UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea

Studio, progetto e realizzazione di un analizzatore di traffico su rete CAN

Relatore:
Prof. Roberto Saletti

Relatore:
Prof. Roberto Roncella

Candidato:
Giuseppe Tognini

ANNO ACCADEMICO 2002-2003
3 dicembre 2003

Le tesi possono essere lavori più o meno originali che possono richiedere più o meno tempo, possono essere più o meno interessanti per chi le legge e per chi le scrive ... un'unica cosa è certa: non sono mai frutto del lavoro e dell'impegno di una sola persona. Per questo motivo voglio ringraziare il mio compagno di sventura in questa carriera universitaria e futuro ingegnere informatico, Manuele Piastra senza il cui aiuto sarei naufragato nei meandri del Visual C++ e del TEX; ringrazio l'ing. Federico Baronti per i sui preziosi consigli, il prof. Roberto Saletti, l'ing. Carlo Napoli, l'ing. Diego Lunardini e la mia ragazza per la pazienza che hanno avuto in tutti questi mesi e soprattutto ringrazio la mia famiglia che mi ha dato questa l'opportunità.

Indice

1 Protocolli di comunicazione industriali **2**

1.1 Il modello a strati . . . 2

1.1.1 Il modello ISO/OSI (OSI reference model) . . . 2

1.1.2 Pila e protocolli . . . 5

1.1.3 CAN, HLPs e profiles . . . 7

1.2 I bus di campo o Fieldbus . . . 10

1.2.1 Cos'è una Fieldbus Network? . . . 10

1.2.2 Perché utilizzare una Fieldbus Network . . . 10

1.2.3 Le tecnologie . . . 12

1.2.4 Caratteristiche a confronto . . . 13

1.3 I protocolli di alto livello del CAN . . . 14

1.3.1 Caratteristiche comuni . . . 14

1.3.2 OSEK/VDX . . . 16

1.3.3 SAE J1939 . . . 17

1.3.4 CAL . . . 19

1.3.5 CANopen . . . 21

1.3.5.1 Object Dictionary . . . 22

1.3.5.2 Tipi di messaggi . . . 22

1.3.5.3 Predefined Connection Set . . . 24

1.3.6 DeviceNet . . . 24

1.3.6.1 CIP e tipi di messaggi . . . 25

1.3.6.2 Caratteristiche . . . 26

1.3.7 SDS . . . 27

1.3.8 CANKingdom . . . 28

1.4 Le applicazioni del CAN . . . 29

1.4.1 Automotive . . . 29

1.4.2 Strumenti elettromedicali . . . 31

1.4.3 Applicazioni industriali . . . 32

2 Gli analizzatori di traffico **34**
2.1 Che cos'è un analizzatore di traffico . . . 34
2.2 DAQ e CAN in applicazioni automotive . . . 35
2.3 Tipi di analizzatori . . . 36
2.4 Descrizione dello stato dell'arte . . . 36
2.4.1 Warwick Control Technology . . . 40
2.4.1.1 X-Analyser . . . 40
2.4.1.2 X-Analyser CANpack . . . 42
2.4.1.3 X-Analyser REMOTE . . . 42
2.4.2 Intrepid Control Systems . . . 43
2.4.2.1 Vehicle Spy . . . 43
2.4.2.2 neoVI . . . 44
2.4.3 Vector Informatik Gmbh . . . 46
2.4.4 Architettura dell'interfaccia hardware . . . 47

3 Informazioni preliminari **51**
3.1 Introduzione al CAN . . . 51
3.1.1 Accesso al bus e struttura dei frame . . . 52
3.1.1.1 Struttura dei frame . . . 53
3.1.1.2 CSMA/CD . . . 60
3.1.1.3 Non-destructive bitwise arbitration . . . 61
3.1.2 Sequenza CRC . . . 65
3.1.2.1 Concetti base . . . 65
3.1.2.2 Calcolo del CRC . . . 66
3.1.2.3 Errori rilevati da CRC . . . 67
3.1.2.4 La sequenza CRC del protocollo CAN . . . 68
3.1.3 Gestione degli errori . . . 69
3.1.3.1 Cause di errore . . . 70
3.1.3.2 Condizioni di errore . . . 70
3.1.3.3 Fault confinement . . . 72
3.1.4 Codifica dei messaggi . . . 74
3.1.4.1 La codifica NRZ . . . 74
3.1.4.2 Bit-stuffing . . . 75
3.1.5 CAN physical layer . . . 75
3.1.6 Bit timing . . . 78
3.1.6.1 Il bit time di una rete CAN . . . 78
3.1.6.2 I quattro segmenti del bit time . . . 80
3.1.6.3 Programmazione dei bit-time segment . . . 80
3.1.6.4 Istante di campionamento . . . 81
3.1.6.5 Hard Synchronization . . . 81
3.1.6.6 Risincronizzazione . . . 82

3.1.7 Impostazione dei parametri 85
3.1.7.1 Il Propagation Delay 85
3.1.7.2 Oscillator Tolerance 87
3.1.7.3 Tolleranza massima ammissibile 88
3.1.7.4 Considerazioni sul Baudrate 92
3.2 Informazioni sul sistema di sviluppo utilizzato 95
3.2.1 Introduzione 95
3.2.2 L'ambiente di sviluppo 96
3.2.3 Implementazione della rete CAN di test 98

4 CAN*interface* 102
4.1 L'interfaccia hardware CAN*interface* 102
4.1.1 Il microcontrollore ATMEL T89C51CC01 103
4.1.1.1 Il circuito di generazione del clock 108
4.1.1.2 Il circuito di reset 111
4.1.1.3 Attivazione della modalità di programmazione 112
4.1.2 La memoria 113
4.1.3 Il transceiver Philips PCA82C250 114
4.1.4 L'interfaccia di collegamento RS232 con il PC 117
4.1.5 Il circuito di alimentazione 120
4.2 CAN2USB*interface*: una nuova interfaccia seriale 120
4.2.1 USB e RS232 a confronto 123
4.2.2 Caratteristiche del convertitore FT8U232BM 123
4.2.3 Come modificare CAN*interface* 124
4.2.3.1 Modifiche hardware 124
4.2.3.2 Modifiche software 127

5 Il software di CAN*analyser* 129
5.1 MICRO*software* 129
5.1.1 Struttura ed elenco dei comandi 130
5.1.1.1 I comandi inviati da CAN*software* 130
5.1.1.2 I comandi inviati da MICRO*software* 135
5.1.2 Dati inviati al PC durante l'acquisizione: "CAN messages" 137
5.1.2.1 "CodaRX": struttura ed utilizzo 137
5.1.2.2 Struttura dei "CAN message" 139
5.1.2.3 Gestione dell'overflow 141
5.1.2.4 Gestione del "time stamp" 143
5.1.3 Le altre procedure 147
5.1.3.1 La porta di comunicazione seriale 148
5.1.3.2 Programmazione della UART 148

5.1.3.3 Rilevamento del CAN bit-rate in automatico . 149
5.2 CAN*software* 150
5.2.1 L'architettura documento/vista 152
5.2.2 La classe "CCanDATA" 155
5.2.2.1 Le variabili membro 157
5.2.2.2 Le funzioni membro 160
5.2.3 CAN*software*: applicazione multithread 162
5.2.3.1 La classe "CCoda" 162
5.2.3.2 Task 3: la gestione della seriale 166
5.2.3.3 Task 2: costruzione degli oggetti CAN 167
5.2.3.4 Task 1: visualizzazione dei dati 171
5.2.4 Menu: "setup" 173
5.2.4.1 Parametri di bit timing 175
5.2.4.2 Filtro dei messaggi 178
5.2.4.3 Gestione dei "segnali" 179
5.2.4.4 Parametri di funzionamento dell'applicazione 184
5.2.5 Menu: "action" 184
5.2.5.1 Modalità di "Listening" 186
5.2.5.2 Modalità di "Playback" 186
5.2.6 Menu: "file" 186
5.2.6.1 Salvataggio dei dati 187
5.2.6.2 Esportazione dei dati 187
5.2.7 La barra di stato 188

6 Conclusioni **192**
6.1 Test dell'analizzatore 192
6.2 Performance 198
6.3 Limiti 201
6.4 Note dell'autore 203

A Configurazione hardware dell'ATMEL T89C51CC01 **205**
A.1 Configurazione dei pin 205
A.2 Circuito interno di generazione del clock 206
A.3 Circuito interno di generazione del reset 206

B ISP e flash memory del T89C51CC01 **210**
B.1 FM0 e FM1 210
B.2 In-System Programming (ISP) 211

C Gestione dei parametri di bit timing nel T89C51CC01 **216**

D Lo standard OBD-II **220**

E Diffusione del CAN nei sistemi di diagnostica on-board **224**

F Formati dei bit **229**
F.1 Il campo "Format Type" . . . 229
F.2 Il campo "Data Type" . . . 229

G Calcolo del "WorkLoad" **232**
G.1 "WorkLoad" istantaneo . . . 232
G.2 "WorkLoad" medio . . . 234

H Struttura dei file esportati **235**
H.1 Esportazione in formato ".csv" . . . 235
H.2 Esportazione in formato ".dif" . . . 236
H.2.1 Organizzazione di un file ".dif" . . . 238
H.2.2 Header . . . 238
H.2.3 Body . . . 239

I Codice di MICRO*software* **240**
I.1 File: "t89c51cc01.h" . . . 240
I.2 File: "compiler.h" . . . 248
I.3 File: "can_drv.h" . . . 250
I.4 File: "can_drv.c" . . . 251
I.5 File: "autobaud.a51" . . . 255
I.6 File: "config.h" . . . 259
I.7 File: "can_lib.h" . . . 260
I.8 File: "can_lib.c" . . . 264
I.9 File: "serial_drv.h" . . . 271
I.10 File: "serial_drv.c" . . . 271
I.11 File: "main.c" . . . 275

Elenco delle figure

1.1 OSI/ISO reference model . . . 3
1.2 Suddivisione in layer di due sistemi connessi . . . 6
1.3 Modello funzionale di uno *strato*. . . . 6
1.4 Layer, standard e implementazione di un bus CAN. . . . 8
1.5 CANopen: communication profiles e device profiles. . . . 9
1.6 Analogia tra linguaggio e protocolli di alto e basso livello. . . . 9
1.7 Esempio di sistema in rete a controllo distribuito[10]. . . . 11
1.8 Tabella riepilogativa delle caratteristiche di più diffusi protocolli "open" per Fieldbus Networks[13]. [6] . . . 15
1.9 Esempio applicativo del protocollo SAE J1939. . . . 18
1.10 CANopen Object Dictionary . . . 21
1.11 Struttura del campo CAN-Identifier nel CANopen-Predefined Connection Set. . . . 24
1.12 DeviceNet network . . . 26
1.13 Struttura del campo CAN-Identifier nel DeviceNet-Predefined Master/Slave Connection Set. . . . 27
1.14 SDS: struttura del campo CAN-Identifier. . . . 28
1.15 CANKingdom: rappresentazione convenzionale di una rete CAN. . . . 29
1.16 Schematizzazione di una rete CAN impiegata in campo automotive. . . . 30
1.17 Schematizzazione din una rete CAN impiegata per la realizzazione di un sistema *X-Ray angio-biplane*. . . . 31

2.1 Schema funzionale a blocchi di un analizzatore di traffico per rete CAN. . . . 37
2.2 Schema funzionale a blocchi della CAN interface . . . 38
2.3 Architettura di un CAN Controller di tipo *Stand-alone*. . . . 40
2.4 Warwick Control Technology: X-Analyser . . . 41
2.5 Warwick Control Technology: utilizzo di X-Analyser REMOTE per l'analisi remota di un sistema CAN. . . . 42
2.6 Intrepid Control Systems: Vehicle Spy . . . 43

2.7 Vector Informatik CANalyser 46
2.8 ESD: schema funzionale a blocchi dell'interfaccia "CAN-USB Mini Interface". 48
2.9 ESD: Schema funzionale a blocchi dell'interfaccia "CAN - Bluetooth". 49

3.1 Struttura schematica di una rete CAN. 52
3.2 Formato dei DATA e REMOTE frame - Formato standard (CAN1.2). 54
3.3 Formato dei DATA e REMOTE frame - Formato esteso (CAN-2.0). 55
3.4 Struttura dell'ERROR Frame. 58
3.5 Struttura dell'INTERFRAME Space. 59
3.6 Struttura dell'OVERLOAD Frame. 59
3.7 Comportamento del nodo CAN al rilevamento di errori nei campi di EOF e Intermission Space. 60
3.8 Meccanismo del Wired-AND. 62
3.9 Diagramma a blocchi esplicativo delle operazioni eseguite da ogni nodo CAN durante la fase di arbitraggio del bus. 63
3.10 Meccanismo dell'arbitraggio *bitwise non distruttivo*: (A) I nodi 1, 2 e 3 iniziano contemporaneamente la trasmissione del messaggio, il nodo 3 rileva un bit dominante, sospende la trasmissione ed entra in ricezione. (B) Anche il nodo 1 entra in ricezione. (C) Il nodo 2 vince l'arbitraggio e *continua* l'invio del frame, il nodo 1 filtra l'ID e lo accetta, il nodo 3 lo rifiuta e non elaborerà i dati trasmessi da 2. 64
3.11 Calcolo del CRC: divisione binaria modulo 2 tra $x^R T(x)$ e $G(x)$. 67
3.12 "CRC Area" e coefficienti del polinomio di calcolo $M(x)$. . . . 68
3.13 Sezioni del frame su cui il CAN verifica la presenza di Form-Error. 71
3.14 Sezioni del frame su cui il CAN verifica la presenza di Bit-Error. 72
3.15 Regole di *fault confinement* dei nodi di una rete CAN. 73
3.16 Codifica dei bit mediante l'uso della tecnica NRZ. 74
3.17 Metodo del bit-stuffing. 76
3.18 ISO11898-2 physical layer. 77
3.19 ISO 11898-3 e ISO 11992 physical layer. 78
3.20 Schematizzazione del processo di determinazione del *bit time*. . 79
3.21 Esemplificazione della procedura di sincronizzazione tra due nodi di una generica rete. 82
3.22 Sincronizzazione dei nodi: errore di fase nullo. 83
3.23 Sincronizzazione dei nodi: errore di fase positivo. 84

3.24 Sincronizzazione dei nodi: errore di fase negativo. 84
3.25 Esemplificazione di un arbitraggio errato per effetto del *Propagation Delay*. 86
3.26 Ritardo introdotto sulla propagazione del segnale *Propagation Delay*. 87
3.27 Valutazione dello shift di fase tra due nodi nel caso peggiore (worst case). 88
3.28 Local Error in presenza di almeno due nodi Error Active. . . . 89
3.29 Local Error in presenza di nodi tutti passivi. 90
3.30 92
3.31 Costruzione del bit time per ottenere la condizione di *Maximum Oscillator Tolerance*. 93
3.32 Bit timing per ottenere *Maximum Bit Rate*. 94
3.33 Variazione del bitrate in funzione della lunghezza del bus. . . . 94
3.34 Schema funzionale del ciclo di sviluppo del software per mezzo del software μVision2. 98
3.35 Rete CAN realizzata per il test del sistema. 99
3.36 Processo di acknowledgment check del messaggio in una rete CAN. 100

4.1 ATMEL T89C51CC01: schema funzionale a blocchi. 103
4.2 ATMEL T89C51CC01: mailbox ed gestione della memoria tramite registri SFR. 105
4.3 Esempio di configurazione di un CAN buffer. 106
4.4 CAN*interface*: schema di collegamento del microcontrollore ATMEL T89C51CC01. 109
4.5 CAN*interface*: schema del circuito di generazione del clock. . . 111
4.6 CAN*interface*: schema del circuito di generazione del reset. . . 112
4.7 CAN*interface*: circuito di gestione delle modalità di programmazione/esecuzione. 112
4.8 ATMEL T89C51CC01: organizzazione della memoria RAM. . 114
4.9 Philips PCA82C250: schema funzionale a blocchi. 116
4.10 CAN*interface*: schema di collegamento del transceiver Philips PCA82C250. 116
4.11 Philips PCA82C250: diagramma di temporizzazione del transceiver. 117
4.12 Maxim MAX232: schema funzionale a blocchi. 118
4.13 CAN*interface*: schema di collegamento del Maxim MAX232. . 119
4.14 CAN*interface*: circuito di alimentazione. 121
4.15 CAN*interface*: schema completo. 122
4.16 USBMOD1: schema del modulo prodotto dalla “Raven”. . . . 125

4.17 CAN2USB*interface*: schema funzionale. 126

5.1 MICRO*software*: descrizione dell'handshake tra CAN*software* e MICRO*software*. 131
5.2 ATMEL T89C51CC01: meccanismo di filtraggio degli identificatori o "Acceptance Filtering". 133
5.3 MICRO*software*: dettaglio della struttura del comando di overbuffer. 136
5.4 ATMEL T89C51CC01: diagramma semplificato a blocchi dell'utilizzo dei registri CANTIM e CANSTMP. 136
5.5 MICRO*software*: struttura ed organizzazione di "CodaRX". . 138
5.6 MICRO*software*: formattazione ed organizzazione dei "CAN message". 140
5.7 MICRO*software*: sincronizzazione dei "CAN messagge" l'invio del comando di overrun. 144
5.8 MICRO*software*: overrun in condizione di "CodaRX" vuota. . 146
5.9 MICRO*software*: overrun in condizione di "CodaRX" piena. . 147
5.10 ATMEL T89C51CC01: timing dei segnali della UART configurata in modalità 1. 149
5.11 MICRO*software*: diagramma di flusso a blocchi della procedura di "autobaud()". 151
5.12 Applicazioni SDI e MDI: gerarchia delle finestre 152
5.13 CAN*software*: architettura dell'interfaccia. 153
5.14 CAN*software*: viste dell'applicazione. 156
5.15 MICRO*software*: esemplificazione dell'interazione tra gli oggetti "CCanDATA" ed una delle viste (CCanDataView). . . . 157
5.16 CAN*software*: schematizzazione delle strutture dati condivise tra i task e delle operazioni che interagiscono con esse. 163
5.17 CAN*software*: implementazione di "CCoda" 165
5.18 Esempio di funzionamento delle primitive semaforiche: i task A, B, C e D condividono la stessa risorsa per mezzo di un semaforo che consente l'accesso contemporaneo alla risorsa ad un massimo di 3 task (semaforo inizializzato a 3). 166
5.19 CAN*software*: diagramma di flusso a blocchi della funzione "run()". 168
5.20 CAN*software*: task di costruzione degli oggetti "CCanDATA". 169
5.21 CAN*software*: meccanismo di visualizzazione dei dati attraverso le 4 viste disponibili. 172
5.22 MICRO*software*: visualizzazione dei messaggi. 173
5.23 MICRO*software*: visualizzazione dei "segnali". 174
5.24 MICRO*software*: visualizzazione grafica dei "segnali". 174

5.25 MICRO*software*: visualizzazione delle statistiche. 175
5.26 CAN*software*: finestra per l'impostazione dei parametri di bit timing della rete CAN. 176
5.27 CAN*software*: tabella riepilogativa dei parametri di configurazione. 177
5.28 CAN*software*: finestra di impostazione dei parametri di filtro dei messaggi. 178
5.29 CAN*software*: finestra di editing dei segnali. 180
5.30 CAN*software*: visualizzazione dei segnali. 181
5.31 CAN*software*: finestra di impostazione dei parametri. 185
5.32 CAN*software*: menu "file". 187
5.33 CAN*software*: esempio di un file ".dif" generato da CAN*software* ed aperto con *Microsoft Windows EXCEL* ©. 189
5.34 CAN*software*: barra di stato dell'applicazione. 190

6.1 Rete CAN realizzata per il test del sistema. 193
6.2 Schema di generazione dei burst utilizzati per il test. 195
6.3 Rete CAN realizzata per la seconda serie di test del sistema. . 196
6.4 Visualizzazione dei dati acquisiti durante una delle prove della seconda serie di test. 197
6.5 Visualizzazione della condizione di "overflow". 202

