un valore intorno a 1,6; ciò significa che il tempo medio di elaborazione è 1 / 1,6 = 0,625s appunto; in altri termini, le stringhe arrivano più in fretta di quanto il nodo riesca ad elaborarle, finendo per perderne alcune. Da segnalare il dato anomalo per 0,4s. Nonostante numerose ripetizioni, si è ottenuto sempre valori inattesi.

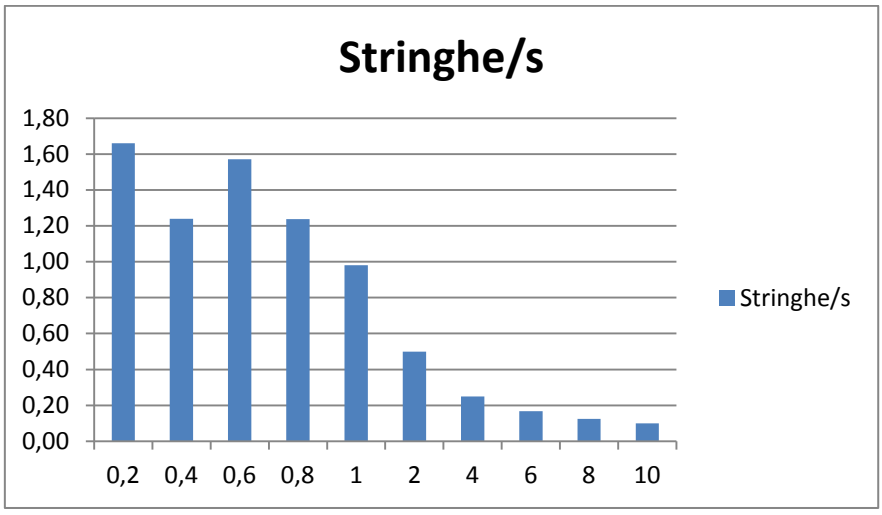


Fig. 15 - Stringhe processate al secondo nel caso Ethernet

Il fatto che nonostante il *flooding* di stringhe il nodo Slave ha continuato a funzionare, semplicemente scartando quanto non riusciva ad elaborare, è segnale della bontà dell'algoritmo a stati finiti implementato.

5.2.2 - Prove sul veicolo

Le prove sul veicolo sono state effettuate presso il laboratorio del DIE della Facoltà di Ingegneria dell'Università di Bologna. Si è quindi collegato il nodo Master, dotato dell'opportuno *shield* CAN, alla centralina del veicolo attraverso un cavo a 9 pin di tipo D-SUB; analogamente, il nodo Slave è stato collegato con cavo di rete direttamente al web server.

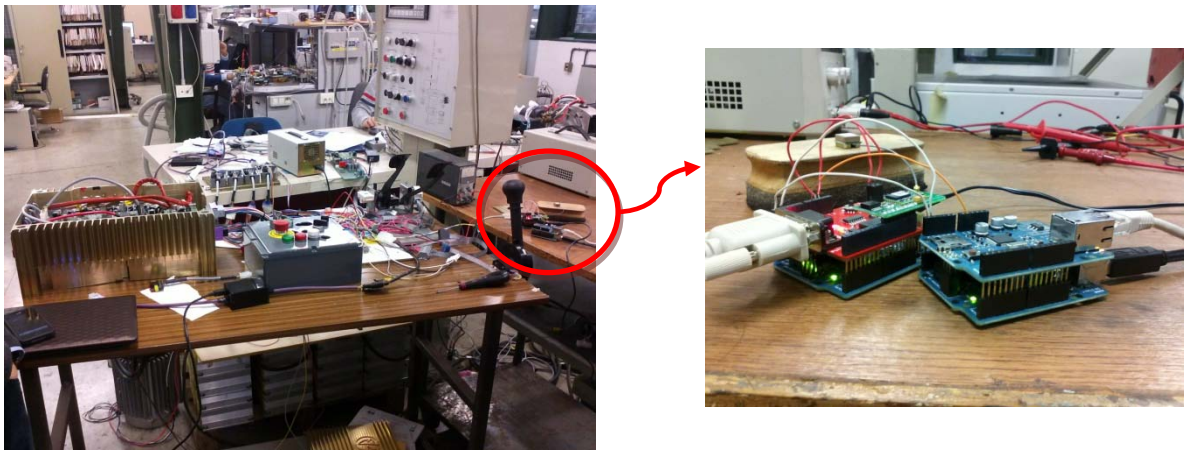
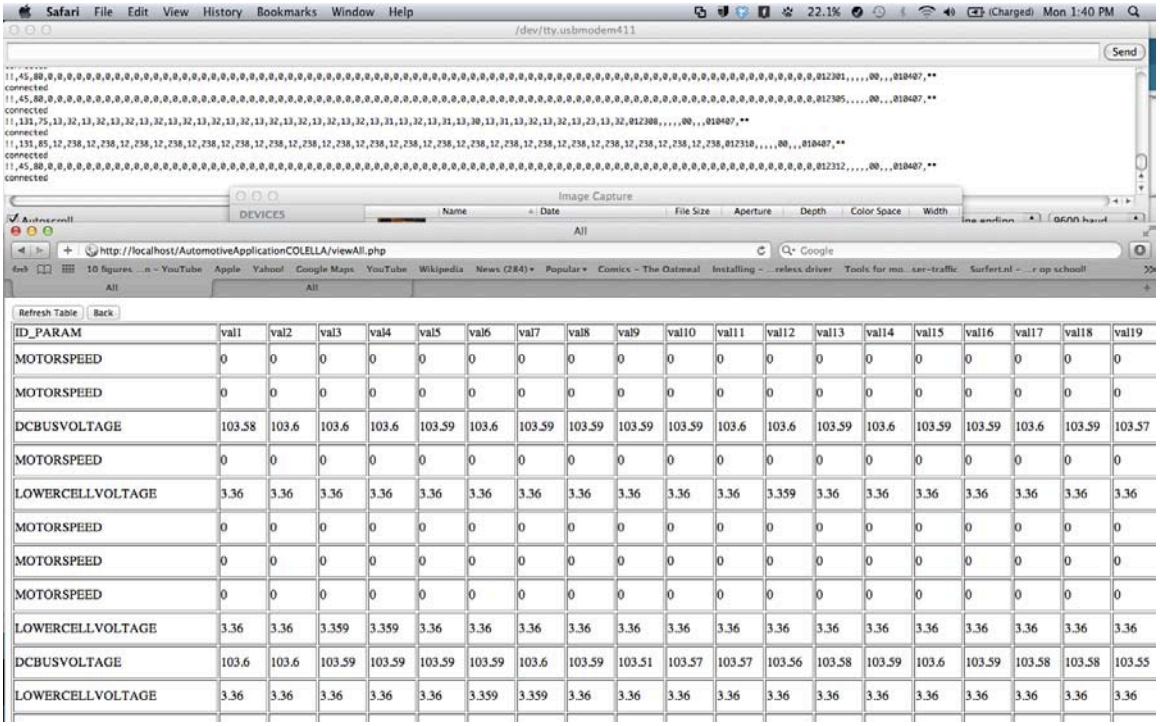


Fig. 15 - A sinistra la parte powertrain del veicolo collegata al sistema di monitoraggio remoto basato su Arduino, ingrandito a destra, dove sono mostrati il nodo Master (shield rosso su board blu) e il nodo Slave (shield blu su board blu).