A.1 ATMEL T89C51CC01: configurazione dei pin. 205
A.2 ATMEL T89C51CC01: parte interna al microcontrollore del circuito di oscillazione per la generazione del clock di sistema. 206
A.3 ATMEL T89C51CC01: parte interna al microcontrollore del circuito di generazione del reset. 207

B.1 ATMEL T89C51CC01: organizzazione della memoria flash. . . 210
B.2 ATMEL T89C51CC01: area di memoria destinata al bootloader. 212
B.3 ATMEL T89C51CC01: schematizzazione del processo di "boot" dopo il reset del sistema. 212
B.4 ATMEL T89C51CC01: schema di programmazione della flash memory. 214

C.1 ATMEL T89C51CC01: struttura generale del bit time ricavato dai parametri di configurazione. 218

D.1 OBD-II: schema del connettore di diagnostica on-board. 221
D.2 OBD-II: posizionamento abituale del connettore. 222

E.1 Tabella riepilogativa dei protocolli attualmente usati e prospettive di impiego del protocollo CAN in applicazioni automotive: PARTE I. 225
E.2 Tabella riepilogativa dei protocolli attualmente usati e prospettive di impiego del protocollo CAN in applicazioni automotive: PARTE II. 226
E.3 Tabella riepilogativa dei protocolli attualmente usati e prospettive di impiego del protocollo CAN in applicazioni automotive: PARTE III. 227

F.1 CAN*software*: schematizzazione di come il CAN*software* interpreta i campi "Start Bit" e "Lenght". 230

G.1 CAN*software*: finestra di visualizzazione dei parametri di "WorkLoad". 232
G.2 CAN*software*: parametri per il calcolo del "WorkLoad". 233
G.3 CAN*software*: apertura della finestra temporale per il calcolo del parametro "WorkLoad" medio. 234

Elenco delle tabelle

1.1 Esemplificazione di alcuni protocolli. 7
1.2 Tecnologie attualmente in uso. 12
1.3 Elenco riepilogativo specifiche SAE relative al protocollo J1939. 18
1.4 CAL: identificatori associati ai servizi offerti dal CAL. 20
1.5 Assegnazione degli identificatori nel CANopen-Predefined Connection Set. 25
1.6 DeviceNet: struttura di un *Explicit Message*. 26

3.1 Tabella riepilogativa dei ritardi introdotti sul segnale dai singoli componenti di un nodo CAN. 86
3.2 Tabella riepilogativa dei valori del baudrate espressi in funzione della lunghezza della linea. 95

4.1 Caratteristiche dei chip ATMEL della serie CANary a confronto. 104
4.2 Philips PCA82C250: elenco delle caratteristiche tecniche del transceiver. 115
4.3 Tabella comparativa caratteristiche RS-232, USB. 123
4.4 MODUSB1: elenco dei pin più significativi. 126

5.1 CAN*software*: comandi inviati. 132
5.2 CAN*analyser*: dettaglio della struttura del comando "iniarray[8]". 134
5.3 MICRO*software*: dettaglio della struttura dei comandi inviati. 135
5.4 MICRO*software*: stima dei valori massimi di "bus load" che garantiscono l'acquisizione di dati in real-time senza perdita di informazioni. 142
5.5 MICRO*software*: comandi di bit timing inviati a CAN*software*. 150
5.6 CAN*software*: elenco ed interpretazione dei campi della vista "CCanDataView". 154
5.7 Tabella riepilogativa valori campo parametri. 190

A.1 ATMEL T89C51CC01: configurazione dei pin. 208

C.1 ATMEL T89C51CC01: struttura del CAN Bit Timing register 1. 216
C.2 ATMEL T89C51CC01: struttura del CAN Bit Timing register 2. 217
C.3 ATMEL T89C51CC01: struttura del CAN Bit Timing register 3. 218

Listati

3.1 Algoritmo di calcolo della sequenza CRC implementato nel protocollo CAN. 69
5.1 can_drv_it_overrun() 146
5.2 MICRO*software*: parte dichiarativa della classe “CCanDATA”. 155
5.3 MICRO*software*: definizione del tipo “tCtrlCmd” utilizzato per definire la variabile membro “m_Control”. 159
5.4 MICRO*software*: implementazione della funzione di generazione del campo CRC del frame per mezzo della funzione “GenerateCRCsequence”. 161
5.5 CAN*software*: parte dichiarativa della classe “CCoda”. 164
5.6 CAN*software*: codice della procedura “Thread_ListenMode” tramite cui si implementa il Task2. 169

A metà degli anni '80 la divisione automotive della società tedesca Robert Bosch GmbH creò un nuovo protocollo di comunicazione denominato Controller Area Network (CAN) nato per realizzare comunicazioni seriali più robuste necessarie all'interconnessione dei numerosi dispositivi elettronici presenti nei moderni veicoli a motore. Il risultato fù quello di realizzare autovetture più sicure, con consumi più contenuti, più semplici da produrre grazie ad un cablaggio meno complesso e quindi anche più leggere. Nato per applicazioni automotive, il protocollo CAN ha guadagnato ampia diffusione nel campo dell'automazione industriale e si è diffuso a tal punto da divenire uno standard internazionale (ISO 11898). In generale CAN può essere impiegato in tutte le applicazioni dove sottosistemi basati su microcontrollori e dispositivi di vario genere (sensori ed attuatori) devono comunicare tra loro. La sua grande diffusione ha fatto nascere l'esigenza di fornire ai progettisti di reti basate su questo protocollo strumenti che siano in grado di monitorare, analizzare ed acquisire le informazioni che transitano sul bus in tempo reale. Lo scopo di questo lavoro è quello di realizzare uno strumento di sviluppo e di diagnosi di reti e sistemi basati su questo protocollo e a cui ci riferiremo con il nome CAN*analyser*.

Il **Capitolo 1** nella prima parte fornisce una introduzione ai protocolli di comunicazione impiegati in reti sviluppate in ambito industriale. Nella seconda parte si esamina in quali settori il protocollo CAN trova oggi maggiore diffusione facendo alcuni esempi applicativi.

Il **Capitolo 2** fornisce una descrizione delle caratteristiche e funzionalità di alcuni analizzatori per reti CAN oggi disponibili in commercio.

Il **Capitolo 3** fornisce una descrizione del protocollo di comunicazione. In particolare, viene posta l'attenzione su tutti quegli aspetti ritenuti fondamentali per comprendere le scelte progettuali fatte nella realizzazione del CAN*analyser*.

Il **Capitolo 4** descrive in dettaglio l'interfaccia hardware da collegare via seriale al Personal Computer (CANinterface).

Il **Capitolo 5** descrive l'architettura del software che gira sull'interfaccia (MICROsoftware) e l'applicativo sviluppato per piattaforma Windows (CANsoftware) per mezzo del quale l'operatore interagisce con la rete.

Il **Capitolo 6** mostra gli scopi ed il modo in cui il CAN*analyser* può essere impiegato. A tale proposito mostreremo i risultati e le misure ottenute in alcune semplici rete che abbiamo testato.

Capitolo 1

Protocolli di comunicazione industriali

Nella prima parte di questo capitolo sarà offerta una visione di cosa sia una modellizzazione a livelli di un processo di comunicazione soffermando la nostra attenzione sul protocollo CAN. Nella seconda parte introdurremo le Fieldbus Network soffermandoci poi su i più importanti protocolli di alto livello oggi disponibili per reti CAN. Alla fine di tutto ciò dovrebbero essere abbastanza chiari i vantaggi di sfruttare una rete a bus ed un protocollo di comunicazione standard.

1.1 Il modello a strati

Gestire le comunicazioni tra i dispositivi di una rete è di per se un problema complesso e le differenze di hardware e software hanno creato, in passato, incompatibilità fra sistemi e difficoltà di interazione tra i servizi. Nei primi anni '80 la rapida evoluzione delle reti cominciò a creare grossi problemi alla interconnessione fra le stesse tanto da costringere le grandi multinazionali dell'information technology ad abbandonare i sistemi proprietari e muovere verso un sistema di integrazione basato su uno standard.

1.1.1 Il modello ISO/OSI (OSI reference model)

Il modello di riferimento OSI (Open System Interconnection) diventato parte dello standard ISO (International Organization for Standardization) nel 1984 divide la gestione di una qualunque rete in blocchi funzionali. Il modello non definisce uno standard tecnologico ma un modello di riferimento comune per le tecnologie ed i servizi che gestiscono le reti. *Esso assicura compatibilità e interoperabilità anche fra diversi tipi di tecnologie.*

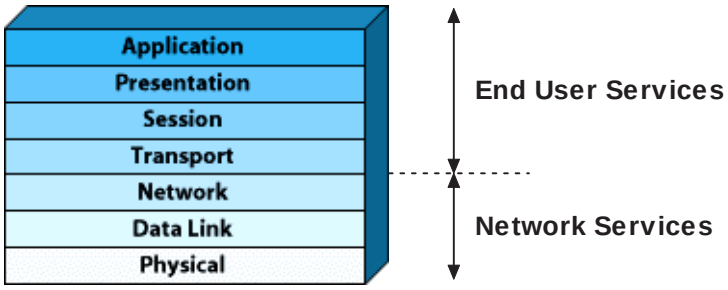


Figura 1.1: OSI/ISO reference model

Per capire il concetto di strutturazione per livelli è necessario comprendere come il flusso dei dati viene gestito e regolato; un buon esempio è il sistema postale. Ogni passaggio di consegna di una lettera avviene per livello di competenza e per regole di smistamento. La lettera viene imbucata in una cassetta postale dal mittente, prelevata dal postino e portata al centro di smistamento dove l'indirizzo sulla busta viene interpretato e la lettera viene veicolata verso un'altro centro più vicino alla destinazione finale dove seguirà il percorso analogo (a ritroso) fino ad arrivare nelle mani del destinatario.

I livelli del modello ISO/OSI sono sette, ciò è dovuto ad una aderenza sostanziale e ad una compatibilità inevitabile con il modello di comunicazione più in voga in quegli anni, il System Network Architecture (SNA) di IBM. Il problema della comunicazione viene suddiviso in blocchi funzionali ognuno dei quali dedicato a un problema specifico. Ogni blocco riceve informazioni dallo strato inferiore e comunica la propria elaborazione al blocco successivo. I blocchi, denominati tecnicamente *layer* o *livelli*, formano una struttura a pila in quanto la comunicazione fluisce dal livello più alto verso gli strati bassi del sistema trasmittente (dal mittente al centro di smistamento postale) e percorre invece la strada opposta sul sistema ricevente (dal centro di smistamento al destinatario). Se consideriamo i livelli nel modo indicato nella figura 1.1, il livello più in basso è quello connesso con il mezzo fisico di trasmissione, mentre quello più in alto si interfaccia direttamente con le applicazioni. Mentre i protocolli dei quattro livelli inferiori del modello OSI sono legati al tipo di infrastruttura trasmissiva usata e hanno lo scopo di ottenere da essa il servizio desiderato quelli dei tre livelli superiori sono orientati esclusivamente ad aspetti elaborativi e non sono legati a problematiche di telecomunicazione.

Livello 7: detto anche *Network Processes to Application* o più semplicemente *Application layer* è il livello più prossimo all'utente. Fornisce supporto alle applicazioni permettendo a queste di scambiarsi informa-

zioni attraverso la rete. Si deve prestare attenzione a non confondere il termine applicativo di questo strato con l'accezione comune di "programma". Il "programma" sfrutta gli strumenti della pila OSI a livello "applicativo" per svolgere funzionalità in rete.

Livello 6: denominato *Data Rapresentation* o *Presentation layer* serve per la trasformazione/conversione del formato dei dati, si occupa di prendere le informazioni dall'applicazione, convertirle in un formato standard comprensibile agli altri layer e viceversa. Con questo livello si provvede a risolvere le differenze che possono sussistere nel formato dei dati su due sistemi differenti. Per comprendere l'utilità di questo livello si considerino i comandi digitati su un terminale da una postazione pc, questi vengono convertiti in formato ASCII e poi trasferiti al sistema remoto, esistono però macchine che utilizzano standard differenti per la rappresentazione dei caratteri. Lo stesso problema esiste nella comunicazione tra sistemi che hanno modi diversi di considerare l'ordine dei bits nelle rappresentazioni numeriche. In questo layer si tendono a concentrare anche le esigenze di compressione/decompressione oppure le operazioni di cifratura e decodifica dei dati. *In conclusione il livello 6 si prende carico di tutti i problemi di presentazione dei dati e fornisce i meccanismi per una conversione trasparente.*

Livello 5: si preoccupa di monitorare e controllare il dialogo tra processi presenti su macchine differenti. Il layer implementa i meccanismi di sincronizzazione e mantenimento della comunicazione e per questo ci si riferisce ad esso con i termini di *Interhost Communication level* o *Session layer.*

Livello 4: a questo livello viene delegata la funzione di invio e ricezione dei dati ed infatti si parla di *Transport layer.* E' il primo livello che lavora in modo completamente indipendente dalla topologia della rete e costituisce la base su cui tutto il software applicativo si poggia per trasmettere le informazioni. Il programmatore viene sollevato dai dettagli della comunicazione e può limitarsi alla conoscenza delle interfacce standard e di poche altre convenzioni.

Livello 3: implementa la definizione del concetto stesso di rete. Il network inteso come insieme di sistemi autonomi che comunicano entro un mezzo fisico di trasmissione prevede un sistema di indirizzamento logico e di inoltro dei dati tra i nodi che viene definito nel *Network layer.*

Livello 2: attraverso il *Data link layer* si stabilisce un collegamento tra la gestione della connessione e la trasmissione dei dati ed tra questi compi-

ti rientra anche la definizione delle procedure di rilevazione degli errori e le successive correzioni. Ad esempio l'algoritmo *Carrier Sense Multiple Access/Collision Detection*[1] (CSMA/CD) presente anche nel protocollo CAN è implementato a questo livello.

Livello 1: si occupa della trasmissione dei dati sul mezzo fisico, è a questo livello detto *Physical layer* che viene specificata la rappresentazione elettrica degli 0 ed 1 binari, le velocità di comunicazione, la distanza massima di trasmissione, gli elementi fisici di interconnessione (la definizione meccanica dei connettori, le tipologie dei cavi ecc. . .). Questo è l'unico livello della gerarchia che riguarda direttamente l'hardware, tutti i blocchi funzionali superiori lavorano ad un livello logico operando su flussi di bit anziché su i segnali elettrici

Questo livello viene suddiviso nei seguenti sublayer (figura 1.4):

- *Physical Signalling*: codifica, temporizzazione e sincronizzazione.
- *Physical Medium Attachment sublayer*: caratteristiche dei driver e ricevitori (transceiver).
- *Medium Dependent Interface*: connettori.

1.1.2 Pila e protocolli

Consideriamo due sistemi[2] come quelli in figura 1.2 collegati tramite un mezzo fisico ossia un canale di comunicazione. Le funzioni che i due sistemi devono svolgere per poter comunicare vengono divise tra i vari layer ottenendo le suddivisioni in figura. I layer di uguale livello sono a comune fra tutti i sistemi connessi alla rete e costituiscono uno *strato*; i sistemi interagiscono tra di loro unicamente attraverso questo *strato* tramite strutture denominate*entity* secondo certe regole che costituiscono il *protocollo* di quel livello. Ogni strato attraverso le sue *entity* fornisce servizio a quello di livello superiore attraverso un'interfaccia[3] tra cui gli strati comunicano. In ogni strato

[1] Prevede la rilevazione del mezzo prima dell'immissione dell'informazione; il messaggio vero e proprio può essere caricato sul bus solo quando non esistono altre comunicazioni in corso. In caso di accesso simultaneo (collisione) ho vengono bloccate le comunicazioni ed assegnati tempi di retry casuali ai nodi responsabili dell'evento o attraverso regole di priorità si assegna il "possesso" del bus ad uno dei nodi.

[2] In questo contesto con il termine "sistema" ci si riferisce ad un qualunque apparato che eventualmente in congiunzione con un operatore umano è in grado di scambiare dati con altri sistemi.

[3] Ogni sistema ha una sola *entity* per layer e le stesse interfacce tra i livelli sono implementate attraverso particolari entity denominate Service Access Point (SAP).

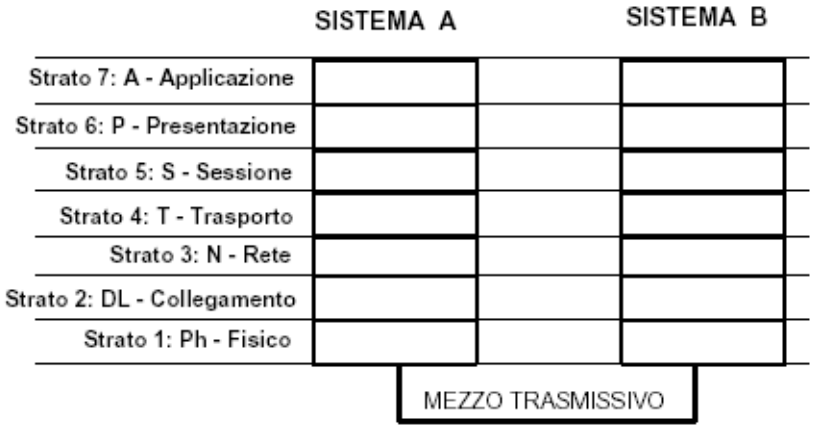


Figura 1.2: Suddivisione in layer di due sistemi connessi

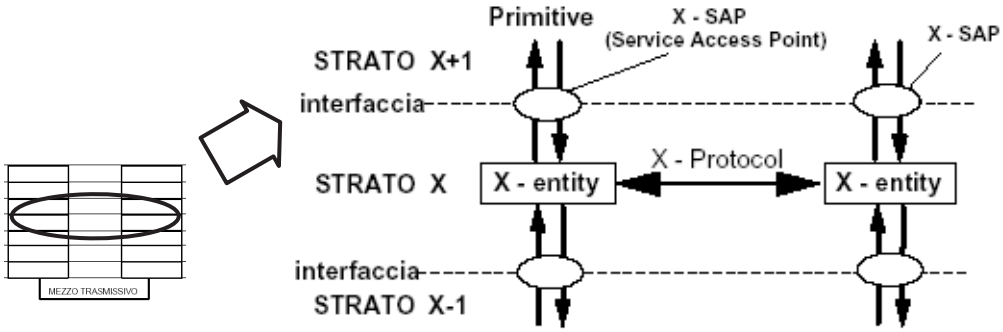


Figura 1.3: Modello funzionale di uno *strato*.

è quindi attivo un protocollo definito dall'ente di standardizzazione che ne definisce le caratteristiche ed attraverso cui i sistemi della rete interagiscono scambiandosi *messaggi*[4]. Le specifiche del protocollo di uno strato sono indipendenti da quelle degli altri e' quindi possibile sostituire il protocollo di uno strato senza dover modificare quello degli altri adiacenti (naturalmente il servizio fornito al livello superiore deve rimanere lo stesso).

In conclusione internamente ad ogni sistema i vari layer interagiscono fornendo e sfruttando servizi; sistemi differenti interagiscono facendo richiesta e fornendo servizi tramite un protocollo di comunicazione.

I protocolli attuali (di cui in tabella 1.1 sono riportati alcuni esempi) sono generalmente modulari ma questo non significa che aderiscano completamen-

[4]I dati vengono trasferiti organizzati in blocchi che il modello OSI chiama *messaggi*. I messaggi di livello 2 vengono usualmente chiamati *frame*.