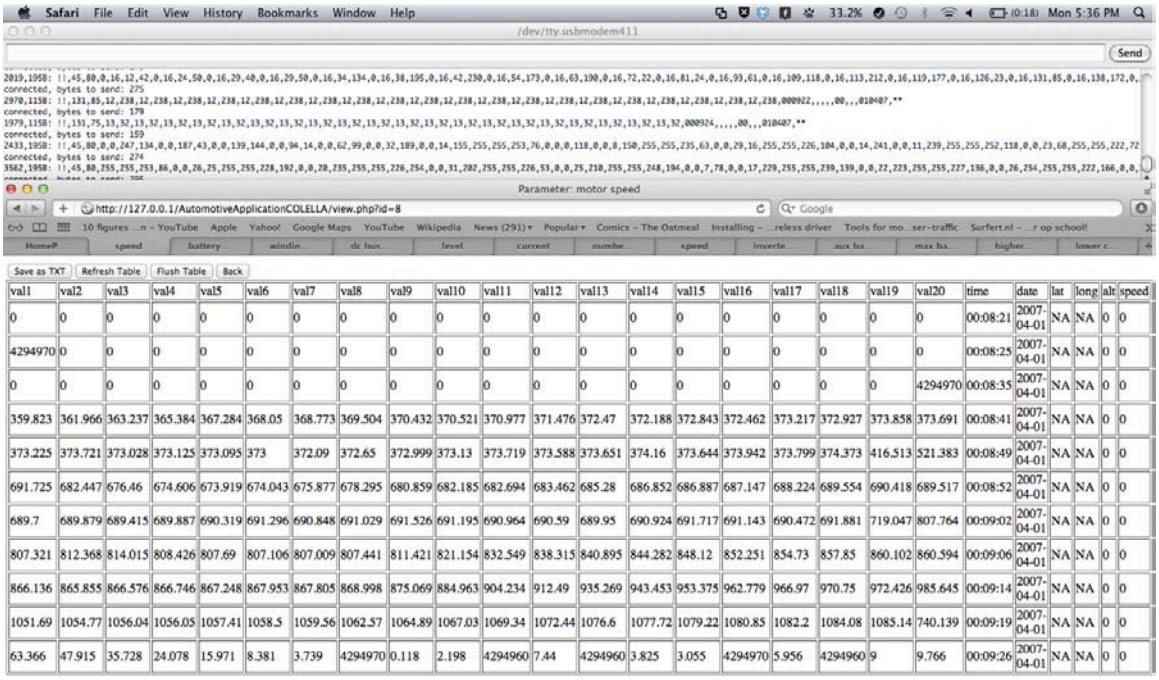
Si è poi avviato il veicolo ed analizzato il contenuto del database. Si sono ottenuti come atteso una serie di dati in accordo con quanto mostrato dall'analizzatore disponibile in laboratorio.



ID_PARAM	val1	val2	val3	val4	val5	val6	val7	val8	val9	val10	val11	val12	val13	val14	val15	val16	val17	val18	val19
MOTORSPEED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOTORSPEED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DCBUSVOLTAGE	103.58	103.6	103.6	103.6	103.59	103.6	103.59	103.59	103.59	103.59	103.6	103.6	103.59	103.6	103.59	103.59	103.6	103.59	103.57
MOTORSPEED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOWERCELLVOLTAGE	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.359	3.36	3.36	3.36	3.36	3.36	3.36	3.36
MOTORSPEED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOTORSPEED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOTORSPEED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LOWERCELLVOLTAGE	3.36	3.36	3.359	3.359	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36
DCBUSVOLTAGE	103.6	103.6	103.59	103.59	103.59	103.59	103.6	103.59	103.51	103.57	103.57	103.56	103.58	103.59	103.6	103.59	103.58	103.58	103.55
LOWERCELLVOLTAGE	3.36	3.36	3.36	3.36	3.36	3.359	3.359	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36	3.36

Fig. 16 - Vista di insieme dei parametri ricavati dal veicolo; nella parte alta, sono mostrate le stampe di debug del nodo Slave su linea seriale.

Si è poi passati ad esaminare l'andamento di un singolo parametro. Per avere un riscontro immediato, si è agito sul pedale del "gas", aumentando quindi la velocità dei giri motore. Come si vede in figura, al passare del tempo si rilevano valori dei giri motori crescenti (corrispondenti al "dare gas") e quindi calanti (corrispondenti con l'arresto del veicolo). Numericamente, i valori coincidevano con quelli dell'analizzatore del laboratorio.



val1	val2	val3	val4	val5	val6	val7	val8	val9	val10	val11	val12	val13	val14	val15	val16	val17	val18	val19	val20	time	date	lat	long	alt	speed
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00:08:21	2007-04-01	NA	NA	0	0
4294970	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00:08:25	2007-04-01	NA	NA	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4294970	00:08:35	2007-04-01	NA	NA	0	0
359.823	361.966	363.237	365.384	367.284	368.05	368.773	369.504	370.432	370.521	370.977	371.476	372.47	372.188	372.843	372.462	373.217	372.927	373.858	373.691	00:08:41	2007-04-01	NA	NA	0	0
373.225	373.721	373.028	373.125	373.095	373	372.09	372.65	372.999	373.13	373.719	373.588	373.651	374.16	373.644	373.942	373.799	374.373	416.513	521.383	00:08:49	2007-04-01	NA	NA	0	0
691.725	682.447	676.46	674.606	673.919	674.043	675.877	678.295	680.859	682.185	682.694	683.462	685.28	686.852	686.887	687.147	688.224	689.554	690.418	689.517	00:08:52	2007-04-01	NA	NA	0	0
689.7	689.879	689.415	689.887	690.319	691.296	690.848	691.029	691.526	691.195	690.964	690.59	689.95	690.924	691.717	691.143	690.472	691.881	719.047	807.764	00:09:02	2007-04-01	NA	NA	0	0
807.321	812.368	814.015	808.426	807.69	807.106	807.009	807.441	811.421	821.154	832.549	838.315	840.895	844.282	848.12	852.251	854.73	857.85	860.102	860.594	00:09:06	2007-04-01	NA	NA	0	0
866.136	865.855	866.576	866.746	867.248	867.953	867.805	868.998	875.069	884.963	904.234	912.49	935.269	943.453	953.375	962.779	966.97	970.75	972.426	985.645	00:09:14	2007-04-01	NA	NA	0	0
1051.69	1054.77	1056.04	1056.05	1057.41	1058.5	1059.56	1062.57	1064.89	1067.03	1069.34	1072.44	1076.6	1077.72	1079.22	1080.85	1082.2	1084.08	1085.14	740.139	00:09:19	2007-04-01	NA	NA	0	0
63.366	47.915	35.728	24.078	15.971	8.381	3.739	4294970	0.118	2.198	4294960	7.44	4294960	3.825	3.055	4294970	5.956	4294960	9	9.766	00:09:26	2007-04-01	NA	NA	0	0

Fig. 17 - Vista dei dati relativi alla velocità del motore; nella parte alta, sono mostrate le stampe di debug del nodo Slave su linea seriale.