Lv	LAYER	PROTOCOLLO
7	Application	CANopen, DeviceNet, CANKingdom, FTP
6	Presentation	-
5	Session	-
4	Transport	TCP, SPX, NetBEUI
3	Network	IP, IPX, NetBEUI
2	Data Link	CAN, TTCAN, IEEE 802[6]
1	Physical	CAN, IEEE 802[6]

Tabella 1.1: Esemplificazione di alcuni protocolli.

te alle specifiche dell'ISO. Il modello OSI è "solo" una raccomandazione ed il risultato di ciò è che i protocolli sono scritti a blocchi ma non si conformano in maniera completamente fedele ai sette livelli e ad esempio i livelli 5 e 6 non sono sempre implementati[5]. Spesso si cerca di concentrare le funzionalità dove possibile anche perché la visione a blocchi introdotta dallo standard comporta un certo overhead dovuto alle interfacce tra i diversi livelli.

1.1.3 CAN, HLPs e profiles

Come detto il modello ISO/OSI è largamente utilizzato per descrivere la funzionalità dei sistemi di comunicazione in base ad un approccio a livelli gerarchici. L'implementazione del protocollo CAN nell'attuale versione (2.0) copre solo parzialmente (figura 1.4) i 2 livelli più bassi mentre varie soluzioni software coprono i restanti layers della pila. I protocolli che lavorano a livello utente e denominati in gergo *Higher Layer Protocols* (HLP) sono sviluppati per differenti campi di applicazione e sulla base delle differenti esigenze degli operatori. Per meglio comprendere cosa sono questi protocolli di alto livello possiamo usare la seguente analogia ([4]); la funzione del CAN nei sistemi è la stessa delle lettere dell'alfabeto per la comunicazione tra le persone, è la base per scrivere un linguaggio ma non è sufficiente per consentire una comunicazione efficiente. Per costruire un linguaggio abbiamo bisogno anche di alcune regole, una grammatica che consenta di scrivere delle frasi comprensibili. Il progettista che implementa una rete sfruttando il protocollo CAN può definire un proprio linguaggio (definire un HLP proprietario) oppure può decidere di utilizzare un protocollo di alto livello standardizzato ed altamente specialistico come DeviceNet, CANKingdom o ancora sfruttare un protocollo

[5]Le finalità dei livelli 5 e 6 sono descritte dallo standard in maniera vaga e sono comunque tali da poterne integrare le relative funzionalità direttamente a livello applicativo.

[6]A cui fanno riferimento i protocolli Ethernet e Token Ring.

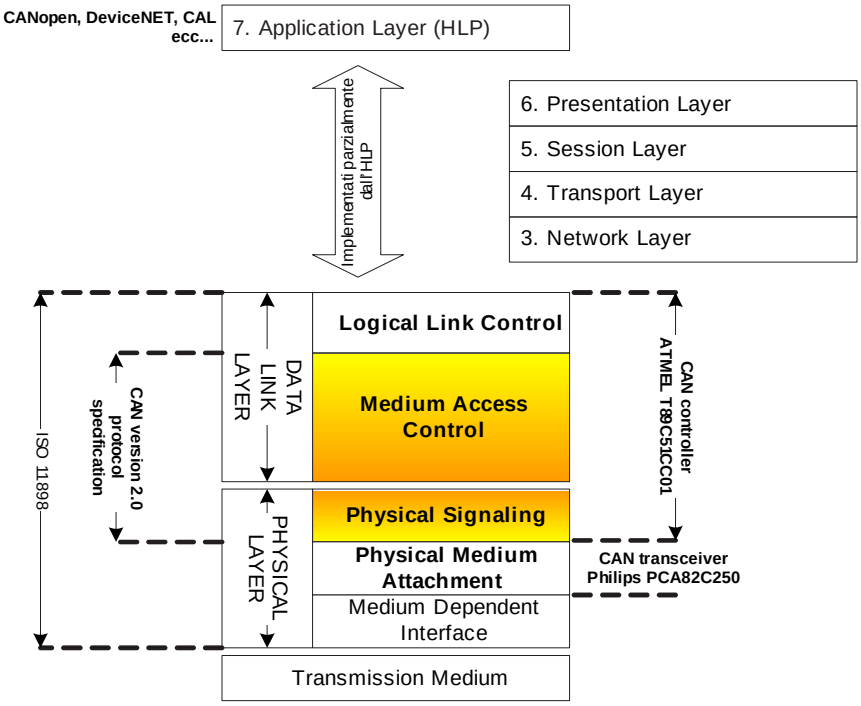


Figura 1.4: Layer, standard e implementazione di un bus CAN.

application independent come il CAN Application Layer detto CAL. Tornando alla nostra analogia l'uso di un dizionario (CAN) e di una grammatica (HLP) non è però sufficiente a rendere la comunicazione efficiente quando vogliamo ordinare da bere in un paese straniero dove la grammatica e le regole sono differenti. Per questo semplice scopo basterebbe avere a disposizioni delle semplici frasi già fatte da usare all'occorrenza per situazioni specifiche. Queste *frasi fatte* vengono chiamate *profile*; alcuni di essi mettono a disposizione di sistemi generici alcuni servizi di base standard altri sono realizzati allo scopo di interfacciarsi solo con una determinata classe di dispositivi ai quali forniscono funzioni altamente specializzate.

I profile si distinguono in *Device Profile* che definiscono le funzionalità dei singoli dispositivi ed in *Communication Profile* per mezzo dei quali vengono specificati i meccanismi di comunicazione e la loro descrizione. CANopen implementa sia funzioni di HLP che di profile (figura 1.5).

Impiegando profili standard dispositivi di I/O con funzionalità equivalenti possono essere tra loro facilmente rimpiazzati anche se realizzati da differenti costruttori proprio perché la definizione delle funzionalità dei singoli dispositivi viene specificata in forma standard. Per CANopen sono oggi disponibili[7]

[7]Sono attualmente in fase di definizione altri *profile* per unità di controllo porte e

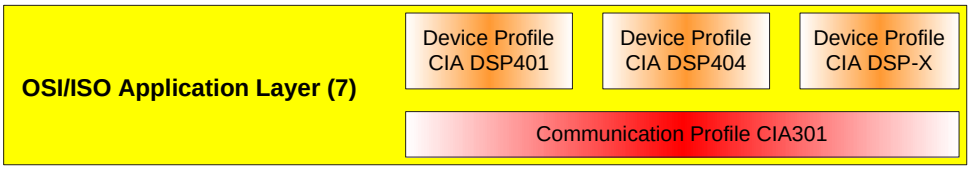


Figura 1.5: CANopen: communication profiles e device profiles.

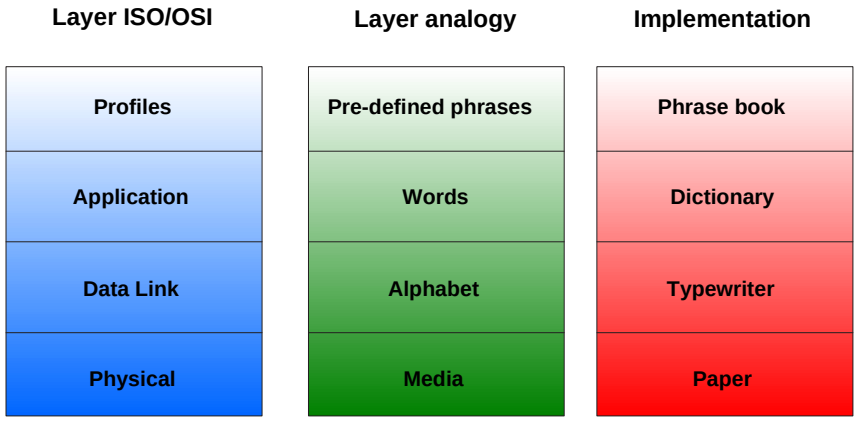


Figura 1.6: Analogia tra linguaggio e protocolli di alto e basso livello.

Devices Profile per:

- moduli generici di I/O
- controller per motori
- dispositivi di misura e controllers a circuito chiuso
- encoders
- valvole idrauliche a controllo proporzionale

In definita un sistema di comunicazione basato sul CAN può essere diviso in quattro livelli. Il livello fisico viene implementato attraverso un CAN controller ed eventualmente con altri componenti come i transceiver chip (se non integrati nello stesso controller). Il livello applicativo ed i suoi profiles vengono implementati via software[8].

sistemi frenanti in ambito ferroviario.

[8] Application layer e profile spesso non sono chiaramente distinguibili

1.2 I bus di campo o Fieldbus

I sistemi di controllo tradizionali impiegati in ambito industriale sono realizzati collegando i dispositivi che controllano piccole parti di un processo produttivo direttamente ad una workstation o ad un PLC; la comunicazione è realizzata mediante collegamenti di tipo punto-punto. Una soluzione certamente più efficiente è quella di connettere tutti i componenti del sistema tra loro sfruttando un bus e gestire la comunicazione punto-punto tra gli stessi tramite opportune implementazioni con il vantaggio che il sistema distribuito realizzato è molto più flessibile; lo scambio di informazioni può essere organizzato secondo esigenze applicative specifiche ad esempio in modalità broadcast piuttosto che punto-punto.

1.2.1 Cos'è una Fieldbus Network?

Una Fieldbus Network è una speciale rete locale dedicata alle applicazioni nel campo dell'acquisizione dati e del controllo di sensori ed attuatori. Normalmente come mezzo di collegamento si impiega un cavo a due coppie twistato ed a differenza di altre reti come Ethernet dove le prestazioni della rete vengono valutate facendo riferimento quasi esclusivamente alla velocità di trasferimento di grosse quantità di dati (throughput), le reti Fieldbus sono ottimizzate per lo scambio di messaggi di comando e di stato cioè di frame di dimensioni normalmente modeste.

Esempi di protocolli standard impiegati in reti di questo tipo oggi disponibili sono il Filbus, LonWorks, WorldFIP, Profibus e poi tutti i protocolli di alto livello che si appoggiano al CAN e cioé CANopen, CANKingdom, SDS, CAL e DeviceNet. Sul mercato ne esistono molti altri; alcuni di essi sono protocolli proprietari[9] e richiedono un'opportuna licenza per poterli sfruttare, altri sono "open" e sviluppati da *user group* che si occupano della definizione delle specifiche e della loro diffusione.

1.2.2 Perché utilizzare una Fieldbus Network

Reti di questo genere sono impiegate ed oggi trovano sempre più ampia diffusione per i grandi vantaggi che offrono e che brevemente descriviamo di

[9] Ad esempio lo Smart Distributed System (SDS) è realizzato dalla Honeywell (divisione del gruppo Microsoft), il DeviceNet è stato sviluppato originariamente dalla Allen Breley ed ora è un diventato un protocollo aperto sostenuto da una associazione internazionale di costruttori (Open DeviceNet Vendor Association - ODVA).

[10] Il dispositivo di controllo locale comunica con il resto del sistema di regolazione e con la stazione di controllo che è impiegata per settare i parametri, acquisire lo stato ed estrarre le statistiche del processo.

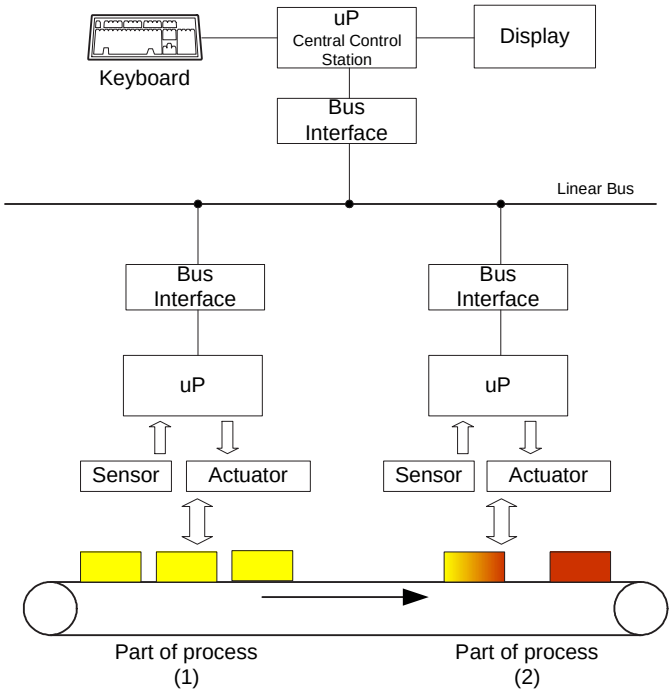


Figura 1.7: Esempio di sistema in rete a controllo distribuito[10].

seguito [5]:

- Cablaggio meno complesso e meno ingombrante. Non essendo più presente l'esigenza di una connessione fisica punto-punto diminuisce il numero delle unità di controllo (PLC o Workstation) da impiegare. Viene minimizzata l'occupazione degli spazi e tutto ciò si traduce in minori costi.

- Si riduco i tempi di installazione. Nuovi dispositivi possono essere rapidamente connessi in rete e funzioni di diagnostica possono essere implementate per identificare eventuali errori di interconnessione.

- Anche dal lato progettuale le cose si fanno più semplici e più rapide. Si riducono i tempi, i costi ed i rischi di sviluppo scegliendo soluzioni standard.

- Upgrade del sistema, modifiche e reimpiego per altri scopi si realizzato con spese minime.

- Migliorano ed aumentano i servizi che il sistema è in grado di fornire al suo utente.

Protocolli di comunicazione					
RS-232	RS-485	CAN	ARCNET	IEC 1158-2	BITBUS

Tabella 1.2: Tecnologie attualmente in uso.

1.2.3 Le tecnologie

Nella tabella 1.2 riportiamo l'elenco delle tecnologie (protocolli di basso livello [5]) impiegate in reti di questo genere e che sono oggi disponibili sul mercato. Considerando che uno degli scopi per cui sono nati i FieldBus è quello di ridurre la complessità del cablaggio si comprende perché questi protocolli implementano una comunicazione di tipo seriale, teniamo poi presente che moltissimi dispositivi integrati come convertitori analogico/digitali, LCD, sensori di temperatura e molti altri supportano una qualche tipo di interfaccia seriale. La scelta di impiegare bus di tipo parallelo rimane una scelta chiaramente vincente nelle situazioni in cui sono richieste operazioni di fetching e quindi di indirizzamento e trasferimento di blocchi di dati; la tendenza è quella di sfruttare accessi di tipo parallelo internamente per lo scambio di informazioni tra periferiche che compongono lo stesso dispositivo.

Per poter comprendere le caratteristiche e le differenze tra i vari tipi di interfaccia e tra i vari protocolli è necessario introdurre un pò di terminologia.

- In un bus *asincrono*[11] i dati sono inviati e ricevuti senza una precisa temporizzazione (la trasmissione di un secondo messaggio può partire ad un istante arbitrario rispetto la fine della trasmissione del precedente). In un bus *sincrono* è invece presente un *timing clock* che scandisce i tempi di I/O dei dati.

- Trattando della possibilità di comunicare in modo indipendente in entrambe le direzioni si parla di comunicazione *full-duplex* quando i dati possono essere simultaneamente trasmessi e ricevuti, di *half-duplex* quando il dispositivo può trasmettere e ricevere ma non contemporaneamente e *simplex* in cui il dispositivo è in grado trasmettere o ricevere.

- *Master/slave* descrive un tipo di bus in cui uno dei nodi svolge funzione di gestore (controlla e gestisce l'accesso al bus) della rete, tutti gli atri sono detti *slave* e sono semplici utilizzatori del canale di comunicazione.

[11] Elevate velocità di trasmissione sono raggiungibili con comunicazioni sincrone tuttavia i protocolli asincroni sono più facili da implementare. La maggior parte dei protocolli "industriali" sono asincroni.

Un bus di questo tipo è normalmente sincrono ed in tale ipotesi è il master a generare il timing clock.

- Un *Multi-master bus* è invece una rete di tipo master/slave in cui esistono più dispositivi che svolgono le funzionalità di master. In questi bus esiste un sistema di arbitraggio che risolve i conflitti che nascono quando più master richiedono contemporaneamente il controllo del canale.

- Una *Peer interface* o connessione *Point-to-point* descrive una relazione diretta di scambio dati tra due dispositivi. Tra gli elementi in comunicazione si stabilisce una connessione di tipo *client-server*.

- Il termine *Multi-drop* descrive un sistema in cui esistono più ricevitori e un solo trasmettitore.

- Con *Multi-point* si identifica un bus in cui esistono più di due transceivers.

- Una rete in cui non ci sono dispositivi che si dedicano al controllo degli accessi al bus è detta rete di tipo *Producer/Consumer*. Qualunque nodo può acquisire momentaneamente il controllo del bus ed iniziare una trasmissione (interfaccia *Producer*), gli altri nodi in questa fase si attiveranno in ricezione (interfacce *Consumer*). In questo modello di comunicazione il nodo della rete non specifica "a quale altro nodo è destinata l'informazione trasmessa", non fornisce l'indirizzo del destinatario ma specifica "da quale interfaccia i dati vengono prodotti (*Producer*)". Il CAN è un protocollo di comunicazione di questo genere ed i suoi HLP sfruttano questo modello adattandolo a comunicazioni di tipo *broadcast* o *punto-punto* asseconda delle applicazioni.

- Il *Token* è il gettone, in reti di questo tipo il nodo che ha il token ha il controllo del bus. Il modo ed i tempi con cui il token passa da un nodo all'altro dipendono dall'implementazione del protocollo di gestione del network.

1.2.4 Caratteristiche a confronto

La scelta del protocollo da impiegare e quindi la filosofia con la quale si decide di gestire il traffico sul bus dipende dalle caratteristiche e dai vincoli temporali di gestione delle variabili che devono transitare sul bus. Queste applicazioni dette *tempo critiche* hanno richieste di sincronizzazione, tra le informazioni prodotte dai sensori e quelle richieste dagli attuatori, più o meno

stringenti. Il bus è unico e quindi bisogna scegliere l'ordine con cui i nodi trasmettono. Le alternative sono le seguenti:

Approccio distribuito: quando non si hanno esigenze temporali stringenti si possono usare dei *Token* o impiegare metodi alternativi come i sistemi a *rilevazione di collisione*[12] utilizzato nelle reti CAN.

Approccio centralizzato: quando viceversa i vincoli di timing sono "importanti" conviene dotare la rete di uno o più nodi che funzionano da arbitri di controllo del canale di comunicazione.

Nella figura 1.8 sono riportate le caratteristiche di alcuni dei più diffusi protocolli "open" utilizzati in reti FieldBus.

> La scelta di adottare una FieldBus Network è una scelta quasi obbligata in molti campi applicativi. Esistono diversi protocolli tra cui poter individuare quello adatto a gestire dati che transitano sul bus. I parametri di comparazione tra queste alternative vanno dalle velocità di trasmissione supportate alle dimensioni della rete alle tecniche di accesso al bus.

1.3 I protocolli di alto livello del CAN

Nel capitolo 3 parleremo in modo approfondito del protocollo di comunicazione CAN in questa sezione invece l'attenzione si sofferma sui protocolli di alto livello nati ed adatti per sfruttare al meglio le potenzionalità di questo protocollo.

1.3.1 Caratteristiche comuni

Gli elementi comuni ai diversi HPL del CAN possono sostanzialmente essere riassunti negli scopi stessi per cui nascono i protocolli di alto livello e cioé definiscono:

- il comportamento dei nodi e quindi della rete nella fase di start-up;
- le regole con cui assegnare gli identificatori dei messaggi ai vari nodi;

[12] Un esempio è la tecnica CSMA/CD.

[13] Con riferimento alla figura 1.8 V: Vehicle automation, B: Building automation, F: Factory automation, P: Process industry, S: Smart instrumentation, D: Data communication

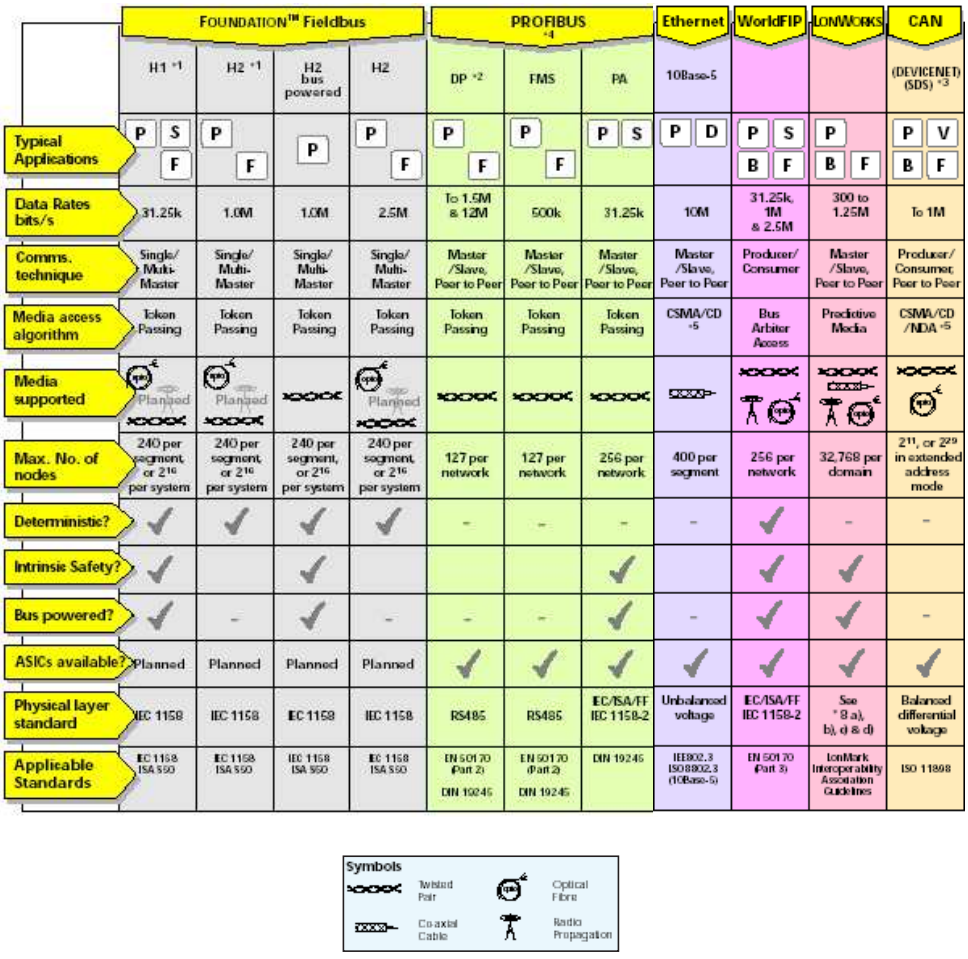


Figura 1.8: Tabella riepilogativa delle caratteristiche di più diffusi protocolli "open" per Fieldbus Networks[13]. [6]

- il controllo del flusso dei messaggi, la sincronizzazione delle funzioni e la definizione di un riferimento temporale;
- la possibilità di avere messaggi in grado di trasportare più dei tradizionali 8 byte di dati;
- il contenuto dei *Data Frame* cioé forniscono uno standard di codifica dell'informazione;
- i meccanismi di report dello stato del sistema (notifica degli errori e diagnostica);
- un modello per la descrizione delle funzionalità e dei parametri del dispositivo;
- un appropriato modello di comunicazione che normalmente può essere di tipo *Broadcast* o *Client-server*.

Ogni HPL implementa queste funzionalità in modo diverso adattandole alle specifiche esigenze del campo applicativo per cui è stato sviluppato. CANopen, DeviceNet e SDS nascono per applicazioni industriali standardizzate e quindi abbastanza generali, CAL fornisce una base standard da impiegare per l'ottimizzazione delle applicazioni di rete, il SAE J1939 e l'OSEK/VDX sono stati sviluppati per essere impiegati nei sistemi di diagnostica nelle autovetture.

> Grazie alle caratteristiche sopra elencate oggi il protocollo CAN rappresenta una soluzione particolarmente attraente per sviluppare applicazioni in sistemi distribuiti.

1.3.2 OSEK/VDX

Questo protocollo [7]è nato con l'obbiettivo di realizzare uno standard industriale da impiegare per un'architettura aperta destinata alla distribuzione dei controlli negli autoveicoli. L'OSEK si caratterizza in modo marcato dagli altri protocolli perché definisce a livello applicativo un vero e proprio sistema operativo (OS). L'OS fornisce i servizi per la gestione e sincronizzazione dei tasks, un meccanismo di interruzione, gestione degli errori e degli allarmi; attraverso di esso è stata realizzata una piattaforma per integrare i moduli software prodotti da differenti costruttori ed è questo uno dei motivi che rende oggi l'OSEK un'ottima soluzione per la progettazione di applicazioni time-critical che devono lavorare su un'ampia gamma di dispositivi hardware. La comunicazione tra il task locale e quello remoto avviene attraverso l'OS

per mezzo di due tipi di *Message Object*. Attraverso gli *State Message* i task si scambiano il valore attuale delle variabili di sistema mentre con gli *Event Message* il task remoto riceve tutte le altre informazioni di cui necessita. Attraverso questi messaggi è possibile implementare il modello di comunicazione desiderato: peer-to-peer o multicast con trasferimento dei dati di tipo periodico, event-driven o periodico/event-driven.

Le applicazioni automotive hanno specifiche stringenti dal lato sicurezza e per questo l'OSEK implementa un sofisticato *Network Management System* per il controllo della rete. Il monitoraggio dello stato dei nodi può avvenire in modo diretto "Direct Monitoring" da parte di tutti gli altri nodi della rete oppure in modo meno sofisticato sfruttando una seconda tecnica detta "Indirect Monitoring". Con il Direct Monitoring la stazione sotto osservazione invia un opportuno messaggio chiamato *Network Management message (NM message)* e siccome tutti i nodi devono uno dopo l'altro inviarne uno si rende necessario un sistema di sincronizzazione che ne gestisce i tempi di generazione. A tale scopo l'OSEK impiega un sistema di avviso (ring) attraverso cui la singola interfaccia viene abilitata ad inviare il messaggio NM agli altri ed a ricevere messaggi da loro. Se questo meccanismo risulta troppo complesso da implementare su uno dei dispositivi, sul nodo con queste problematiche si può applicare la tecnica di Inderect Monitoring[14].

1.3.3 SAE J1939

Si tratta di un altro tipo di protocollo open nato dagli sforzi della SAE (*Society of Automotive Engineers Heavy Truck and Bus Division*) originariamente sviluppato per coordinare le comunicazioni tra i sistemi elettronici (*Electronic Control Unit - ECU*) in veicoli mobili pesanti (autoarticolati) ed in tutte quelle applicazioni in cui si impieghino componenti derivati da essi.

Il SAE J1939 è più precisamente un set di *profile* (le cui specifiche sono standardizzate da SAE nelle specifiche riportate in tabella 1.3 basato su rete CAN rispondente allo standard ISO11898 e che impiega la versione extended del protocollo (identificatori a 29 bit).

Più precisamente il physical layer su cui il J1939 si poggia prevede l'impiego di un bus costituito da un'unica linea realizzata con una coppia di fili intrecciati e schermati (J1939/11) o meno (J1939/15) che percorre il veicolo per tutta la sua lunghezza ed a cui le ECU sono connesse. Per motivi di sicurezza la linea deve essere segmentata cioè realizzata interconnettendo tra loro più tratti ed utilizzando a tale scopo dei bridge (descritti dalle specifiche) che

[14]Si può consultare a tale proposito: OSEK Network Management, Version 2.0, Draft 1.1, 1997

J1939	Recommended Practice for a Serial Control & Communications Vehicle Network
J1939/11	Physical Layer, 250Kbit/s, Shielded Twisted Pair
J1939/13	Off-Board Diagnostic Connector
J1939/15	Reduced Physical Layer, 250Kbit/s, Unshield Twisted Pair
J1939/21	Data Link Layer
J1939/31	Network Layer
J1939/71	Vehicle Application Layer
J1939/73	Application Layer - Diagnostics
J1939/81	Network Management

Tabella 1.3: Elenco riepilogativo specifiche SAE relative al protocollo J1939.

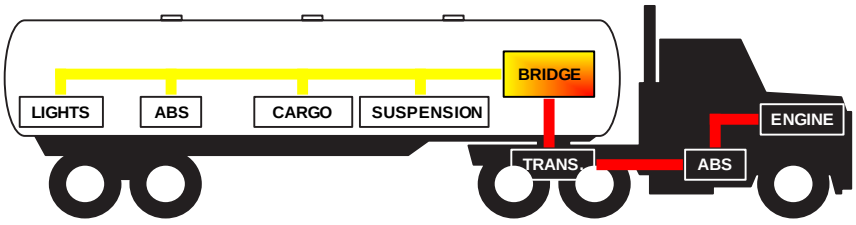


Figura 1.9: Esempio applicativo del protocollo SAE J1939.

assicurano l'isolamento elettrico tra le diverse sezioni che finiscono per costituire il bus, così facendo eventuali sezioni (ad esempio il rimorchio in figura 1.9) che dovessero presentare malfunzionamenti elettrici non degenerano le prestazioni delle altre parti della rete (motrice). La velocità di trasmissione è fissata in 250Kbit/s e la linea deve essere terminata da due resistori che smorzano gli effetti di riflessione dei segnali.

Lo standard prevede che i messaggi vengano raggruppati in 8 classi di priorità, hanno un tipo predefinito ed assegnando adeguatamente gli identificatori è possibile implementare un modello di comunicazione di tipo channel-oriented o broadcast. L'identificatore CAN viene costruito combinando tra loro le rappresentazioni binarie dei *Parameter Group Number (PGN)* ognuno dei quali codifica una particolare informazione: tipo di dato trasmesso con il messaggio, range, frequenza di trasmissione ecc... [15]. In più il J1939 introduce maggiore flessibilità nella trasmissione dei dati consentendo al nodo trasmittente (server) di specificare ad inizio trasmissione il numero di frame che gli altri nodi devono attendere di ricevere prima che il bus venga rilasciato e torni ad essere disponibile per una nuova fase di arbitraggio. Oggi questo protocollo è stato adattato nella versione J1939/73 per essere impiegato in sistemi di diagnostica, le sue funzionalità sono state ulteriormente estese consentendo l'implementazione di funzioni di start/stop, trasferimento di informazioni binarie (funziona, non funziona) ed informazioni di calibrazione. L'ultima revisione di questo protocollo standardizza anche la codifica e la tecnica di trasmissione dei DTC (Diagnostic Trouble Code) e cioè dei codici alfanumerici con cui i produttori di autoveicoli identificano lo stato del veicolo e che oggi (almeno quelli detti "generici" previsti dalle normative di protezione ambientale) sono divenuti comuni comuni a tutti i produttori[16]

1.3.4 CAL

Il CAL [8](sviluppato da *Philips Medical Systems*) è il protocollo di alto livello adottato da un gruppo di produttori ed utilizzatori del CAN denominato *CiA User Group*. La peculiarità del CAL (CAN Application Layer) è quella di offrire un sistema di sviluppo per l'implementazione di reti distribuite basate sul CAN indipendente dall'applicazione da realizzare e consente un approccio

[15] I PGN sono codificati e documentati dallo standard J1939/71 che nella sua ultima revisione conta circa 150 tipi di messaggi.

[16] In questo modo il J1939 si pone oggi come valida alternativa agli altri protocolli non basati sul CAN (J1850, Keyword2000, ISO9141) utilizzati attualmente nei sistemi OBD (On-Board Diagnostic) implementati in tutte le nuove autovetture e che dal 2003 in avanti vedono un sempre più ampio uso del CAN come protocollo standard di basso livello (appendice E).

CAN Application Layer	
COB-ID	**Usage**
0	NMT start/stop services
1-220	CMS objects priority 0
221-440	CMS objects priority 1
441-660	CMS objects priority 2
661-880	CMS objects priority 3
881-1100	CMS objects priority 4
1101-1320	CMS objects priority 5
1321-1540	CMS objects priority 6
1541-1760	CMS objects priority 7
1761-2015	NMT Node Guarding
2016-2031	NMT, LMT, DBT services

Tabella 1.4: CAL: identificatori associati ai servizi offerti dal CAL.

object-oriented alla descrizione della stessa. Il CAL fornisce al progettista quattro distinti servizi di alto livello:

1. **CMS** (*CAN-based Message Specification*):

   per mezzo del quale vengono programmate le funzionalità dei singoli nodi.

2. **NMT** (*Network ManagemenT*):

   offre servizi di supporto per la gestione della rete come la sua inizializzazione, start e stop dei nodi, rilevazione di malfunzionamenti; il servizio è implementato sfruttando connessioni di tipo master-slave (esiste un solo NMT master).

3. **DBT** (*DistriBuTor*):

   offre una distribuzione dinamica degli identificatori CAN ai nodi della rete. Questi identificatori vengono chiamati *Communication Object Identifier (COB-ID).*

4. **LMT** (*Layer ManagemenT*):

   offre la possibilità di modificare i parametri di configurazione dei nodi come ad esempio l'indirizzo NMT di uno dei dispositivi o il bit timing e baudrate della rete.

Il servizio CMS definisce otto classi di priorità per i messaggi, ogni classe dispone di 220 identificatori riservati. Gli identifiatori 0 e 1761-2031 sono dedicati agli altri servizi (vedi tabella 1.4).

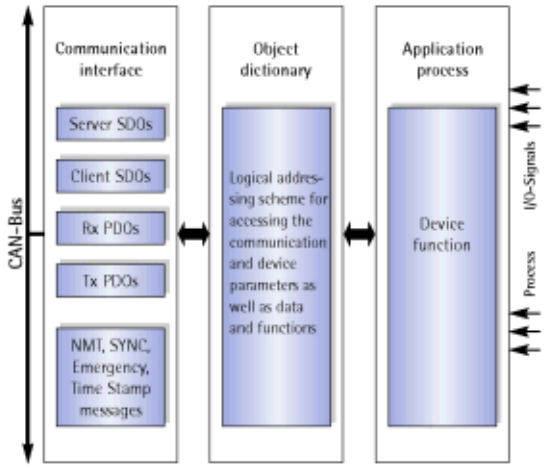


Figura 1.10: CANopen Object Dictionary

1.3.5 CANopen

Il protocollo CAL mette a disposizione una completa serie di servizi per la gestione della rete e dei messaggi ma non definisce il contenuto degli oggetti CMS; sostanzialmente definisce "come" avviene avviene la comunicazione e "come" il protocollo CAN viene sfruttato ma non defisce "cosa" transita sul bus ed è in questo contesto che si inserisce il CANopen.

CANopen ([9]) usa un sottoinsieme dei servizi definiti dal CAL per fornire l'implementazione di un sistema di controllo distribuito completo. CANopen è stato sviluppato dal CiA User Group[17]; nato inizialmente per il controllo di sistemi industriali è oggi impiegato anche per veicoli off-road, strumenti elettromedicali, e nella gestione dell'elettronica in campo navale e ferroviario.

Questo HLP sfrutta il concetto dei *profile* per garantire ai sistemisti la possibilità di far dialogare dispositivi realizzati da differenti produttori all'interno della solita rete senza l'esigenza di dover scrivere software dedicato per ogni singolo nodo. Per mezzo dei *Device profile standard* creati dal CiA (alcuni esempi sono stati riportati in figura 1.5) le funzionalità più comuni sono a disposizione del progettista, funzionalità più specifiche e strettamente legate all'hardware dell'interfaccia vengono gestite attravero profile messi a disposizione dai produttori stessi.

[17] CiA DS 301 Application Layer and Communication Profile conosciuto come CANopen specification, standard EN50325-4.

1.3.5.1 Object Dictionary

I profili sono descritti attraverso un database standardizzato[18] detto *Object Dictionary (OD)*. L'OD (figura 1.10) è un concetto base dell'implementazione di CANopen mentre non è definito in CAL. Si tratta di una tabella in cui i vari gruppi di oggetti vengono identificati attraverso un indice a 16 bits e per consentire l'accesso ai singoli elementi della struttura dati viene utilizzato un ulteriore sottoindice a 8 bit.

Per ogni nodo esiste un Object Dictionary che contiene tutti i parametri descrittivi del dispositivo ed il suo comportamento all'interno della rete. La parte dell'OD che descrive i parametri di comunicazione sono comuni a tutti i nodi mentre una sezione è riservata alla caratterizzazione del dispositivo. Il *Communication Profile* è unico e descrive la forma generale dell'OD e gli oggetti che vengono riservati a contenere i parametri per la configurazione della comunicazione. Esistono invece più *Device Profile* che definiscono in differente maniera alcuni degli oggetti dell'OD; ad ogni oggetto viene assegnato un nome, la funzione, indice e sottoindice (collocazione), tipo, viene specificato se è o meno obbligatorio e se l'informazione che contiene è modificabile o meno.

I campi di questa tabella sono accessibili in lettura e scrittura ed i messaggi per mezzo dei quali avviene l'interrogazione e la risposta da parte dei nodi sono detti *Service Data Object* (SDO). Un SDO specifica attraverso il campo DATA del messsaggio CAN, indice e sottoindice dell'oggetto dell'OD che si vuole leggere o scrivere quindi attraverso questo tipo di messaggi il protocollo fornisce un modo standard di accesso al dispositivo: segnali di I/O, parametri, funzioni e parametri di configurazione della rete.

1.3.5.2 Tipi di messaggi

Una rete CANopen deve avere un master e uno o più nodi slave. Il master si occupa di gestire la sequenza di boot, manipola i campi dell'*Object Dictionary* e gli identificatori CAN dei vari dispositivi connessi. I dati possono essere trasmessi in modo sincrono consentendo di coordinare acquisizione dati e attuatori oppure possono essere inviati in ogni istante e consentono di notificare immediatamente ad un nodo una richiesta di trasferimento senza attenderne una sincrona. Il contenuto di entrambi i tipi di messaggi possono essere riconfigurati al boot della rete dal master durante la fase di network management.

Senza entrare troppo nel dettaglio del modello di comunicazione del CA-

[18] Questo concetto è alla base di altri protocolli come il *Profibus* e l'*Interbus-S*.

Nopen, possiamo brevemente descrivere i 4 tipi di messaggi previsti dal protocollo:

1. **Administrative Message**:

   sono i messaggi per la gestione della rete e la distribuzione degli identificatori. Sono messaggi ad elevata priorità che sfruttano i servizi NMT e DBT del CAL. Attraverso di essi al boot del sistema il master stabilisce un dialogo con ogni slave della rete e stabilita la connessione le informazioni di configurazione vengono scaricate sul nodo.

2. **Special Function Objects**:

   in questa categoria rientrano messaggi predefiniti tra cui:

   - *SYNC*: utilizzato per la sincronizzazione del trasferimento dei dati, ovviamente si tratta di un messaggio ad altissima priorità;
   - *Time Stamp*: fornisce un riferimento temporale comune a tutti i nodi;
   - *Emergency*: generato all'occorrenza per la segnalazione di uno o più condizioni di errore;
   - *Node/Life Guarding*: il nodo che si occupa della gestione della rete (NMT master) monitora lo stato degli altri dispositivi. Ogni nodo viene continuamente interrogato per mezzo di un REMOTE frame, ogni nodo che supporta questa funzionalità risponde al master fornendo indicazioni sul proprio stato operativo. Tempi di inibizione (timeouts) possono essere impostati in moto tale che se il master non riceve risposta entro un tempo prestabilito identifica il nodo come non funzionante e lo gestisce di conseguenza. CANopen prevede opzionalmente il controllo anche del nodo master da parte degli altri nodi e questa funzioanlità è detta *Life Guarding*.
   - *Boot-up*: inviando questo messaggio l'NMT slave comunica al master il suo stato operativo. Ogni slaves funziona come una macchina a stati dotata di 4 step denominati *Initializzation, Pre-operational, Operational* e *Prepared mode*. Attraverso specifici messaggi, il master è in grado modificare lo stato di uno o più nodi simultaneamente.