5.3 - Prove del nodo in versione GPRS

5.3.1 - Prove mediante simulatore

Si è collegato il nodo Slave dotato dello *shield* GPRS al simulatore; si sono quindi svolte le stesse prove svolte nel caso Ethernet, ottenendo i seguenti risultati:

Tempo [s]	Stringhe Inviate	Tempo di Simulazione [s]	Stringhe Elaborate	Stringhe Ricevute	Successo [%]	Stringhe/s
0,2	1000	200	57	55	5,50	0,29
0,4	1000	400	102	101	10,10	0,26
0,6	1000	600	171	169	16,90	0,29
0,8	1000	800	214	210	21,00	0,27
1	1000	1000	275	273	27,30	0,28
2	1000	2000	449	445	44,50	0,22
4	900	3600	714	666	74,00	0,20
6	600	3600	553	545	90,83	0,15

8	450	3600	445	443	98,44	0,12
10	360	3600	353	351	97,50	0,10

Graficando i due parametri di interesse si ottiene:

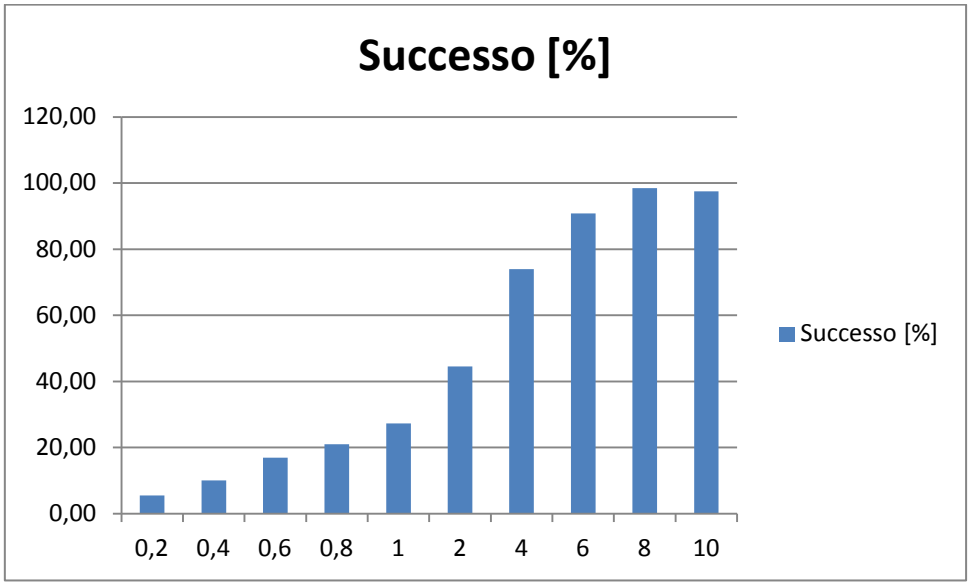


Fig. 18 - Successo percentuale nel caso GPRS

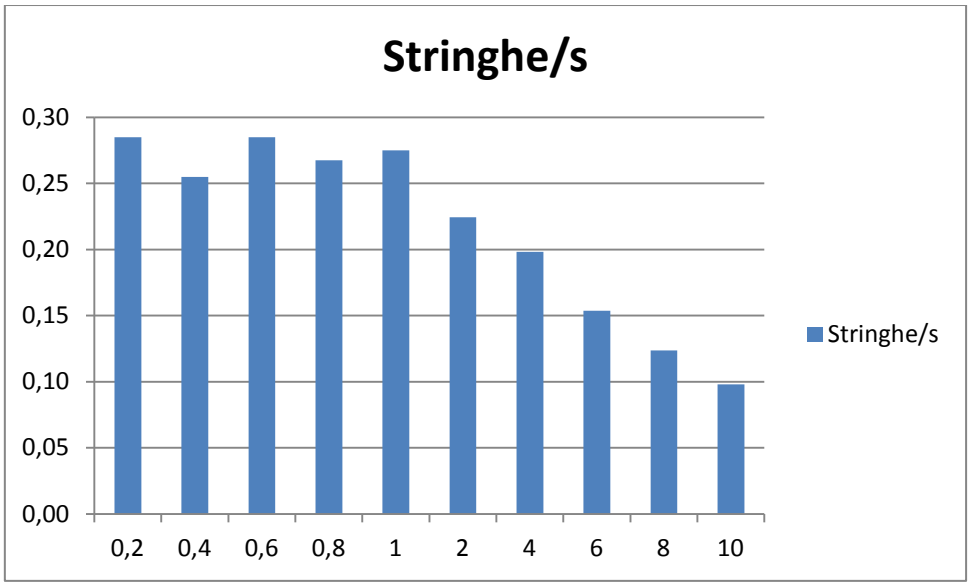


Fig. 19 - Stringhe processate al secondo nel caso GPRS

Si ottengono risultati del tutto simili al caso Ethernet, tuttavia prestazionalmente di molto inferiori. In questo caso, il numero di stringhe processate al secondo satura ad un valore di circa 0,29 (anche qui si ha un valore anomalo per 0,4s), quindi si ha un tempo di elaborazione medio di circa 1 / 0,29 ≈ 3,44s. Questo si riflette nel fatto che da 4s in poi il nodo inizia a perdere un gran numero di stringhe.

Se da un lato è un buon risultato, perché garantisce la funzionalità del nodo anche in condizioni operative sfavorevoli, dall'altro il fatto che per un tempo di interarrivo di 2s la perdita sia di oltre il 50% suggerisce che si debba accettare un *trade-off* fra l'esigenza di mobilità, che solo il nodo in versione GPRS può garantire, e i requisiti prestazionali.

Conclusioni

L'obiettivo di questo lavoro di Tesi Magistrale era dimostrare che anche con un microcontrollore a basso costo e a limitate prestazioni fosse possibile sviluppare un sistema di monitoraggio da remoto, via Internet, dei valori di esercizio di un autoveicolo elettrico.

Nel corso del progetto è stato realizzato il sistema controllore desiderato, costituito da due nodi e da un web server remoto. Il primo nodo, denominato Master, ha il compito di collegarsi alla centralina dell'automobile e prelevare i dati messi a disposizione attraverso la rete CAN. L'altro nodo, denominato Slave, comunica con esso attraverso bus I2C, ed invia i dati ricevuti ad una web application situata su un server remoto.

Il nodo Slave è stato realizzato in due versioni: una cablata, utilizzante la tecnologia Ethernet per la connessione alla rete, ed un'altra senza fili, utilizzante la tecnologia GPRS. Nella progettazione del nodo Slave è risultata vincente la scelta di affidarsi a tecnologie open source, il che ha permesso il riuso di parte del software appartenente ad altri progetti, semplificando e velocizzando così l'intero processo. Ciò ha permesso inoltre di realizzare una libreria software per la gestione del modulo GPRS ricca di funzioni, alcune delle quali non direttamente necessarie al raggiungimento dello scopo prefisso, ma tuttavia utili per progetti futuri.

La web application è stata anch'essa realizzata totalmente con tecnologie open source. È composta da due parti concettualmente distinte, una dedicata alla ricezione dei dati e al loro inserimento nel database, ed un'altra dedicata alla consultazione di questi dati. Questa seconda parte ha, al momento, grandi margini di miglioramento, dato che l'aspetto grafico è scarno; è possibile infatti elaborare i dati ricevuti direttamente *server-side* per mostrare grafici, statistiche ed i dati stessi in modo più chiaro ed accattivante.

Successivamente alla fase realizzativa, grande importanza è stata data alla verifica delle funzionalità e delle prestazioni del sistema. Sono stati realizzati numerosi test per validare il nodo Slave in entrambe le versioni. Si è mostrato quindi che la versione Ethernet ha elevate prestazioni in termini di dati elaborati nell'unità di tempo, di molto superiori a quanto richieste per il normale funzionamento del sistema; la versione GPRS invece richiede di accettare un *trade-off* fra prestazioni e mobilità.

Il lavoro svolto ha ulteriori sviluppi futuri: a partire dagli schematici delle *board* e degli *shield* utilizzati, sarà realizzata un'unica scheda che verrà installata a bordo del veicolo; inoltre, essendo il progetto Arduino in continua evoluzione, si dovrà tenere conto delle

nuove funzionalità introdotte dagli aggiornamenti software e delle nuove potenzialità dell'hardware; ciò farà sì che, pur mantenendo contenuti i costi, si avrà la possibilità, ad esempio, di integrare i due nodi in uno unico o di aggiungere più funzioni ai nodi stessi. Si è infatti prospettata, ad esempio, la possibilità di salvare eventuali file di log di sistema su scheda SD o quella di implementare la comunicazione in entrambi i sensi, ossia dare la possibilità all'utente remoto non solo di consultare i valori di alcuni parametri della centralina del veicolo, ma anche di poterne modificare alcuni, nel rispetto ovviamente della sicurezza che tale operazione richiede.

Appendice A - Codice

File **index.php**:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-
8" />

<title>HomePage</title>

</head>

<body>

<h2>Operazioni disponibili:</h2>

<p><img src="arrow.png" width="24" height="24" align="texttop" />
<a href="createDB.php">Creazione database</a></p>

<p><img src="arrow.png" alt="" width="24" height="24"
align="texttop" /> Visualizzazione dati:

<ul type="circle">

<?php

require("config.php");

$connessione = mysql_connect($ip,$sql_user,$sql_password) or
die("connessione non riuscita: ".mysql_error());

mysql_select_db($database_name, $connessione) or die("(nessun
dato disponibile)");



$result=mysql_query("SELECT * FROM `parameters` ORDER BY
`ID`",$connessione) or die("errore nella lettura dela tabella");
```

```
//per ogni parametro presente nella tabella parameters, crea un
link

while($row = mysql_fetch_row($result))

echo "<li><a
href='view.php?id=".$row[0]."'>".$row[3]."</a></li>";

mysql_free_result($result);

mysql_close($connessione);

echo "<li><a href='viewAll.php'>tutti i dati</a></li>";

?>

</ul>

</p>

</body>

</html>
```

File **createDB.php**:

```
<html>

<head>

<title>Database Creation</title>

</head>

<body>

<form action="index.php" name="frm" method="post">

<input type="submit" name="back" value="Back">

</form>



<?php

require("config.php");
```

```
$connessione = mysql_connect($ip,$sql_user,$sql_password) or
die("connessione non riuscita: ".mysql_error());

//creazione db

mysql_query("CREATE DATABASE `".$database_name."`;",$connessione)
or die("errore nella creazione del database");

mysql_select_db($database_name, $connessione) or die("errore
nella selezione del database");

//creazione tabella data

mysql_query("CREATE TABLE IF NOT EXISTS `data` ( `ID` int(11) NOT
NULL AUTO_INCREMENT,`ID_GPS` int(11) NOT NULL, `ID_PARAM` int(11)
NOT NULL, `val1` float NULL,`val2` float NULL, `val3` float NULL,
`val4` float NULL, `val5` float NULL, `val6` float NULL, `val7`
float NULL, `val8` float NULL, `val9` float NULL, `val10` float
NULL, `val11` float NULL, `val12` float NULL, `val13` float NULL,
`val14` float NULL, `val15` float NULL, `val16` float NULL,
`val17` float NULL, `val18` float NULL, `val19` float NULL,
`val20` float NULL, PRIMARY KEY (`ID`), KEY `ID_GPS` (`ID_GPS`),
KEY `ID_PARAM` (`ID_PARAM`), KEY `ID` (`ID`) ) ENGINE=InnoDB
DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;",$connessione) or
die("errore nella creazione della tabella data");

//creazione tabella gps

mysql_query("CREATE TABLE IF NOT EXISTS `gps` ( `ID` int(11) NOT
NULL AUTO_INCREMENT, `time` time NULL, `date` date NULL, `lat`
text NULL, `long` text NULL, `alt` float NULL, `speed` float
NULL, `sat` int(11) NULL, PRIMARY KEY (`ID`), KEY `ID` (`ID`) )
ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1
;",$connessione) or die("errore nella creazione della tabella
gps");

//creazione tabella parameters

mysql_query("CREATE TABLE IF NOT EXISTS `parameters` ( `ID`
int(11) NOT NULL AUTO_INCREMENT, `name` text NULL, `data_address`
int(11) NULL, `description` text NULL, PRIMARY KEY (`ID`), KEY
`ID` (`ID`) ) ENGINE=InnoDB DEFAULT CHARSET=latin1
AUTO_INCREMENT=1 ;",$connessione) or die("errore nella creazione
della tabella parameters");



//creazione foreign keys
```

```
mysql_query("ALTER TABLE `data` ADD CONSTRAINT `data_ibfk_1`
FOREIGN KEY (`ID_GPS`) REFERENCES `gps` (`ID`), ADD CONSTRAINT
`data_ibfk_2` FOREIGN KEY (`ID_PARAM`) REFERENCES `parameters`
(`ID`);",$connessione) or die("errore nella creazione delle
foreign keys");



//inserimento parametri

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'MOTORCURRENT',10180,'motor current');",$connessione) or
die("errore nell'aggiunta dei parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'BATTERYCURRENT',10301,'battery current');",$connessione)
or die("errore nell'aggiunta dei parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'MOTORWINDINGTEMPERATURE',13101,'motor winding
temperature');",$connessione) or die("errore nell'aggiunta dei
parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'DCBUSVOLTAGE',33631,'dc bus voltage');",$connessione) or
die("errore nell'aggiunta dei parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'CHARGELEVEL',32791,'charge level');",$connessione) or
die("errore nell'aggiunta dei parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'CHARGECURRENT',34901,'charge current');",$connessione) or
die("errore nell'aggiunta dei parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'NEQUALIZINGCELLS',34231,'number of equalizing
cells');",$connessione) or die("errore nell'aggiunta dei
parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'MOTORSPEED',11600,'motor speed');",$connessione) or
die("errore nell'aggiunta dei parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'INVERTERTEMPERATURE',13051,'inverter
temperature');",$connessione) or die("errore nell'aggiunta dei
parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'AUXBATTERYVOLTAGE',10031,'aux battery
```

```
voltage');",$connessione) or die("errore nell'aggiunta dei
parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'MAXBATTERYTEMPERATURE',13151,'max battery
temperature');",$connessione) or die("errore nell'aggiunta dei
parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'HIGHERCELLVOLTAGE',33621,'higher cell
voltage');",$connessione) or die("errore nell'aggiunta dei
parametri");

mysql_query("INSERT INTO `parameters` VALUES
(NULL,'LOWERCELLVOLTAGE',33611,'lower cell
voltage');",$connessione) or die("errore nell'aggiunta dei
parametri");

mysql_close($connessione);

echo "database e tabelle correttamente creati";