3. **Process Data Object (PDO)**:

   sono utilizzati per il trasferimento real-time dei dati; le informazioni trasferite sono limitate a 8 byte e la natura dei dati è codificata attraverso l'identificatore CAN associato al messaggio (la tipologia del

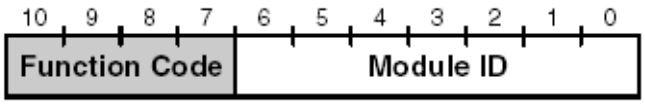


Figura 1.11: Struttura del campo CAN-Identifier nel CANopen-Predefined Connection Set.

dato definisce anche la priorità del messaggio e quindi l' "importanza" dell'informazione trasmessa). La trasmissione può avvenire in modo sincrono alla ricezione del messaggio SYNC che ha cadenza periodica oppure in modo asincrono su richiesta (a seguito della ricezione di un REMOTE frame) oppure al verificarsi di particolari eventi (se con il PDO si invia l'uscita digitale di un convertitore A/D usato per acquisire informazioni, quando il valore del segnale analogico cambia tramite un PDO l'interfaccia invia il valore aggiornato).

4. **Service Data Object (SDO)**:

   attraverso gli SDO CANopen consente l'accesso ai campi dell'OD dei singoli dispositivi. Essendo impiegati in fase di boot per la configurazione della rete sono messaggi a bassa priorità che consentono il trasferimento di grosse quantità di dati (superiore agli 8 byte che costituiscono un limite invalicabile per i messaggi PDO).

1.3.5.3 Predefined Connection Set

Allo scopo di ridurre il lavoro del progettista le specifiche CiA301 forniscono uno schema standard (*Predefined Connection Set* per l'assegnazione degli identificatori che viene riportato in figura 1.11. Per mezzo dei quattro bit più significativi (*Function Code field* vengono identificate 16 funzioni di base che altro non sono che le quattro tipologie di messaggi prima descritte e che riassumiamo nella tabella 1.5. Il campo *Node ID* viene impostato via hardware per ogni nodo normalmente per mezzo di un dip-switch.

1.3.6 DeviceNet

DeviceNet[19] è uno dei più sofisticati protocolli di alto livello oggi disponibili ed è particolarmente usato nel campo dell'automazione industriale. Come CANopen anche DeviceNet è un protocollo "aperto" cioè ogni membro

[19]Le specifiche sono oggi sviluppate dalla Open DeviceNet Vendor Association (ODVA). L'ultima versione attualmente disponibile è la 2.0.

Message	Function code	Identifier	Note
NMT	0000	0x000	PRIORITA' ALTA
SYNC	0001	0x080H	
TIME STAMP	0010	100H	
EMERGENCY	0001	081H-0FFH	080H+DIP
PDO1 (tx)	0011	181-1FFh	180h+DIP
PDO1 (rx)	0100	201-27Fh	200h+DIP
PDO2 (tx)	0101	281-2FFh	280h+DIP
PDO2 (rx)	0110	301-37Fh	300h+DIP
SDO (tx-server)	1011	581-5FFh	580h+DIP
SDO (rx-client)	1100	601-67Fh	600h+DIP
Nodeguard	1110	701-77Fh	700h+DIP

Tabella 1.5: Assegnazione degli identificatori nel CANopen-Predefined Connection Set.

della *Open DeviceNet Vendor Association (ODVA)* può partecipare al suo sviluppo. L'utilizzo della tecnologia DeviceNet non è sottoposto a licenza è sufficiente ricevere un ID dalla ODVA per l'identificazione del costruttore. Attualmente esisto oltre 300 compagnie registrate e approssimativamente 700 compagnie offrono prodotti compatibili con questo standard.

1.3.6.1 CIP e tipi di messaggi

DeviceNet assieme ad altri due standard: ControlNet e Ethernet/IP, basati su un application layer comune denominato *Control and Information Protocol (CIP)*. La parte *Control* del CIP implementa lo scambio dei dati in real-time attraverso i messaggi detti *I/O Message* o anche *Implicit Message*; questi non hanno una struttura predefinita dal protocollo e per questo motivo possono essere sfruttati per gestire lo scambio di informazioni in modo estremamente efficiente (il progettista è libero di gestirne l'implementazione). La parte *Information* definisce lo scambio di dati per le fasi di configurazione, diagnostica e gestione della rete per mezzo degli *Explicit Message* che dovendo essere riconosciuti da qualunque nodo della rete hanno una struttura definita a priori[20] (tabella 1.6).Attraverso il CIP si forniscono all'utente 4 funzioni di base e cioè:

- servizi di controllo;

[20]Parte del campo DATA del messaggio CAN è riservato a campi di configurazione che hanno un uso specifico e che limita l'impiego di questo tipo di messaggi per il trasferimento dati.

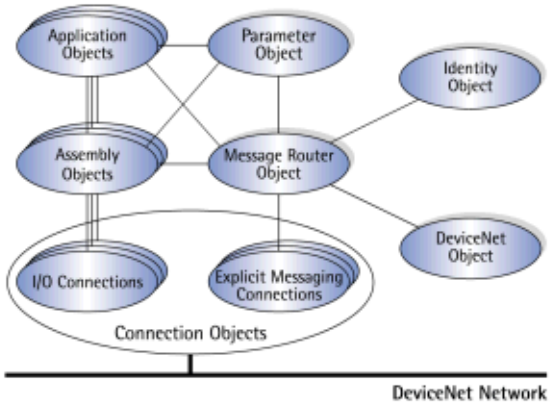


Figura 1.12: DeviceNet network

- servizi di comunicazione;
- distribuzione dei messaggi.

Byte	**CAN Identifier**							
0	D7	D6	D5	D4	D3	D2	D1	D0
1	FRAG	XID	MAC Id					
2	R/R	Service Code						
3	Class Id							
4	Instance Id							
5	Optional Data							
6								
7								

Tabella 1.6: DeviceNet: struttura di un *Explicit Message*.

1.3.6.2 Caratteristiche

Anche DeviceNet fornisce una configurazione predefinita degli identificatori per realizzare un semplice sistema Master-Slave; si tratta della così detta *Predefined Master/Slave Connection Set*.

Una rete basata su questo protocollo sopporta fino a 64 nodi usando baudrate compresi tra 125, 250 e 500 Kbaud. Inoltre le specifiche DeviceNet modificano in parte il layer 1 e 2 del CAN consentendo tra l'altro che l'alimentazione dei dispositivi possa essere fornita per mezzo del bus.

Confrontato con il CANopen, DeviceNet fornisce approssimativamente le stesse funzionalità organizzate ovviamente in modo differente. La gestione della rete (il network management) è distribuita su tutti i nodi, non c'è un NM master che supervisiona il traffico come accade in CANopen. Se da un lato il meccanismo di comunicazione è meno complesso e quindi i dispositivi dovrebbero essere più semplici, dall'altro l'alto livello di sicurezza implementato dal procotocollo richiede un forte impiego di risorse.

Entrando nel dettaglio di come vengono assegnati gli identificatori ai vari nodi la tecnica è quella di impiegare uno schema node-oriented. Ognuno dei 64 nodi del sistema ha un set di identificatori a sua disposizione suddivisi in 4 gruppi come mostrato in figura 1.13.

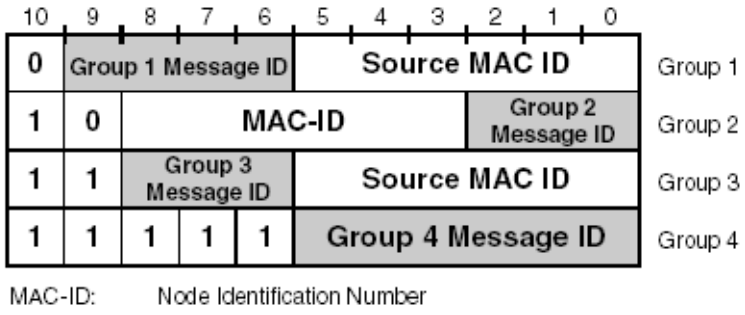


Figura 1.13: Struttura del campo CAN-Identifier nel DeviceNet-Predefined Master/Slave Connection Set.

1.3.7 SDS

Lo *Smart Distributed System (SDS)* è un protocollo di alto livello CAN-based sviluppato originariamente dalla *Honeywell Micro Switch*, nato per l'interconnessione soprattutto di sensori ed attuatori binari in lunghe catene di montaggio.

Lo schema di assegnamento degli identificatori è molto simile al DeviceNet (figura 1.14), in questo caso abbiamo due soli gruppi e la completa predefinizione dei messaggi che si distinguono in due categorie:

- *Short form*: in cui il messaggio CAN con possiede un campo DATA e ciò rende la dimensione di questi frame molto piccola e quindi di rapida trasmissione. Gli *Short form* vengono efficientemente impiegati per la gestione di dispositivi di attuazione che richiedono un semplice consenso di manovra e per sensori altrettanto elementari.

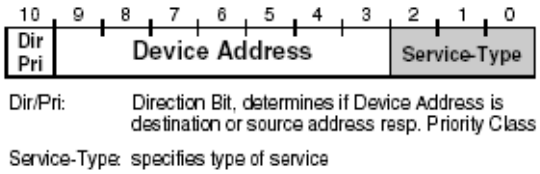


Figura 1.14: SDS: struttura del campo CAN-Identifier.

- *Long form*: per mezzo dei quali la rete viene configurata e gestita.

1.3.8 CANKingdom

CANKingdom sfrutta a pieno le potenzialità del CAN e fornisce al progettista massima libertà nella creazione del proprio sistema; nonostante che si utilizzi come protocollo di basso livello il CAN e quindi ne vengano sfruttate tutte le peculiarità il progettista è in grado di implementare virtualmente qualunque topologia e qualunque modello di comunicazione.

L'approccio con cui sono progettati i protocolli di rete precedentemente descritti è detto NSM (Network Serves the Module); quando un nuovo modulo viene connesso alla rete deve già sapere qual'è il suo ruolo all'interno del sistema, questo è uno dei ruoli che hanno i profili. Questa tecnica può da un lato essere limitante dato che le caratteristiche della rete devono essere note e definite a priori. La filosofia alla base di CANKingdom è esattamente duale alla precedente: "i nodi sono al servizio della rete" (MSN-Modules are to Serve the Network). Quando un modulo viene collegato alla rete deve solo annunciare che è connesso, il nodo dispone di tutte le informazioni di cui ha bisogno per inizializzare il sistema ed il progettista è completamente responsabile del software implementato in ogni singolo nodo.

In CANKingdom un sistema di controllo distribuito viene descritto come una nazione, un regno, con una capitale e tante città (i nodi). Il re (the King) che si trova nella capitale è il sovrano del regno mentre i suoi subalterni (Major) sono responsabili dei governi locali nelle singole città (figura 1.15). King e Major comunicano tra loro solo attraverso mail che un efficiente sistema postale smista tra la capitale e le città.

Il protocollo CANKingdom più che un HPL è un *meta protocol* definisce cioè delle primitive di protocollo, spetta al sistemista sfruttarle per definire un opportuno modello di comunicazione. Altra peculiarità consiste è che prevede un sistema unico di identificazione dei nodi attraverso l'impiego della standard EAN/UPC, in questo modo ogni singolo nodo è identificato in modo univoco a livello mondiale attraverso un codice numerico seriale.

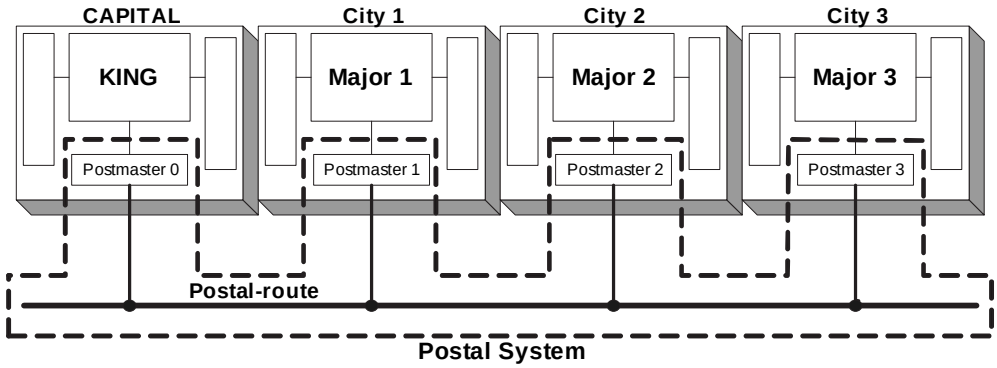


Figura 1.15: CANKingdom: rappresentazione convenzionale di una rete CAN.

1.4 Le applicazioni del CAN

1.4.1 Automotive

Il protocollo CAN si è sviluppato quando l'elettronica è iniziata ad entrare pesantemente nella gestione dei moderni veicoli a motore; esempi di questi dispositivi sono le sospensioni attive, l'ABS, il controllo del cambio, il controllo delle luci, il condizionamento dell'aria, gli airbag e le centraline di comando delle serrature. Il crescente impiego di dispositivi elettronici è il risultato in parte del desiderio di migliorare la sicurezza ed il comfort dell'automobilista ed in parte dell'esigenza di ridurre i consumi e migliorare le prestazioni del mezzo. La complessità delle funzioni realizzate necessita uno scambio continuo di informazioni che con i sistemi tradizionali si traduce nella realizzazione di cablaggi dedicati che collegano tra loro le singole interfacce ed i problemi nello "stendere" queste connessioni diventano tanto più grandi quanto più le funzioni da realizzare sono sofisticate e quindi richiedono la connessione di numerose unità di controllo. La soluzione consiste nell'impiegare un bus di tipo seriale realizzando così una Fieldbus Network come quella schematicamente riportata in figura 1.16 in cui le "vecchie" connessioni punto-punto sono sostituite da un'unica linea (bus) che percorre il veicolo per tutta la sua lunghezza e alla quale le interfacce e le centraline di comando si collegano.

La Daimler-Benz è stata il primo costruttore ad impiegare il protocollo CAN nella progettazione dei suoi sistemi di controllo; oggi molti veicoli prodotti da questa compagnia sfruttano una rete CAN a 500 Kbit/s per la gestione del motore e molte altre grosse aziende stanno seguendo il suo esempio (Audi, BMW, Saab, Volvo ecc...). Come si vede anche nella figura 1.16 il bus CAN sfruttato a velocità inferiori può essere validamente utilizzato per la connessione di tutta quell' elettronica che non ha stringenti esigenze di tipo

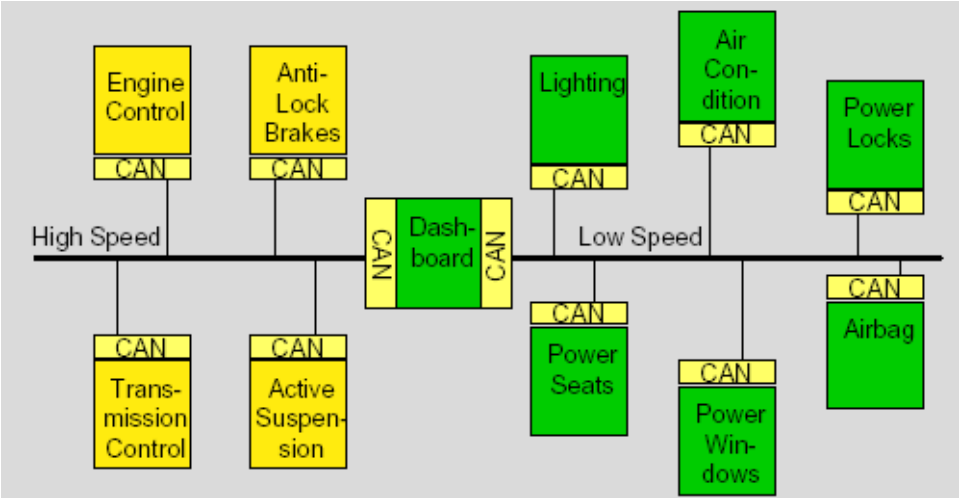


Figura 1.16: Schematizzazione di una rete CAN impiegata in campo automotive.

real-time come i sistemi di condizionamento d'aria, il sistema di comando e controllo della chiusura cetralizzata, la regolazione dei sedili ed altro ancora.

I vantaggi introdotti dall'uso di un sistema distribuito di connessione sono particolarmente evidenti nella realizzazione di autoveicoli a propulsione ibrida. In questi mezzi nei quali normalmente al classico motore a combustione viene affiancato un motore elettrico sono molti i sistemi di controllo che devono dialogare. Il sistema gestisce in modo automatico il passaggio da una forma di propulsione all'altra a seconda delle condizioni di moto e ricarica automaticamente le batterie del motore elettrico, tutto ciò contribuisce a ridurre enormemente i consumi e le emissioni.

Oggi il CAN è il protocollo principale usato in applicazioni automotive in europa e questa sua diffusione non è destinata ad arrestarsi. I moderni sistemi di diagnostica e di controllo delle emissioni ambientali la cui implementazione è obbligatoria negli Stati Uniti ormai da diversi anni, tende sempre più pesantemente a sfruttare il CAN come protocollo di basso livello in abbinamento allo standard ISO15765 nato e sviluppato appositamente come protocollo di alto livello per l'implementazione di sistemi di diagnostica basati su CAN.

Dall'automotive ai veicoli per i trasporti pubblici come le ferrovie il passo è breve. Molti treni impiegano il CAN per la connessione delle centraline di controllo dei freni e di quelle per il controllo delle porte presenti in ogni vagone.

1.4.2 Strumenti elettromedicali

La caratteristica di protocollo *safety critical* ha reso il bus CAN un sistema adatto ad essere utilizzato nella progettazione di strumentazione elettronica ad uso medico. Uno dei numerosi esempi è lo strumento denominato *Espen* ([10]) della *Plexus* impiegato per il controllo simultaneo di diverse centinaia di campioni di sangue utilizzati per culture di microrganismi e batteri. I campioni di sangue destinati alla cultura devono essere tenuti in ambienti sterili e controllati ed a tale scopo lo strumento attraverso una rete CAN si occupa del mantenimento delle condizioni ottimali per la proliferazione di queste culture batteriche. Questo strumento ma in generale qualunque altro deve essere in grado di comunicare i risultati delle proprie analisi agli operatori attraverso interfacce di comunicazione di vario genere e quindi deve essere in grado di interagire con altri sistemi ed altri strumenti. La possibilità di disporre attraverso il CAN di un sistema distribuito in cui ogni singolo nodo può lavorare in modo autonomo dagli altri ed allo stesso tempo colloquiare con gli altri dispositivi a reso il CAN la scelta vincente nella realizzazione di questo e di molti altri strumenti come ad esempio il sistema a raggi X di figura 1.17 sviluppato da Siemens e che usa CANopen come protocollo di alto livello per l'interconnessione dei dispositivi di cui è composto.

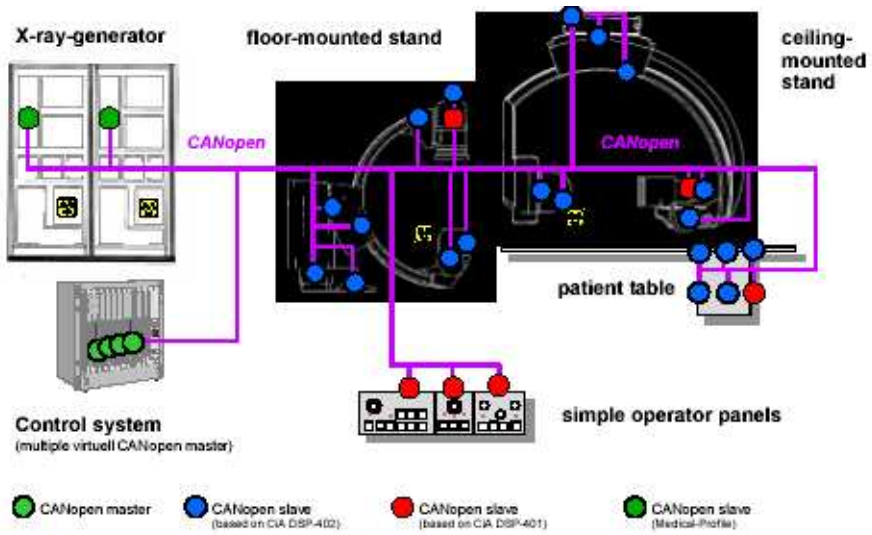


Figura 1.17: Schematizzazione din una rete CAN impiegata per la realizzazione di un sistema *X-Ray angio-biplane*.