?>

</body>

</html>
```

File **viewAll.php**:

```
<html>

<head>

<title>All</title>

</head>

<body>



<?php



   require("config.php");
```

```
   //pulsante per il refresh della tabella

   echo "<form name=\"frm1\" method=\"post\">\n";

   echo "<input type=\"button\" value=\"Refresh Table\"
onClick=\"window.location.reload()\">";

   echo "</form>\n";



   //pulsante per lo svuotamento della tabella (chiama
delete.php)

   echo "<form action=\"delete.php?id_param=*\" name=\"frm2\"
method=\"post\">\n";

   echo "<input type=\"submit\" name=\"flush\" value=\"Flush
Table\">\n";

   echo "</form>\n";



   //pulsante back

   echo "<form action=\"index.php\" name=\"frm3\"
method=\"post\">\n";

   echo "<input type=\"submit\" name=\"back\" value=\"Back\">\n";

   echo "</form>\n";



   //connessione al database

   $connessione = mysql_connect($ip, $sql_user, $sql_password) or
die("connessione non riuscita: ".mysql_error());

   mysql_select_db($database_name, $connessione) or die("errore
nella selezione del database");



   //seleziono tutti i record
```

```
   $data_result = mysql_query("SELECT * FROM `data`") or
die("errore query");

   $gps_result = mysql_query("SELECT * FROM `gps`") or
die("errore query");



   //stampa la prima riga contenente i nomi dei campi; prima da
data:

   echo "<table border='1'><tr>";



   for($i=0; $i<mysql_num_fields($data_result); $i++)

   {

       $field = mysql_fetch_field($data_result);

       if($i>1) echo "<td>{$field->name}</td>"; //per evitare di
stampare ID, ID_GPS e ID_PARAM

   }



   //quindi da gps:

   for($i=0; $i<mysql_num_fields($gps_result); $i++)

   {

       $field = mysql_fetch_field($gps_result);

       if($i>0) echo "<td>{$field->name}</td>"; //per evitare di
stampare il campo ID

   }



   mysql_free_result($gps_result);



   echo "</tr>\n";
```

```
   //stampa ogni record del database in una riga della tabella

   while($data_row = mysql_fetch_row($data_result))

   {



         echo "<tr>";



         $data_type_res = mysql_query("SELECT * FROM `parameters` 
WHERE `ID` = ".$data_row[2]) or die("errore query");

         $data_type=mysql_fetch_row($data_type_res);



        echo "<td>".$data_type[1]."</td>";



         for($i=3; $i<sizeof($data_row); $i++) //per evitare di 
stampare ID, ID_GPS e ID_PARAM

            echo "<td>".$data_row[$i]."</td>";



         $gps_result = mysql_query("SELECT * FROM `gps` WHERE `ID` 
= ".$data_row[1]) or die("errore query");

         $gps_row=mysql_fetch_row($gps_result);



         for($i=1; $i<sizeof($gps_row); $i++) //per evitare di 
stampare il campo ID

            echo "<td>".$gps_row[$i]."</td>";



         mysql_free_result($gps_result);
```

```
        echo "</tr>\n";


   }


   echo "</table>";


?>


</body>

</html>
```

File **saveTxt.php**:

```
<?php

   require("config.php");

   $id_param = $_GET["id"];

   //connessione al database

   $connessione = mysql_connect($ip, $sql_user, $sql_password) or
die("connessione non riuscita: ".mysql_error());

   mysql_select_db($database_name, $connessione) or die("errore
nella selezione del database");

   $result = mysql_query("SELECT `name` FROM `parameters` WHERE
`ID` = ".$id_param) or die("errore query");

   $name=mysql_fetch_row($result);

   mysql_free_result($result);
```

```
   //indica al browser che il contenuto del file presente è testo, da salvare come allegato

   header('Content-type: text/plain');

   header('Content-Disposition:    attachment;    filename='    . $name[0].'.txt');

   //seleziono tutti i record relativi al parametro richiesto

   $data_result  =  mysql_query("SELECT  *  FROM  `data`  WHERE `ID_PARAM` = ".$id_param) or die("errore query");

   $gps_result  =  mysql_query("SELECT  *  FROM  `gps`")  or die("errore query");

   //stampa la prima riga contenente i nomi dei campi; prima da data:

   for($i=0; $i<mysql_num_fields($data_result); $i++)

   {

       $field = mysql_fetch_field($data_result);

       if($i>2)  echo  ($field->name)."\t";  //per  evitare  di stampare ID, ID_GPS e ID_PARAM

   }

   //quindi da gps:

   for($i=0; $i<mysql_num_fields($gps_result); $i++)

   {

       $field = mysql_fetch_field($gps_result);

       if($i>0)  echo  "{$field->name}\t";  //per  evitare  di stampare il campo ID

   }

   mysql_free_result($gps_result);

   echo "\n";
```

```
   //stampa ogni record del database in una riga della tabella

   while($data_row = mysql_fetch_row($data_result))

   {

       for($i=3;  $i<sizeof($data_row);  $i++)  //per  evitare  di
stampare ID, ID_GPS e ID_PARAM

          echo $data_row[$i]."\t";

        $gps_result = mysql_query("SELECT * FROM `gps` WHERE `ID`
= ".$data_row[1]) or die("errore query");

        $gps_row=mysql_fetch_row($gps_result);

        for($i=1;  $i<sizeof($gps_row);  $i++)  //per  evitare  di
stampare il campo ID

          echo $gps_row[$i]."\t";



        mysql_free_result($gps_result);

        echo "\n";

   }

?>
```

File **delete.php**:

```
<?php


   require("config.php");


   $id_param = $_GET["id_param"];
```

```
   $connessione = mysql_connect($ip,$sql_user,$sql_password) or
die("connessione non riuscita: ".mysql_error());

   mysql_select_db($database_name, $connessione) or die("errore
nella selezione del database");



   if($id_param=="*"){



      mysql_query("TRUNCATE `data`",$connessione) or die("errore
query");

       mysql_query("TRUNCATE `gps`",$connessione) or die("errore
query");



   }else{



      $result = mysql_query("SELECT `ID_GPS` FROM `data` WHERE
`ID_PARAM` = ".$id_param,$connessione) or die("errore query");



      mysql_query("DELETE FROM `data` WHERE `data`.`ID_PARAM` =
".$id_param,$connessione) or die("errore query");



      while($row = mysql_fetch_row($result))

         mysql_query("DELETE FROM `".$database_name."`.`gps`
WHERE `gps`.`ID` = ".$row[0],$connessione) or die("errore
query");



   }



   mysql_close($connessione);
```

```
   if($id_param=="*"){
      header("Location: viewAll.php");
   }else{
      header("Location: view.php?id=".$id_param);
   }

?>
```

File **Colella_GPRS.cpp**; implementazione dei metodi riutilizzati dal progetto gsm-playground:

```
/************************************************************
  Initializes receiving process
************************************************************/
void Colella_GPRS::RxInit(){
   rx_state = RX_NOT_STARTED;

   prev_time = millis();

   memset(comm_buf,0,sizeof(comm_buf)); //erases communication
buffer
   comm_buf[0] = 0x00; //end of string
   p_comm_buf = &comm_buf[0];
   comm_buf_len = 0;

   Serial.flush(); // erase rx buffer
}
```

```
/*************************************************************

  Checks if receiving process is finished or not.