1.4.3 Applicazioni industriali

Il confronto tra i requisiti di un bus da impiegare in un motoveicolo e quelli di un bus da utilizzare in applicazioni industriali denota somiglianze sorprendenti: basso costo, alte prestazioni in real-time, impiego in ambienti elettricamente rumorosi. Come ormai più volte detto l'impiego di reti intelligenti è scelta obbligata quando vi sia l'esigenza di connettere dispositivi di I/O come sensori ed attuatori all'interno di un macchinario o in un impianto.

Il primo impiego in campo industriale del CAN risale agli anni 90 quando venne impiegato in macchinari tessili; i primi telai equipaggiati con sistemi di controllo modulari comunicanti in tempo reale via CAN risalgono a quegli anni, oggi esiste addirittura una associazione chiamata *CAN Textile Users Group* membro del CiA che sostiene lo sviluppo del CAN in questo campo applicativo.

L'espansione del CAN in settori con caratteristiche simili come gli impianti per la lavorazione della carta e del legno è stata molto rapida. Altro grosso settore di impiego è rappresentato dall'automazione industriale mentre in campo edilizio CAN stà trovando diffusione nella realizzazione dei sottosistemi che si occupano del controllo dell'illuminazione e delle finestre, della distribuzione dell'aria condizionata e del riscaldamento, della gestione degli accessi fino ed ai sistemi di controllo degli ascensori.

Il CAN è oggi impiegato anche in applicazioni speciali come in certi telescopi, simulatori di volo ed in particolari esperimenti scientifici. Oltre a questi settori di frontiera altri campi in cui qualche applicazione inizia a nascere sono alcuni elettrodomestici di uso professionale, in controlli per pompe di carburante e in alcune grosse macchine fotocopiatrici.

Bibliografia

[1] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, third edition, 1986.

[2] Institute of Electrical and Electronics Engineers Inc. (IEEE). www.ieee.org.

[3] International Standard for Organization (ISO). www.iso.ch.

[4] CAN in Automation Users Group (CiA). www.can-cia.org.

[5] Fieldbus Specialists. *Fieldbus Specific Technical Information.* www.fieldbus.com.au/techinfo.htm.

[6] The MTL Instruments Group plc. www.mtl-inst.com.

[7] *OSEK/VDX Operating System*, 2.0 edition, 1997.

[8] CiA. *CAN Application Layer for Industrial Applications*, 1.1 edition, 1994. CiA DS 201-207.

[9] H. Boterenbrood. *CANopen high-level protocol for CAN-bus*, 2000. version 3.0.

[10] Brent Lavin. *How CAN Network fit medical devices.* http://www.medicaldesignnews.com/full_story.php?WID=5142.

Capitolo 2

Gli analizzatori di traffico

L'ampia diffusione del protocollo CAN e dei relativi HPLs language è in parte dovuta alla disponibilità sul mercato di diversi strumenti di sviluppo, configurazione e test della rete. Questi strumenti a cui ci riferiremo con il termine di "analizzatori di traffico" o *CAN Analyser* sono l'oggetto di questo capitolo; nella prima parte viene fornita una loro generica descrizione per poi nella seconda caratterizzare alcuni degli analizzatori disponibili in commercio e che rappresentano lo stato attuale dell'arte in questo settore.

2.1 Che cos'è un analizzatore di traffico

Molto semplicemente si tratta di un sistema di acquisizione dati (Data AQuisition - DAQ) che nel nostro caso è in grado di interfacciarsi con una qualunque rete che usa il CAN come protocollo di comunicazione tra i vari nodi. Da un lato lo strumento viene fisicamente connesso al bus, dall'altro attraverso un'opportuna interfaccia i dati acquisiti, elaborati ed interpretati sono inviati all'operatore, questi possono poi essere manipolati ed analizzati.

Molti strumenti non si fermano a questo, vengono impiegati come veri e propri sistemi di sviluppo dato che offrono la possibilità di inviare sul bus messaggi opportunamente formattati con cui l'operatore può testare il funzionamento della rete. Altri ancora consentono di compiere operazioni più complesse, ad esempio, permettono di descrivere per via grafica e/o tramite opportuno linguaggio le caratteristiche funzionali di un nodo. Lo strumento utilizza la descrizione di questo dispositivo virtuale per simularne le funzionalità in tempo reale, lo strumento diviene praticamente un "emulatore" che reagisce agli stimoli della rete secondo le specifiche del nodo descritto dal progettista.

Con l'utilizzo di questi strumenti lo sviluppatore di sistemi CAN

è in grado di tagliare pesantemente costi e tempi di installazione (visto che i test sulla rete possono essere condotti in modo estremamente semplice attraverso il monitoraggio dei dati che transitano sul bus) e di progettazione (il nodo in fase di design viene descritto su un semplice PC, ne viene simulato il funzionamento e spesso attraverso il PC stesso se ne può testare il comportamento in situazioni applicative reali).

2.2 DAQ e CAN in applicazioni automotive

Il settore dell'automotive è certamente quello in cui il CAN è oggi più diffuso; particolarmente in questo ambiente applicativo la combinazione del bus CAN e di un moderno tool di analisi rappresenta una scelta ricca di vantaggi significativi.

Una delle fasi fondamentali che precedono la messa in commercio di un'autovettura è il test della stessa; l'analisi accurata delle risposte del veicolo richiede l'impiego di un elevato numero di sensori e di speciali dispositivi che vengono collocati all'interno del autoveicolo, combinando l'impiego di questi dispositivi con un moderno sistema DAQ si è certi di compiere un'analisi accurata in tempo reale.

Dal punto di vista dell'acquisizione dei dati con l'impiego di un bus CAN molte delle informazioni necessarie per caratterizzare il veicolo durante il suo funzionamento sono già disponibili sul bus stesso perché fornite dai vari dispositivi ad esso collegati. Essere in grado di filtrare queste informazioni significa ridurre di molto il numero dei sensori addizionali che bisognerebbe installare durante la fase di set-up del test riducendo così anche i tempi per la realizzazione dello stesso. Il sistema DAQ è in grado di acquisire tutte le informazioni che transitano sul bus CAN (sia quelle critiche che le funzionalità secondarie), memorizza queste informazioni come se arrivassero da sensori convenzionali dedicati, le elabora e le visualizza.

Durante le fasi di test vengono usati diversi sensori come strain gauge, accelerometri, sensori di posizione, di pressione e di trazione. Al di là delle caratteristiche costruttive di questi dispositivi, un'aspetto cruciale è la scelta di un adeguato sistema di acquisizione dati. Oggi non sì può prescindere dall'usare un sistema DAQ che oltre ad avere un adeguato numero di canali A/D non sia in grado di interfacciarsi con un bus CAN in tempo reale e quindi funzioni da analizzatore di traffico.

2.3 Tipi di analizzatori

Quando si parla di questi strumenti ci si può riferire a dei *Protocol Analyzers* o a dei semplici *Physical Layer Analyser*.

Analizzatori di protocollo Implicitamente in precedenza si è sempre fatto riferimento ad un sistema di acquisizione dati che funzionasse da analizzatore di protocollo; il sistema acquisisce le informazioni che transitano sul bus senza eseguire alcun controllo a livello fisico. I dati vengono poi visualizzati ed eventualmente interpretati in base al protocollo di alto livello che l'operatore decide di scegliere e che seleziona sullo strumento.

Analizzatori di segnale [1] Verificano il segnale analogico sul bus rilevando errori come livelli di tensione non conformi, transizioni troppo lente e tutto quanto possa essere in grado di causare errori nella trasmissione dei dati per effetto della propagazione del segnale attraverso il mezzo trasmissivo.

Gli errori a livello fisico sul bus CAN non possono essere rilevati da un analizzatore di protocollo mentre un analizzatore di segnale non è in grado di interpretare i dati che transitano sul bus; i due sistemi si integrano per prestazioni e funzionalità.

> Il CAN*analyser* oggetto di questa tesi è un analizzatore di protocollo, è dunque di primaria importanza descrivere lo schema implementativo e le caratteristiche di uno strumento di questo genere.

2.4 Descrizione dello stato dell'arte

Lo schema funzionale della maggior parte degli analizzatori di protocollo oggi disponibili non si discosta molto da quanto riportato sinteticamente in figura 2.1. L'utilizzo di un semplice Personal Computer come hardware per l'interpretazione, visualizzazione ed analisi dei dati è sicuramente la scelta più economica e veloce per realizzare uno strumento di questo genere. In conclusione realizzare un analizzatore di traffico significa:

- progettare un'interfaccia che sia in grado di connettersi al bus e di tradurre le informazioni acquisite in modo tale che siano trasferibili al PC tramite una qualche porta di comunicazione disponibile sulla macchina host;

[1] Un esempio di questo tipo di oggetti è il "CANwatch" della EMS. [1]

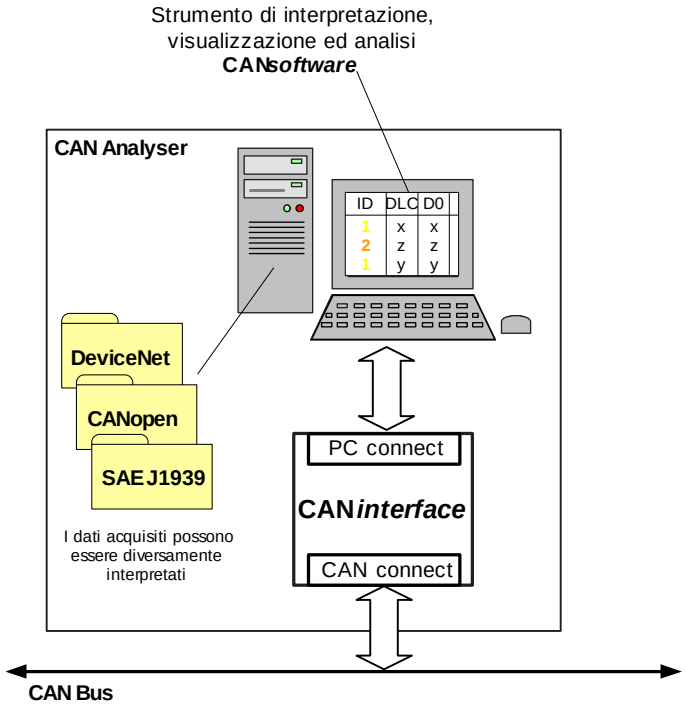


Figura 2.1: Schema funzionale a blocchi di un analizzatore di traffico per rete CAN.

- progettare un software che giri sulla macchina host e che sia in grado di comunicare con l'interfaccia CAN, di memorizzare i dati da essa ricevuti, di interpretarli e di visualizzarli. Il software deve essere altresì in grado di fornire all'operatore tutte quelle funzionalità necessarie per condurre un minimo di analisi sulle informazioni ricevute.

Da un rapido esame delle interfacce che vengono utilizzate dagli analizzatori che verranno descritti nella seconda parte di questo capitolo si ricava uno schema di principio del tutto generale per l'interfaccia con il bus (CAN Interface) comune a tutta questa serie di strumenti e che riportiamo nello schema a blocchi di figura 2.2.

- La CAN interface fornisce la connessione fisica al bus di cui l'analizzatore a bisogno, in figura 2.2 sono evidenziati i blocchi funzionali che la compongono.
  - In basso a destra è stato rappresentato un connettore attraverso cui prelevare fisicamente il segnale dal bus[2]. I connettori costi-

[2] A tale proposito esistono degli standard di connessione come quelli previsti nel CiA

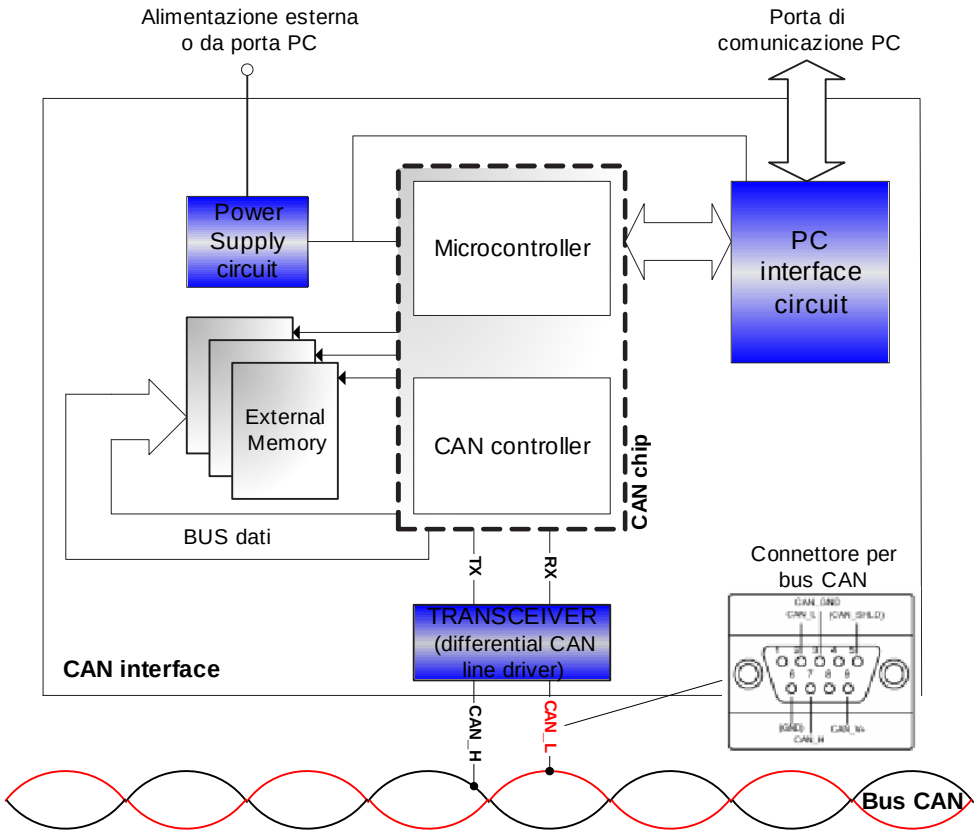


Figura 2.2: Schema funzionale a blocchi della CAN interface

tuiscono l'implementazione fisica del livello più basso (il *Medium Dependent Interface sublayer*) del modello ISO/OSI descritto in 1.1.3.

– Il *Transceiver* si occupa di gestire a livello elettrico i segnali sul bus; esegue in ricezione ed in trasmissione la conversione tra i dati che transitano serialmente sulle linee tx, rx ed i segnali elettrici adatti alla gestione differenziale della linea. Alcune interfacce impiegano più transceiver allo scopo di rendere lo strumento elettricamente compatibile con i vari standard che implementano il *Physical Medium Attachment sublayer* [3].

DS-102 che prevede l'uso di un connettore DSub9 come quello in figura di cui ovviamente fornisce lo schema di utilizzo dei pin. [2]

[3]Come detto le specifiche del protocollo CAN non si occupano della descrizione del mezzo fisico, per sopperire a questa esigenza sono nati diversi standard ognuno dei quali con proprie peculiarità ed in grado di soddisfare particolari esigenze applicative. Come conseguenza ognuno di questi standard descrive le specifiche che devono rispettare i transceiver

  - Il ruolo del *CAN controller* è quello di gestire interamente il protocollo, eseguire anche le operazioni di filtraggio dei messaggi, memorizzare i frame acquisiti ed interfacciarsi con la CPU; la sua architettura è quella mostrata in figura 2.3 e con riferimento ad essa, il *Protocol Controller* esegue le operazioni di codifica/decodifica, stuffing/destuffing dei messaggi, le operazioni di acknowledgement, di sincronizzazione e rilevazione degli errori. Per mezzo dell'*Hardware Acceptance Filter* si scarica sul CAN controller il peso di verificare l'identificatore del messaggio e riconoscerne la destinazione, mentre i *Message Buffer* sono aree di memoria integrata destinate a contenere i frame ancora da trasmettere o appena ricevuti e non ancora elaborati dalla CPU.
  - La CPU coordina l'intero sistema; esistono in commercio CPU che integrano al loro interno le funzinalità di CAN controller. La soluzione rappresentata in figura 2.2 dove CAN controller e microcontrollore fanno parte di un unico chip è la soluzione adottata nella realizzazione di CAN*analyser*[4]. La CPU oltre a controllare l'interfaccia con il bus CAN gestisce anche la comunicazione con il PC tramite il *PC interface circuit* e sfrutta la memoria esterna come polmone in cui scaricare momentaneamente i dati prelevati dalla rete.
  - Altro elemento essenziale dell'interfaccia è il circuito che gestisce la connessione con il PC. Le alternative possibili sono diverse e coprono tutte le più differenti esigenze. Dalla semplice, poco costosa e lenta RS-232, si passa alla veloce e flessibile interfaccia USB fino ad arrivare ai bus PCI o ISA. Soluzioni all'avanguardia integrano connessioni wireless tramite Bluetooth, per la gestione in remoto viene sfruttata l'interfaccia Ethernet, connessioni dedicate ai notebook si ottengono tramite PCMCIA[5].

- Il Personal Computer fornisce le risorse ed il sistema operativo su cui far girare il software che, congiuntamente all'interfaccia hardware consente l'analisi del sistema CAN. La grande potenza di calcolo e le elevate

per essere "conformi". Esempi di questo genere sono: l'ISO 11898-2 (CAN High-Speed network), ISO 11519-2 (CAN Low-Speed network), ISO 11898-3 (CAN Fault-Tolerant), ISO 11992 (Truck/Trailer), SAE J2411 (CAN Single-Wire).

[4]Nella fattispecie si tratta dell'integrato ATMEL T89C51CC01. I CAN controller che non implementano funzionalità di CPU sono detti *Stand-Alone CAN controller*, per contro esistono chip che integrano anche il transceiver e sono detti *Single-Chip Node*.

[5]Sono anche disponibili interfacce per VMEbus, PC/104, CompactPCI, GSM e soluzioni dedicate alla connessione con PLC.

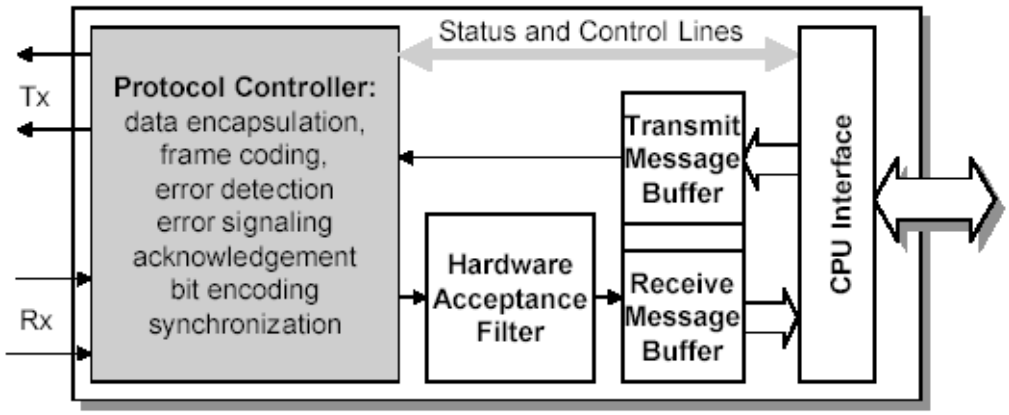


Figura 2.3: Architettura di un CAN Controller di tipo *Stand-alone*.

velocità che i moderni personal hanno raggiunto fanno si che dal lato del software di analisi i problemi implementativi risultino molto ridotti.

Possiamo a questo punto iniziare a descrivere le caratteristiche di alcuni prodotti di questo genere disponibili sul mercato.