  Rx process is finished if defined inter-character tmout is
reached



  start_reception_tmout - maximum response time



  Returns:

        RX_NOT_FINISHED - not finished yet

        RX_FINISHED - finished (inter-character tmout occurred)

        RX_TMOUT_ERR - initial communication tmout occurred

*************************************************************/

byte Colella_GPRS::HasRxFinished(unsigned long
start_reception_tmout){

  byte num_of_bytes;

  byte ret_val = RX_NOT_FINISHED;  // default not finished



  // Rx fsm

  if (rx_state == RX_NOT_STARTED) {

    // Reception is not started yet - check tmout

    if (!Serial.available()) {

      // still no character received => check timeout

      if ((unsigned long)(millis() - prev_time) >=
start_reception_tmout) {

        // timeout elapsed => GSM module didn't start with
response
```

```
        // so communication is takes as finished
        comm_buf[comm_buf_len] = 0x00;
        ret_val = RX_TMOUT_ERR;
      }
    }
    else {
      // at least one character received => so init inter-
character
      // counting process again and go to the next state
      prev_time = millis(); // init tmout for inter-character
space
      rx_state = RX_ALREADY_STARTED;
    }
  }


  if (rx_state == RX_ALREADY_STARTED) {
    // Reception already started
    // check new received bytes
    // only in case we have place in the buffer
    num_of_bytes = Serial.available();
    // if there are some received bytes postpone the timeout
    if (num_of_bytes) prev_time = millis();


    // read all received bytes
    while (num_of_bytes) {
```

```
      num_of_bytes--;

      if (comm_buf_len < COMM_BUF_LEN) {

        // we have still place in the GSM internal comm. buffer
=>

        // move available bytes from circular buffer

        // to the rx buffer

        *p_comm_buf = Serial.read();

        p_comm_buf++;

        comm_buf_len++;

        comm_buf[comm_buf_len] = 0x00;  // and finish currently
received characters

                                        // so after each
character we have

                                        // valid string finished
by the 0x00

      }

      else {

        // buffer is full and we are in the data state => finish

        // receiving and dont check timeout further

        ret_val = RX_FINISHED;

        break;

      }

    }



    // finally check the inter-character timeout

    if ((unsigned long)(millis() - prev_time) >= INTERCHAR_TMOUT)
{
```

```
      // timeout between received character was reached

      // reception is finished

      // ----------------------------------------------

      comm_buf[comm_buf_len] = 0x00;  // for sure finish string
again

                                      // but it is not necessary

      ret_val = RX_FINISHED;

    }

  }

  return (ret_val);

}



/*******************************************************

   Waits for response.



   start_reception_tmout - maximum waiting time for receiving the
first response

                           character (in msec.)



   Returns:

      RX_FINISHED - finished, some character was received

      RX_TMOUT_ERR - finished, no character received (initial
communication tmout occurred)

*******************************************************/

byte Colella_GPRS::WaitResp(unsigned long
start_reception_timeout){

  byte status;
```

```
  RxInit();

  // wait until response is not finished

  do {

    status = HasRxFinished(start_reception_timeout);

  } while (status == RX_NOT_FINISHED);

  return (status);

}



/*******************************************************

   Waits for response with specific response string.



   start_reception_timeout - maximum waiting time for receiving
the first response

                           character (in msec.)

   expected_resp_string - expected string



   Returns:

      RX_FINISHED_STR_RECV - finished and expected string
received

      RX_FINISHED_STR_NOT_RECV - finished, but expected string
not received

      RX_TMOUT_ERR - finished, no character received initial
communication tmout occurred

*******************************************************/

byte Colella_GPRS::WaitResp(unsigned long
start_reception_timeout, char const *expected_resp_string)

{
```

```
byte status;
byte ret_val;


RxInit();
// wait until response is not finished
do {
  status = HasRxFinished(start_reception_timeout);
} while (status == RX_NOT_FINISHED);


if (status == RX_FINISHED) {
  // something was received but what was received?
  // ---------------------------------------------
  if(IsStringReceived(expected_resp_string)) {
    // expected string was received
    // ----------------------------
    ret_val = RX_FINISHED_STR_RECV;
  }
  else ret_val = RX_FINISHED_STR_NOT_RECV;
}
else {
  // nothing was received
  // --------------------
  ret_val = RX_TMOUT_ERR;
}
```

```
  return (ret_val);
}

/*******************************************************
   Checks received bytes

   compare_string - pointer to the string which should be find

   Returns:
        0 - string was NOT received
        1 - string was received
*******************************************************/
byte Colella_GPRS::IsStringReceived(char const *compare_string){
  char *ch;
  byte ret_val = 0;

  if(comm_buf_len) {
    ch = strstr((char *)comm_buf, compare_string);
    if (ch != NULL) {
      ret_val = 1;
    }
  }

  return (ret_val);
}
```

```
/************************************************************
   Sends AT command and waits for response

   AT_cmd_string - pointer to the AT command string to be sent
   start_reception_tmout - maximum response time
   response_string - pointer to the response string

   Returns:
      AT_RESP_ERR_NO_RESP - no response received
      AT_RESP_ERR_DIF_RESP - response_string is different from
the response
      AT_RESP_OK - response_string was included in the response
************************************************************/
byte Colella_GPRS::SendATCmdWaitResp(char const *AT_cmd_string,
char const *response_string, unsigned long
start_reception_timeout){
  byte status;
  byte ret_val = AT_RESP_ERR_NO_RESP;

  //delay 500 msec. before sending next AT command
  delay(AT_DELAY);

  Serial.println(AT_cmd_string);
  status = WaitResp(start_reception_timeout);
  if (status == RX_FINISHED) {
```

```
      // something was received but what was received?
      // -----------------------------------------
      if(IsStringReceived(response_string)) {
        ret_val = AT_RESP_OK;
        // response is OK
      }
      else ret_val = AT_RESP_ERR_DIF_RESP;
    }
    else {
      // nothing was received
      // --------------------
      ret_val = AT_RESP_ERR_NO_RESP;
    }

  return (ret_val);
}
```

Implementazione dei metodi riutilizzati dal progetto AVR-netino:

```
/************************************************************
   Url-encodes a string; WARNING: there must be enough space
   in urlbuf (in the worst case that is 3 times the length
   of str).

   str - the string to be encoded
   urlbuf - the url-encoded string
```

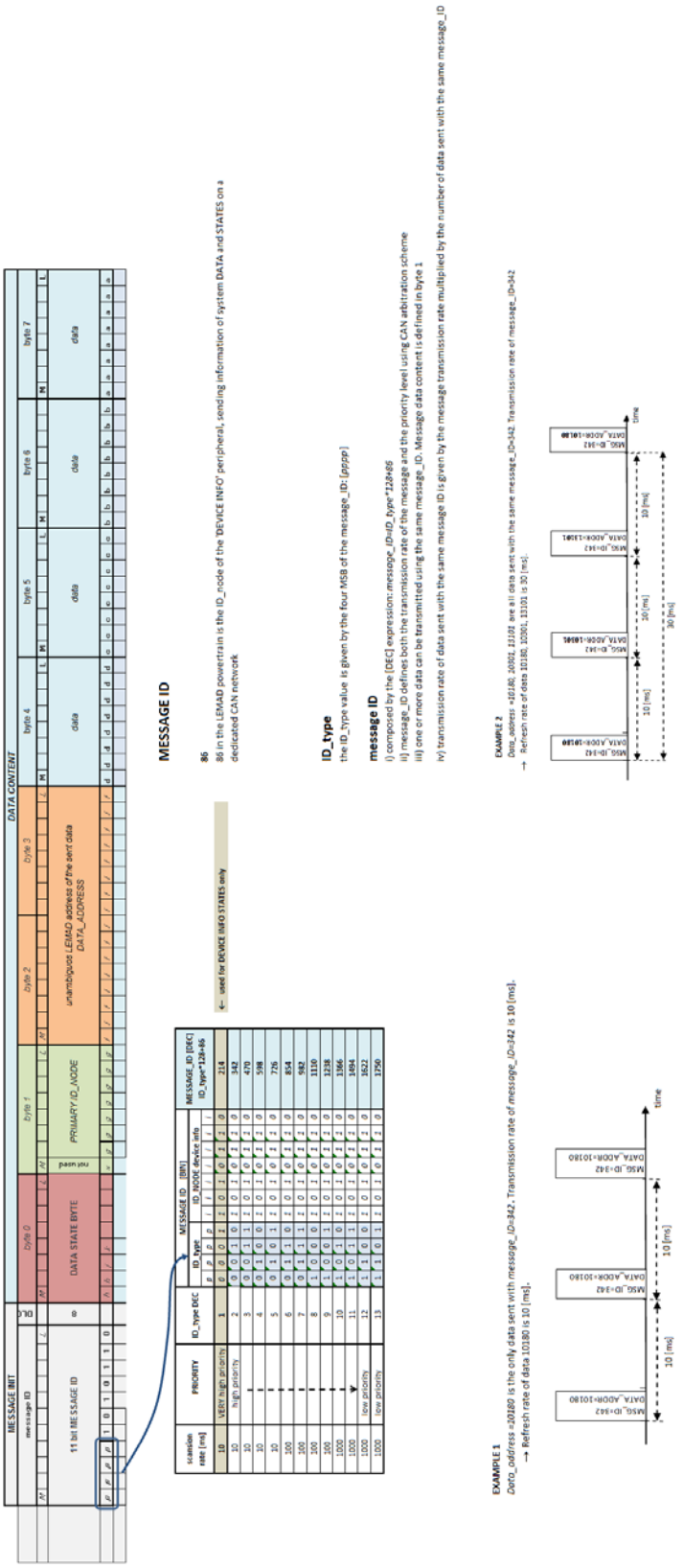
```
**********************************************************/
void Colella_GPRS::Urlencode(char *str,char *urlbuf){
   char c;
   while((c = *str)){
      if(c == ' '||isalnum(c)){
         if(c == ' ')
            c = '+';
         *urlbuf=c;
         str++;
         urlbuf++;
         continue;
      }
      *urlbuf='%';
      urlbuf++;
      int2h(c,urlbuf);
      urlbuf++;
      urlbuf++;
      str++;
   }
   *urlbuf='\0';
}

/**********************************************************
   Decodes an url-encoded string, e.g. "hello%20joe" or
```

```
   "hello+joe" becomes "hello joe".


   urlbuf - the url-encoded string
*********************************************************/
void Colella_GPRS::Urldecode(char *urlbuf){
   char c;
   char *dst;
   dst=urlbuf;
   while ((c = *urlbuf)){
      if(c == '+') c = ' ';
      if(c == '%'){
         urlbuf++;
         c = *urlbuf;
         urlbuf++;
         c = (h2int(c) << 4) | h2int(*urlbuf);
      }
      *dst = c;
      dst++;
      urlbuf++;
   }
   *dst = '\0';
}


/*********************************************************
   Converts a single character to a 2 digit hex string
```

```
   including the terminating '\0'.

   c - the input character
   hstr - the output string containing the hex value of c
*******************************************************/
void Colella_GPRS::int2h(char c, char *hstr){
   hstr[1]=(c & 0xf)+'0';
   if ((c & 0xf) >9)
      hstr[1]=(c & 0xf) - 10 + 'a';
   c=(c>>4)&0xf;
   hstr[0]=c+'0';
   if (c > 9)
      hstr[0]=c - 10 + 'a';
   hstr[2]='\0';
}

/*******************************************************
   Converts a single hex digit character to its integer
   value.

   c - the input character

   Returns: the integer value of c
*******************************************************/
```