2.4.1 Warwick Control Technology

I prodotti della famiglia *X-Analyser* [3] coprono un'ampia gamma di campi applicativi, di seguito sono riepilogate le caratteristiche di alcuni di essi.

2.4.1.1 X-Analyser

X-Analyser[6] è un software che gira su piattaforma Windows; abbinato ad una interfaccia hardware CAN costituisce un completo sistema di analisi. Le caratteristiche fondamentali di questo prodotto sono riepilogate di seguito [4].

- Simulazione di semplici nodi CAN.
- Gestisce i protocolli di alto livello: DeviceNet, CANopen e SAE J1939 per mezzo di alcuni plug-in interscambiabili ed acquistabili separatamente.

[6]Una versione dimostrativa del pacchetto software è disponibile per il download dal sito internet del produttore.

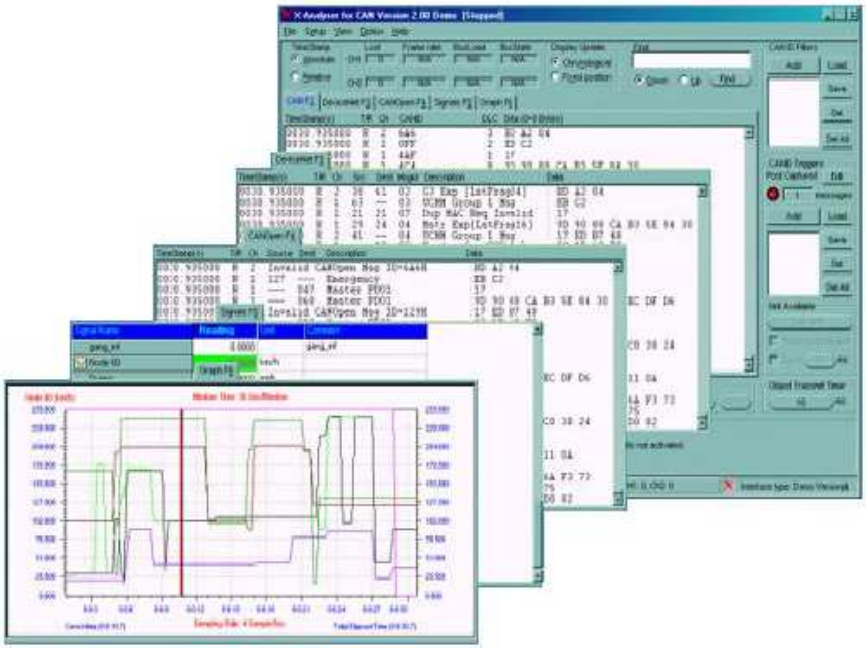


Figura 2.4: Warwick Control Technology: X-Analyser

- Immediatamente dopo la loro ricezione converte i messaggi in base alle specifiche dell'HPL selezionato.
- Consente la trasmissione di messaggi multipli in modo periodico, a comando o alla ricezione di un certo identificatore.
- Dispone di un buffer di logging dimensionabile dall'utente per migliorare le prestazioni in fase di acquisizione.
- Fornisce informazioni di vario genere: numero dei messaggi persi, frequenza di quelli arrivati (numero/s), occupazione di banda (%).
- Esegue il filtro sugli identificatori.
- Per ogni messaggio fornisce informazione di timestamp con una risoluzione di 12.8 μs.
- Riproduce in pseudo real-time i dati già acquisiti (riproduzione off-line in playback).
- Supporta la gestione di due distinti canali entrambi di tipo CAN 2.0B.
- L'interfaccia estremamente semplice ed intuitiva ne facilita l'utilizzo.

2.4.1.2 X-Analyser CANpack

Il software precedente è disponibile anche nella versione *CANpack*. Si tratta di una estensione del precedente pacchetto software sviluppato per lavorare su piattaforme portatili Compaq iPAQ che lavorano con sistema operativo *Microsoft Windows PocketPC*. Il prodotto mantiene inalterate le sue caratteristiche funzionali, le più limitate risorse hardware impongono ridotte dimensioni del buffer configurabile e come conseguenza un maggior numero di messaggi persi durante la fase di acquisizione. Se da un lato si limitano le performance dall'altro il vantaggio dell'estrema portabilità del sistema è evidente.

2.4.1.3 X-Analyser REMOTE

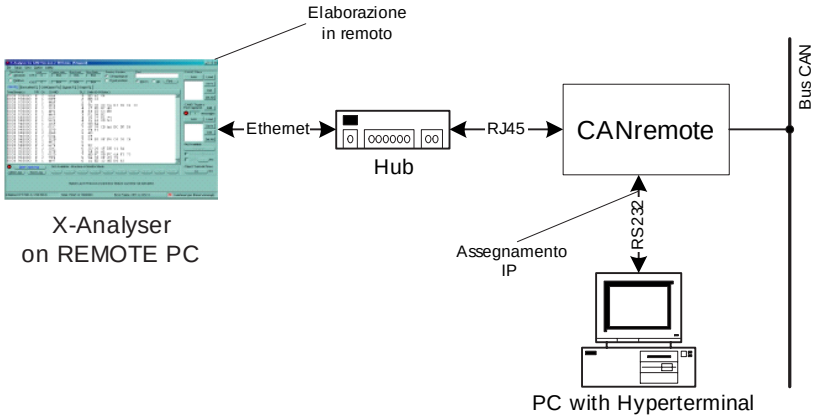


Figura 2.5: Warwick Control Technology: utilizzo di X-Analyser REMOTE per l'analisi remota di un sistema CAN.

Alle funzionalità precedenti l'X-Analyser nella versione *Remote* affianca la possibilità di compiere le analisi in remoto sfruttando l'interfaccia Ethernet di cui dispone. Il sistema è composto da una interfaccia hardware con supporto Ethernet chiamata *CAN Remote interface* ed una versione adattata del precedente software.

Caratteristiche tecniche dell'hardware:

- Porta RS-232 per la configurazione IP.
- Dispone di un solo canale CAN.
- Utilizza un Philips SJA1000 come CAN controller.

- L'interfaccia Ethernet supporta il protocollo a 10 Mbit/s.

2.4.2 Intrepid Control Systems

Intrepid Control Systems Inc. è una società americana particolarmente attiva nell'ambito della realizzazione di sistemi di sviluppo per l'elettronica veicolare, i suoi prodotti principali sono:

- *Vehicle Spy*: software per l'analisi di moltissimi protocolli di rete per la maggior parte impiegati in automotive (tra cui ovviamente CAN).
- Il supporto hardware per realizzare un completo sistema di sviluppo è fornito attraverso l'interfaccia *neoVi*.
- Per mezzo del software *Vehiclescape* fornisce una piattaforma per l'automatizzazione dei test e le simulazioni in tempo reale.

Figura 2.6: Intrepid Control Systems: Vehicle Spy

2.4.2.1 Vehicle Spy

L'elemento che contraddistingue questo software è il pieno supporto in simultanea di più protocolli: SAE J1850, CAN, ISO 9141, Keyword 2000, GMLAN, Ford UBP, SAE J1939, GM Class 2, Ford SCP e LIN. Non c'è da sorprendersi se alcuni di questi non sono stati mai menzionati fino ad ora, molti sono usati esclusivamente in ambiente automotive e proprio a questo campo applicativo si rivolgono in modo particolare gli applicativi della *Intrepid*. Dal lato funzionale il software è equivalente allo X-Analyser, da esso si distingue proprio perchè dedicato al mondo automotive e come tale fornisce pieno supporto agli standard di alto livello impiegati in questo settore. Attraverso questo software si fornisce all'ingegnere un'ambiente di sviluppo

altamente personalizzabile.

Alcune caratteristiche tecniche peculiari sono:

- L'accesso al sistema GPS (Global Positioning System).
- L'accesso a sistemi di acquisizione dati National Instruments.
- Il pannello grafico per la visualizzazione dei segnali è completamente personalizzabile (stile NI LabView).
- Consente la descrizione testuale di nodi virtuali che vengono poi fatti interagire con il sistema per mezzo di semplici script command del tipo: "Jump To Step", "Transmit Message", "Wait" ecc...
- Consente l'importazione/esportazione dei messaggi in/da un database esterno di tipo *Microsoft Access (*.mdb)*.

2.4.2.2 neoVI

neoVi [5] è uno strumento che come detto si caratterizza per fornire pieno supporto hardware di differenti tipi di rete e soprattutto l'operatività su reti differenti è garantita in modo simultaneo. Si pensi al caso in cui si debba mettere mano ad un veicolo *General Motors* che normalmente impiega per la diagnostica dell'autovettura protocolli e bus differenti come: Keyword 2000, CAN e SAE J1850; utilizzare uno strumento di questo genere abbinato ad un software come *Vehicle Spy* facilita sicuramente il compito da svolgere.

Caratteristiche tecniche:

- gestisce fino a 6 differenti reti in simultanea.
- La connessione avviene tramite un connettore DSub25 dal quale si preleva anche l'alimentazione[7].
- Dispone di un timestamp comune a tutti i network sotto test con una risoluzione che nella rete CAN è di 100μs.
- Dispone di porte di comunicazione con il PC di tipo RS232 e USB.
- Doppio buffer di trasmissione per ogni rete.

[7]Nei sistemi di diagnostica veicolare come l'OBD-II alcune linee del connettore portano la tensione a 12V della batteria che può essere utilizzata da interfacce come *neoVI* per alimentarsi.

- Funzione di Wake-up.
- Limitatamente al CAN:
  - la gestione del protocollo è garantita dal CAN controller Microchip MCP2510.
  - Velocità fino a 1 Mbit/s.
  - Gestione delle reti CAN High-Speed e CAN Medium-Speed (ISO 11898-2) per mezzo del transceiver Philips 82C251.
  - Gestione della rete di tipo CAN Low-Speed Fault-Tolerant (ISO 11898-3) per mezzo del transceiver Philips TJA1054.
  - Gestione di rete CAN Single-Wire (SAE J2411) per mezzo del transceiver Philips TLE6255.
  - Modalità di funzionamento: "Normal Mode", "Listen Only" (non sono generati errori o acknowledge) e "Loopback" (funzionamento in off-line, i messaggi trasmessi non sono inviati sulla rete ma trattati come messaggi ricevuti).
  - Possibilità di configurazione manuale dei parametri di bit timing.
  - E' garantito pieno supporto alle specifiche[8] CAN per mezzo di Banana Jacks opportuni e resistenze on-board facilmente sostituibili.
- Attraverso le *neoVI-API* [6] con cui si fornisce l'accesso all'hardware è possibile sfruttare ambienti di programmazione come Visual Basic, Visual C e C++ o ambienti di sviluppo come National Instruments LabView, LabWindows CVI, per modellare un generico nodo CAN e quindi simularne il comportamento attraverso l'interfaccia stessa. Le semplici funzioni messe a disposizione sono:
  - "OpenPort": inizializzazione del driver e creazione dell'oggetto neoVI.
  - "TxMessages": metodo per la trasmissione del messaggio.
  - "GetMessages": metodo per la lettura dei messaggi dalla rete.
  - "GetErrorMessages": metodo per la lettura dei messaggi di errore.
  - "ClosePort": chiude il collegamento con il driver.

[8] Le reti CAN rispondenti alle specifiche ISO 11898 devono essere terminate ai due estremi con resistenze da 120Ohm mentre una rete di tipo Single-Wire richiede la terminazione di ogni nodo con due resistenze il cui valore dipende dal numero di nodi presenti. Le specifiche cambiano in funzione della rete e l'interfaccia se vuole gestire reti differenti deve essere in grado di adattarsi alle differenti richieste.

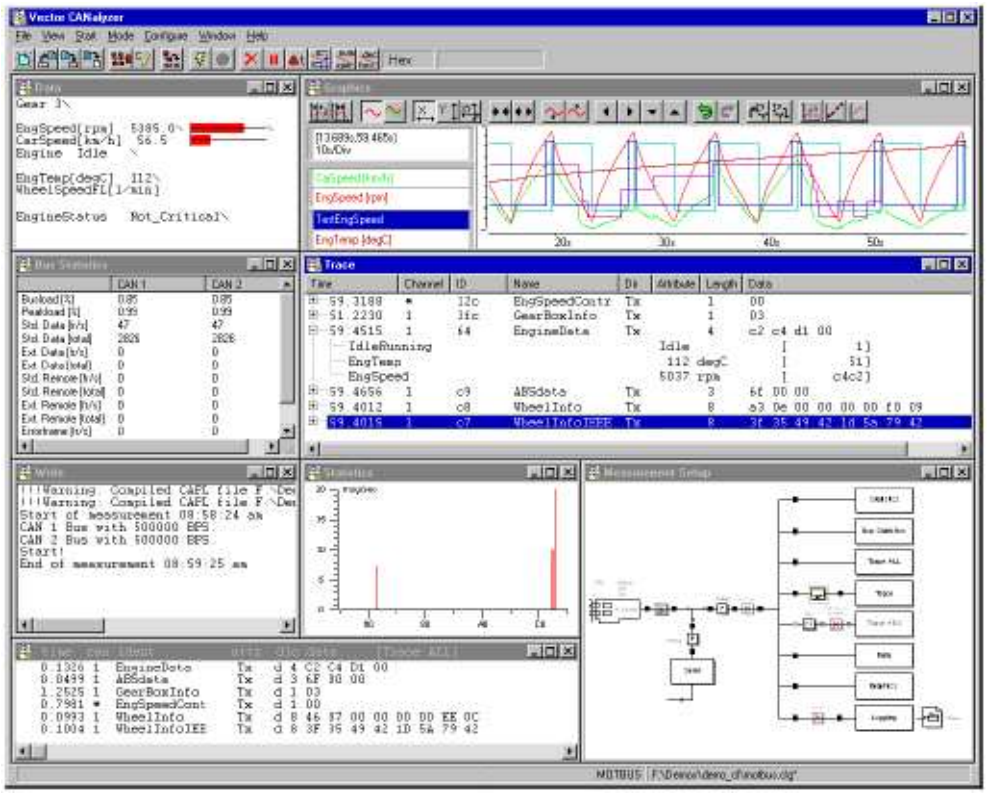


Figura 2.7: Vector Informatik CANalyser

– "FreeObject": elimina l'oggetto neoVI e lo rende disponibile ad un nuovo utilizzo.

2.4.3 Vector Informatik Gmbh

La *Vector Informatik* è un società tedesca che al pari delle precedenti è da anni impegnata nello sviluppo di tool per l'analisi per sistemi di controllo distribuiti. Il *CANalyser*[7] prodotto dalla Vector è un ambiente di sviluppo estremamente completo ed adattabile a qualunque impiego sia destinato il sistema CAN sotto analisi. Come *Vehicle Spy* che però nasce ed è sviluppato puntando al settore automotive, *CANanlyser* fornisce la possibilità all'utente di lavorare a basso livello manipolando i frame così come transitano sul bus oppure a livello applicativo dove per mezzo dei valori contenuti in un database dedicato i dati grezzi vengono convertiti. Caratteristiche principali:

- il programma gira su sistemi Microsoft Windows;

- la piattaforma di analisi viene configurata direttamente dall'utente e descritta attraverso un diagramma di flusso a blocchi; ogni blocco è chiamato finestra (window). I parametri del sistema vengono assegnati in fase di configurazione delle singole finestre.
- Nuovi blocchi possono essere definiti dall'utente attraverso un linguaggio di programmazione proprietario C-based denominato CAPL (CAN access programming language). Il software include un ambiente di editing per creare, modificare e compilare i programmi scritti in questo linguaggio.
- Consente la generazione di Error Frame.
- Attraverso i due canali di cui dispone è possibile simulare il comportamento di un nodo nella rete ed al tempo testo testarne il funzionamento, si possono sfruttare i due canali per realizzare anche un gateway tra due bus che lavorano a velocità differenti o altro ancora.
- In totale è in grado di gestire simultaneamente 32 canali (virtuali e reali); più interfacce hardware CAN possono essere gestite per inviare e ricevere messaggi.
- Include un programma denominato CANdb per la gestione del database precedentemente menzionato, attraverso cui assegnare nomi simbolici, formule, unità di misura ecc... ai messaggi CAN e ai loro Data Frame.
- Sono disponibili versioni del programma in grado di gestire i seguenti protocolli di alto livello: SAE J1939, CANopen, SDS e DeviceNet. Usando questi plug-in il linguaggio CAPL si espande e consente al sistemista di gestire in modo completo e diretto i messaggi a livello applicativo specifici dell'HLP.

2.4.4 Architettura dell'interfaccia hardware

In precedenza sono state descritte le caratteristiche dell'interfaccia *neoVi* prodotta dalla *Intrepid Systems*, allargando l'analisi ad altri prodotti vogliamo renderci conto quali sono state le scelte architetturali impiegate per la realizzazione dei dispositivi attualmente in commercio in modo tale da avere un punto di riferimento per il progetto dell'interfaccia hardware di CAN*analyser*.

Gli schemi funzionali[9] presentati in figura 2.8[8] e 2.9[9] hanno evidenti ana-

[9] Queste schematizzazioni si riferiscono ad interfacce CAN-Personal Computer prodotte dalla *ESD electronic system design Gmbh*.

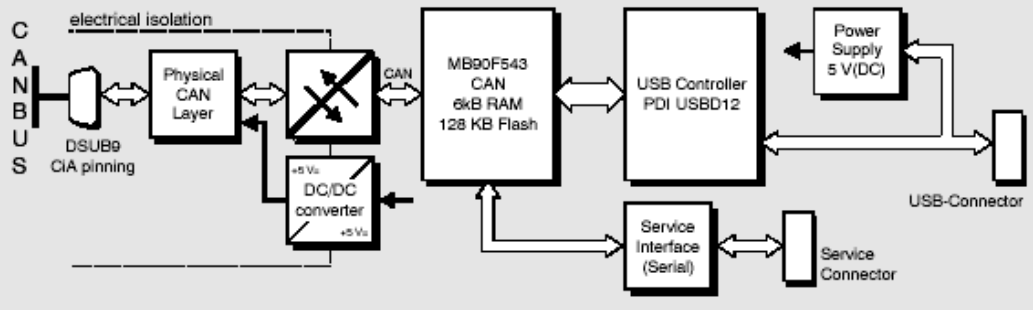


Figura 2.8: ESD: schema funzionale a blocchi dell'interfaccia "CAN-USB Mini Interface".

logie con la descrizione teorica presentata in sezione 2.4. Al di là dei circuiti per l'isolamento elettrico (optoisolatore e convertitore DC/DC) troviamo nel primo circuito un microcontrollore con integrato il CAN controller che abbinato al chip per la gestione del protocollo USB esaurisce le componenti logiche del sistema. Nel caso dell'interfaccia Bluetooth le cose si fanno un pò più complesse; la gestione del modulo Bluetooth richiede molte risorse a tal punto che la soluzione scelta dal costruttore è stata quella di dotare il modulo di un microprocessore PowerPC[10] dotato di sistema operativo embedded Linux. Il micro abbinato ad un semplice CAN controller ed a un bel pò di memoria (16 MB RAM, 2 MB Flash EPROM) sono il cuore dell'interfaccia.

> In conclusione, potenza di calcolo per gestire le porte di I/O con il PC, un CAN chip dedicato alla gestione del protocollo e memoria per poter gestire i dati anche ad elevati Baudrate sono gli elementi comuni a tutte le interfacce hardware che si pongono come obbiettivo di monitorare il traffico in tempo reale su un bus CAN.

[10]L'IBM PCC405 è un processore a 32 bits, tecnologia RISC, 375 Mips, dispone di periferiche embedded tra cui on-chip Ethernet, un motore di decompressione, un'interfaccia PCI, SRAM, interfacce di I/O multifunzionali.