```
unsigned char Colella_GPRS::h2int(char c){
   if (c >= '0' && c <='9')
      return((unsigned char)c - '0');
   if (c >= 'a' && c <='f')
      return((unsigned char)c - 'a' + 10);
   if (c >= 'A' && c <='F')
      return((unsigned char)c - 'A' + 10);
   return(0);
}
```

Appendice B - Documentazione CAN



DATA STATE BYTE

At each data is associated a DATA STATE BYTE reporting information about the data condition
This byte is subdivided in four blocks as follows

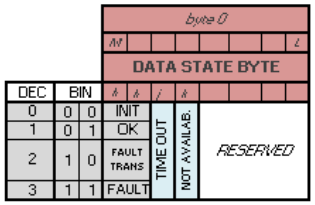
STATUS bit

The two MSB of DATA STATE BYTE gives information about the data

INIT: the primary device computing the data is in INIT phase → the data is not reliable
OK: the datais reliable
FAULT TRANS: the primary device computing the data is moving to a FAULT condition → the data is not reliable
FAULT: the primary device computing the data is in permanent FAULT condition or the DATA is OUT of RANGE

ADDITIONAL STATUS bit

TIME OUT=1: data cannot be collected from the primary device in the due time
NOT AVAILABLE =1: data is not available from the primary device (communication error)

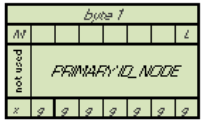


PRIMARY ID_NODE

reports the node ID of the device sending the DATA inside the LEMAD PWT network

The ID_node of each primary device is given by the following table

transmitting node	node ID	BIN	HEX
Device info	86	1010110	56
VMU	21	0010101	15
Traction inverter	41	0101001	29
BMS 1	101	1100101	65
BMS 2	102	1100110	66
BMS 3	103	1100111	67
BMS 4	104	1101000	68
BMS master	117	1110101	75

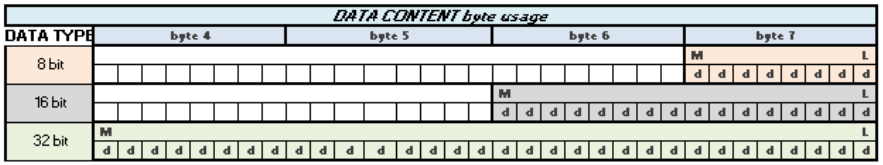


DATA ADDRESS

DATA_ADDRESS is the 16 bit unique address of the displayed data

DATA FIELD

i) data can be represented by 1 byte, 2 byte, 4 byte depending on their type and using different data type
ii) data ar aligned depending on byte usage as represented in the table
iii) data can also be represented by single bit FLAG



DATA SENT by the 'DEVICE INFO' peripherals

LEMAD reserved PCS LEVEL	ID TYPE	variable	unit	scale factor	data lenght	C data type	range	transmission rate [ms]	MESSAGE INIT message ID	DLC	DATA CONTENT byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
									M L		M L	M L	M L		M L	M L	M L	M L
2	3	MOTOR CURRENT	[A_{RMS}]	0,1	2 byte	unsigned short int	0 to 65536	10	0 0 1 1 1 0 1 0 1 1 0 470 message ID with HIGH priority	8	h h j k DATA STATE	0 1 0 1 0 0 1 41 TRACTION INVERTER	0 0 1 0 0 1 1 1 1 1 0 0 0 1 0 0 10180 INDIRIZZO_CORRENTE_DI_INVERTER_RMS		NOT USED		VALUE	
2	3	BATTERY CURRENT	[A]	0,1	2 byte	signed short int	-32768 to 32767	10	0 0 1 1 1 0 1 0 1 1 0 470 message ID with HIGH priority	8	h h j k DATA STATE	1 1 1 0 1 0 1 117 BMS MASTER	0 0 1 0 1 0 0 0 0 0 1 1 1 1 0 1 10301 INDIRIZZO_BATTERY_CURRENT1		NOT USED		VALUE	
5	6	MOTOR WINDING TEMPERATURE	[°C]	0,01	2 byte	signed short int	-32768 to 32767	1000	0 1 1 0 1 0 1 0 1 1 0 854 message ID with LOW priority	8	h h j k DATA STATE	0 0 1 0 1 0 1 21 VCU	0 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 13101 INDIRIZZO_MOTOR_TEMPERATURE1		NOT USED		VALUE	
2	3	DC BUS VOLTAGE	[V]	0,01	2 byte	unsigned short int	0 to 65536	10	0 0 1 1 1 0 1 0 1 1 0 470 message ID with HIGH priority	8	h h j k DATA STATE	1 1 1 0 1 0 1 117 BMS MASTER	1 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 33631 INDIRIZZO_TOT_VOLTAG_BMS		NOT USED		VALUE	
5	6	CHARGE LEVEL	[%]	0,01	2 byte	unsigned short int	0 to 65536	100	0 1 1 0 1 0 1 0 1 1 0 854 message ID with MID priority	8	h h j k DATA STATE	1 1 1 0 1 0 1 117 BMS MASTER	1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 32791 INDIRIZZO_LEVEL_BATTERY_CHARGE_BMS		NOT USED		VALUE	
6	7	CHARGE CURRENT	[A]	0,1	2 byte	unsigned short int	0 to 65536	1000	0 1 1 1 1 0 1 0 1 1 0 982 message ID with LOW priority	8	h h j k DATA STATE	1 1 1 0 1 0 1 117 BMS MASTER	1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 34901 INDIRIZZO_CURRENT_BATTERY_CHARGER		NOT USED		VALUE	
6	7	n. of equalizing cells		1	2 byte	unsigned short int	0 to 65536	1000	0 1 1 1 1 0 1 0 1 1 0 982 message ID with Low priority	8	h h j k DATA STATE	1 1 1 0 1 0 1 117 BMS MASTER	1 0 0 0 0 1 0 1 1 0 1 0 1 1 1 1 34231 INDIRIZZO_NUMERO_CELLE_ON_EQUALIZZATORE_BMS		NOT USED		VALUE	
3	4	MOTOR SPEED	[rpm]	0	4 byte	signed long int	int 32 bit	10	0 1 0 0 1 0 1 0 1 1 0 598 message ID with HIGH priority	8	h h j k DATA STATE	0 1 0 1 0 0 1 41 TRACTION INVERTER	0 0 1 0 1 0 1 1 0 1 0 1 0 0 0 0 11600 INDIRIZZO_ROTOR_SPEED_RPM		VALUE			
5	6	INVERTER TEMPERATURE	[°C]	0,01	2 byte	signed short int	-32768 to 32767	100	0 1 1 0 1 0 1 0 1 1 0 854 message ID with MID priority	8	h h j k DATA STATE	0 1 0 1 0 0 1 41 TRACTION INVERTER	0 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 13051 INDIRIZZO_POWER_TEMPERATURE1		NOT USED		VALUE	
5	6	AUX. battery voltage	[V]	0,1	2 byte	unsigned short int	0 to 65536	100	0 1 1 1 1 0 1 0 1 1 0 982 message ID with MID priority	8	h h j k DATA STATE	0 0 1 0 1 0 1 21 VCU	0 0 1 1 0 0 1 1 1 1 0 1 0 1 1 1 10031 INDIRIZZO_AUX_BATTERY_VOLTAGE		NOT USED		VALUE	
5	6	MAX. BATTERY TEMPERATURE	[°C]	0,01	2 byte	unsigned short int	-32768 to 32768	100	0 1 1 0 1 0 1 0 1 1 0 854 message ID with MID priority	8	h h j k DATA STATE	1 1 1 0 1 0 1 117 BMS MASTER	0 0 1 1 0 0 1 1 0 1 0 1 1 1 1 1 13151 INDIRIZZO_BATTERY_TEMPERATURE1		NOT USED		VALUE	
3	4	LOWER CELL VOLTAGE	[V]	0	2 byte	unsigned short int	0 to 65536	10	0 0 1 1 1 0 1 0 1 1 0 470 message ID with HIGH priority	8	h h j k DATA STATE	0 0 1 0 1 0 1 117 BMS MASTER	0 0 1 0 0 1 1 1 1 0 0 1 0 1 1 1 33621 INDIRIZZO_MIN_CELL_VOLTAGE_BMS		NOT USED		VALUE	
3	4	HIGHER CEL VOLTAGE	[V]	0	2 byte	unsigned short int	0 to 65536	10	0 0 1 1 1 0 1 0 1 1 0 470 message ID with HIGH priority	8	h h j k DATA STATE	 117 BMS MASTER	 33611 INDIRIZZO_MAX_CELL_VOLTAGE_BMS		NOT USED		VALUE	

Bibliografia

[1] http://arduino.cc/en/Main/arduinoBoardUno

[2] http://creativecommons.org/licenses/by-sa/3.0/

[3] http://arduino.cc/en/Main/ArduinoEthernetShield

[4] http://arduino.cc/en/Reference/Ethernet

[5] Sagem HiLo technical specification - URD1 5635.1 005 70086 Edition 03

[6] Sagem HiLo application note - URD1 5635.1 007 70230 Edition 01

[7] Sagem HiLo AT commands - URD1 5635.1 008 70248 Edition 02

[8] ISO 11898-1:2003

[9] http://arduino.cc/it/Reference/Wire

[10] http://www.w3.org/Protocols/rfc2616/rfc2616.html

[11] http://www.libelium.com/squidbee/index.php?title=New_GPRS_module_for_Arduino_%28Hilo_-_Sagem%29

[12] http://arduino.cc/it/Hacking/LibraryTutorial

[13] http://code.google.com/p/gsm-playground/

[14] http://code.google.com/p/avr-netino/

[15] *Brian W. Kernighan; Dennis M. Ritchie, The C Programming Language, 2nd ed., Prentice Hall, [aprile 1988]*

[16] http://httpd.apache.org/docs/2.0/

[17] http://news.netcraft.com/archives/category/web-server-survey/

[18] http://dev.mysql.com/doc/refman/5.5/en/

[19] http://php.net/docs.php