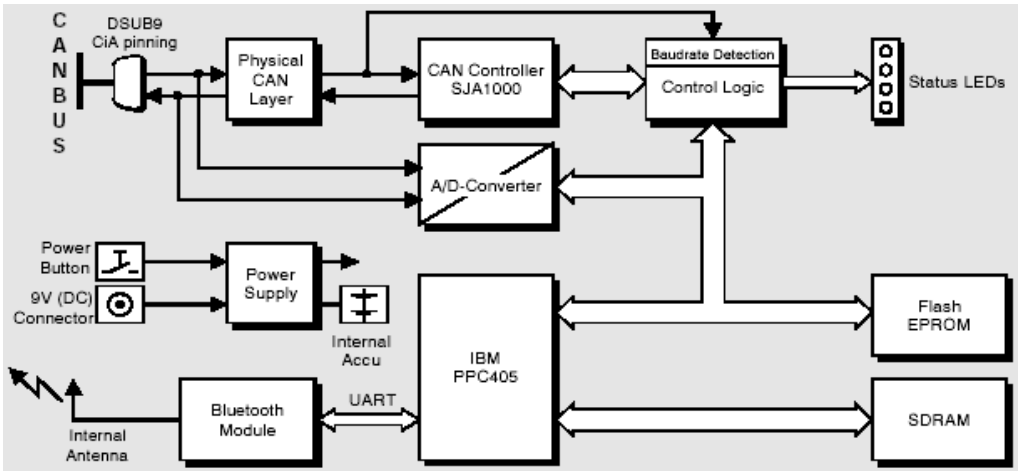


Figura 2.9: ESD: Schema funzionale a blocchi dell'interfaccia "CAN - Bluetooth".

Bibliografia

[1] EMS Dr. Thomas Wuensche. *CAN Physical Layer Analyser*. www.ems-wuensche.com.

[2] CiA. *CAN Physical Layer for Industrial Applications*, 2.0 edition, 1994. CiA DS 102.

[3] Warwick Control Technology Ltd. *Product Guide 2003 - CAN tools, Software e Hardware*, 2003. www.warwickcontrol.com.

[4] Warwick Control Technology Ltd. *X-Analyser 2.0 User's Guide*, 2001. www.warwickcontrol.com.

[5] Intrepidcs Control Technology Inc. *neoVI documentation*, 2001. www.intrepidcs.com.

[6] Intrepidcs Control Technology Inc. *intrepidcs API documentation*, 2001. www.intrepidcs.com.

[7] Vector Informatik Gmbh. *CANalyser manual*, 4.0.2 edition, 2003. www.vector-informatik.de.

[8] ESD electronic system design Gmbh. *CAN-USB Mini-Interface*. www.esd-electronics.com.

[9] ESD electronic system design Gmbh. *Intelligent CAN-Bluetooth-Interface*. www.esd-electronics.com.

Capitolo 3

Informazioni preliminari

In questo capitolo vengono recuperati i concetti base del protocollo CAN limitando però i dettagli alle caratteristiche che si ritengono di una certa importanza per comprendere meglio il funzionamento del CAN*analyser*. Nella seconda parte viene introdotto il sistema di sviluppo utilizzato per il progetto dell'analizzatore e vengono giustificate alcune delle scelte progettuali hardware.

3.1 Introduzione al CAN

Il Controller Area Network (CAN) è un protocollo di comunicazione seriale in grado di gestire con elevata efficenza sistemi di controllo distribuiti di tipo real-time con un elevato livello di sicurezza e di integrità dei dati trasmessi.

Una rete CAN nella sua versione più tradizionale[1] è composta da un bus lineare realizzato con una coppia intrecciata di fili (schermati o meno) e da un numero teoricamente infinito di interfacce (nodi) ad esso collegati e che per mezzo di esso dialogano tra di loro (figura 3.1).

Le specifiche CAN nella versione 1.2 e la successiva 2.0, come già introdotto nel paragrafo 1.1.3, forniscono una descrizione del protocollo limitatamente ad una parte dei due layer più bassi del modello ISO/OSI descritto in precedenza e cioè il Data Link ed il Physical Layer.

[1]Come detto le specifiche del protocollo CAN non si occupano della descrizione del mezzo fisico, per sopperire a questa esigenza sono nati diversi standard ognuno dei quali con proprie peculiarità ed in grado di soddisfare particolari esigenze applicative. Uno degli elementi distintivi tra uno standard e l'altro è il mezzo trasmissivo impiegato per la realizzazione fisica della linea: coppie intrecciate di conduttori, bus a singolo conduttore, fibre ottiche, sistemi wireless via radio e infrarossi ecc... In ogni caso si tratta di un bus di tipo lineare.

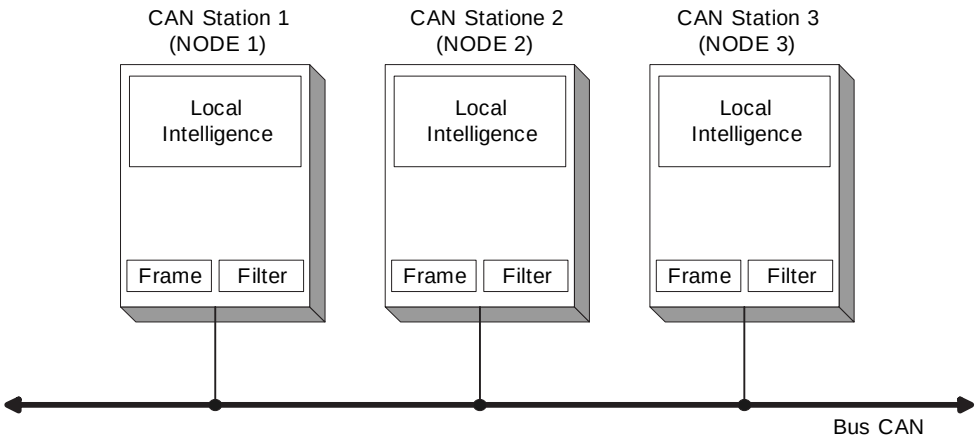


Figura 3.1: Struttura schematica di una rete CAN.

3.1.1 Accesso al bus e struttura dei frame

Il CAN[2] appartiene alla categoria dei protocolli *producer-consumer* in cui tutti i nodi della rete possono teoricamente essere in grado di inviare dati sul bus e tutti possono ricevere qualunque messaggio (si parla anche di *broadcasting* dei messaggi).

Non si può dire che nodi abbiano un indirizzo specifico[3] (per questo il CAN è definito un protocollo *Message-based* anzichè *Address-based*) è invece più corretto affermare che l'informazione di indirizzo è contenuta nel campo *identifier* del frame trasmesso e che questo identificatore viene utilizzato da tutti i nodi della rete per controllare se i dati trasportati dal frame sono o meno di loro competenza (sono utili o meno).

Non si può propriamente parlare di indirizzo perché più interfacce distinte possono riconoscere valido l'identificatore trasmesso e poi il campo *identifier* viene utilizzato anche per stabilire la priorità di trasmissione del messaggio sul bus.

[2]Convenzionalmente nel resto del capitolo un bit con livello logico 1 verrà indicato come bit di tipo "recessivo" ed indicato con "r"; un bit con livello logico 0 è detto bit dominante e verrà indicato con il carattere "d".

[3]Il CAN definisce le specifiche di basso livello del protocollo di comunicazione e lascia liberi di definire un HPL adatto alle specifiche esigenze del progettista. E' possibile definire degli HPL nei quali i nodi abbiano un proprio indirizzo o modellare il CAN in modo che lavori come un protocollo master-slave anziché multicasting come sarebbe sua natura. In questo capitolo ci riferiamo alle caratteristiche del protocollo CAN e non ai suoi HP language.

3.1.1.1 Struttura dei frame

Il campo *identifier* appena introdotto è una parte del così detto *Arbitration Field* di un frame di tipo DATA o REMOTE la cui struttura è riportata nel dettaglio nella figura 3.2 relativamente al protocollo standard (CAN 2.0A) ed in figura 3.3 dove i campi si riferiscono a quanto descritto nelle specifiche CAN 2.0B (protocollo esteso). Il CAN definisce cinque tipi di messaggi detti anche frame:

DATA frame [4]: è il messaggio di tipo più comune ed è usato per trasmettere dati da un nodo ad un altro. Il frame inizia con un bit *Start of frame* usato per la sincronizzazione (hard synchronization) dei nodi e di seguito vengono trasmessi i campi di seguito elencati.

- L'*Arbitration Filed* contiene le informazioni necessarie durante la fase di arbitraggio per assegnare il possesso del bus quando questo viene conteso da più nodi. Questo campo a dimensioni differenti a seconda che il frame rispetti le specifiche 2.0A (standard frame) o 2.0B (extended frame) ma indipendentemente da questo, contiene l'identificatore (11 bit per un frame standard, 29 bit per un frame extended) e l'*RTR (Remote Transmission Request*) bit con cui si distingue un DATA Frame da un REMOTE Frame.
- Il *Control Field* è composto da 6 bit, gli ultimi 4 in ordine di trasmissione costituiscono il campo *DLC (Data Length Code*) che codifica il numero di byte di dato contenuti nel messaggio (0-8 byte). Se il frame è di tipo standard il primo bit del campo di controllo è il bit di *IDE (Identifier Extention)* che specifica proprio il tipo di formato (IDE=d standard, IDE=r extended), questa informazione se il frame è di tipo esteso è contenuta nell'*Arbitration Field* descritto in precedenza.
- Il campo *Data Field* contiene i byte che codificano l'informazione trasmessa con il messaggio. Il campo può avere un'estensione che varia da 0 a 64 bit (solo multipli di 8) in base a quanto specificato nel campo DLC. I bit che compongono i singoli byte del campo vengono trasmessi partendo sempre dal più significativo.
- Il *Cyclic Redundancy Field (CRC Field)* viene impiegato come sistema (non unico) per la rilevazione degli errori di trasmissione ed è composto dalla *CRC sequence* (15 bit calcolati sulla base dei precedenti campi del frame) e da un bit recessivo, il *CRC Delimiter*

[4] I bit SRR (Substitute Remote Request), RB1 e RB0 (Reserved Bit) non vengono descritti perché sono risevati dal protocollo ed hanno valore costante.

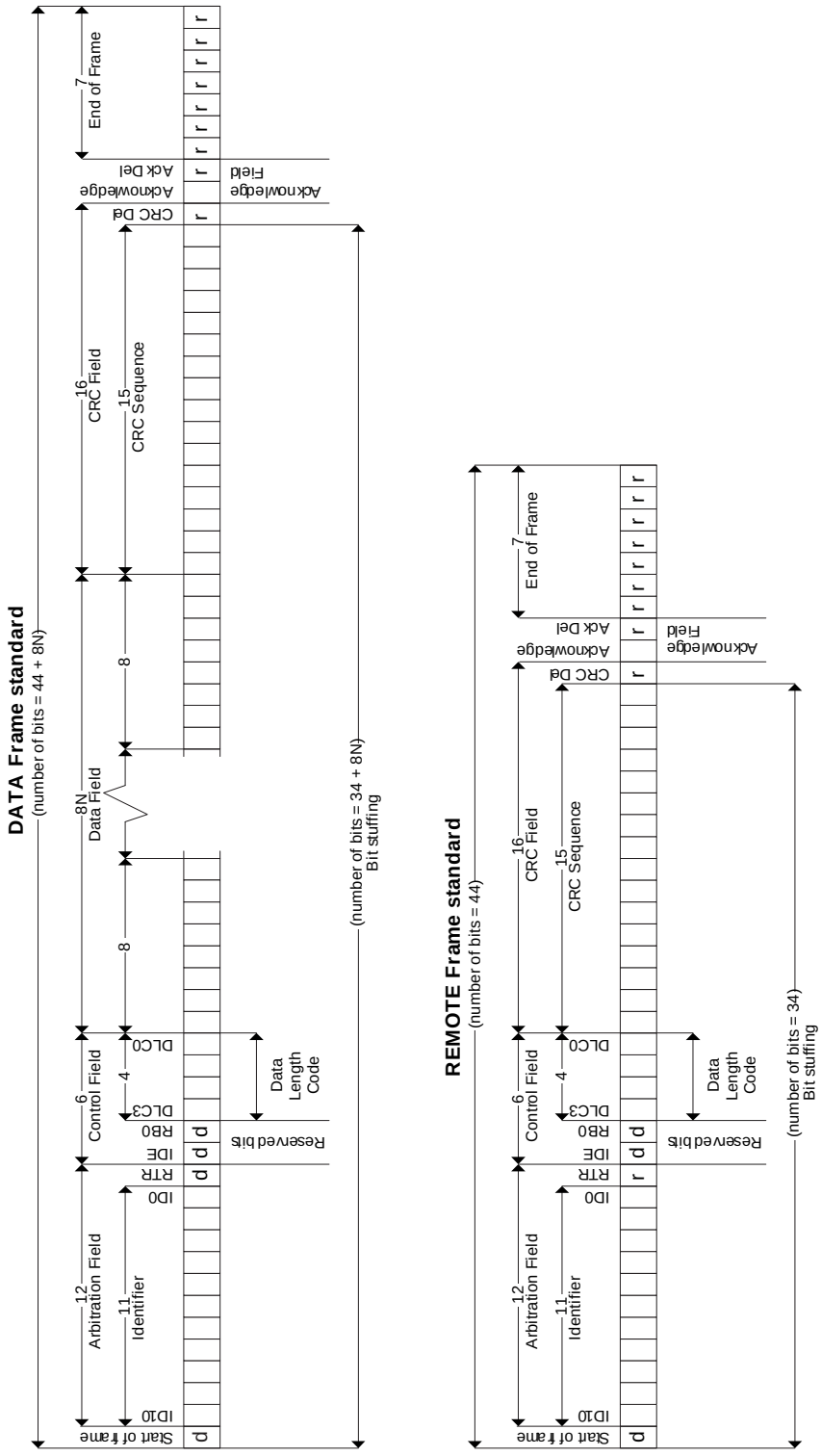


Figura 3.2: Formato dei DATA e REMOTE frame - Formato standard (CAN1.2).

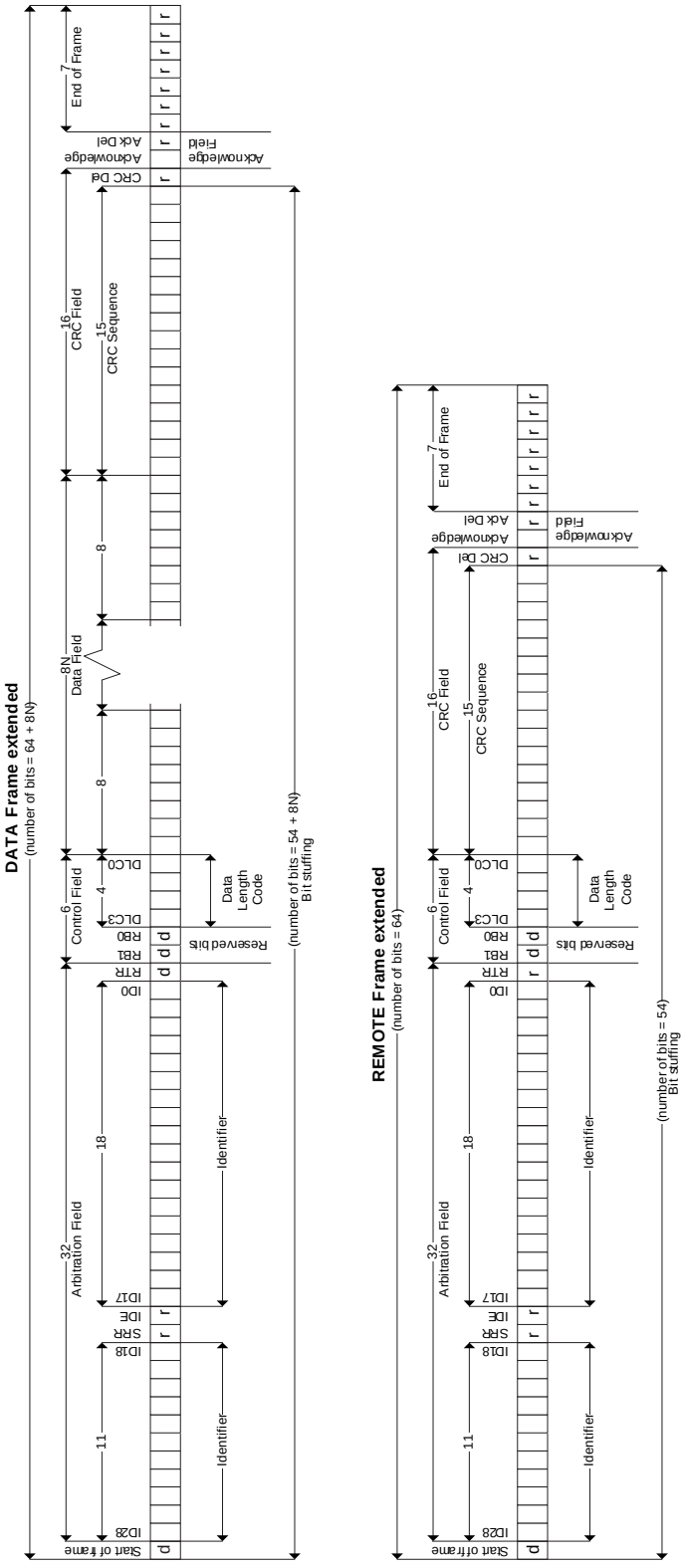


Figura 3.3: Formato dei DATA e REMOTE frame - Formato esteso (CAN2.0).

bit utilizzato per chiudere la trasmissione del CRC ed essere certi che il bus una volta trasmessi i 15 bit sia in stato recessivo e quindi pronto a gestire correttamente la fase di acknowledgment che seguente

- Il campo successivo è l'*Acknowledge Field*. Durante l'*ACK Slot* bit il nodo trasmittente invia un bit recessivo (r), qualunque nodo della rete abbia ricevuto il messaggio formattato in modo corretto risponde inviando un bit dominate (d) che "sovrascrive" il precedente. Il nodo trasmittente riceve in questo modo l'ack cioè la conferma dell'avvenuta corretta ricezione da parte di almeno un nodo e chiude la trasmissione con un bit recessivo che costituisce l'*ACK Delimiter*.
- Il DATA frame si chiude con il campo di *End Of Frame* (7 bit recessivi).

REMOTE frame : Una delle caratteristiche del protocollo CAN è che un nodo della rete non solo può trasmettere informazioni o rimanere in attesa di riceverne altre, può anche "chiedere" informazioni ad altri nodi della rete "presentando una domanda". Per mezzo del REMOTE frame il nodo trasmette una richiesta remota cioè una *Remote Transmit Request (RTR)*.

Un REMOTE frame ha la stessa struttura di un DATA frame con le seguenti differenze:

- il bit RTR è dominante per un DATA frame e recessivo per un REMOTE frame (per mezzo di esso il ricevitore riconosce il tipo di messaggio e si comporta di conseguenza);
- non esiste il campo DATA (non ha senso prevederlo);
- il campo DLC non codifica il numero di byte componenti il campo DATA ma può essere impiegato per specificare altre informazioni: il tipo di dato o il numero di informazioni che il nodo richiede.

ERROR frame : è generato da ogni nodo che rileva un errore[5]. Come mostrato in figura 3.4 il frame di errore è composto da due campi chiamati *Error Flag Field* e *Error Delimiter Field*, quest'ultimo è composto da 8

[5]Gli errori cui il protocollo si riferisce sono i: *Bit Error, Stuff Error, Form Error* e *Ack Error* al rilevamento dei quali viene subito inviato un ERROR Frame. Se viene rilevato un *CRC Error* il frame viene inviato anche in questo caso ma solo successivamente all'*ACK Delimiter*.

bit recessivi a seguito dei quali i nodi possono riprendere le comunicazioni sul bus. La struttura del campo Error Flag dipende dallo stato[6] in cui si trova il nodo al momento in cui il frame di errore deve essere generato.

- Se il nodo è *Error Active* l'Error Flag è composto da 6 bit dominanti. Questa sequenza a seconda del momento in cui viene trasmessa viola le regole di bit stuffing oppure la struttura del campo di Acknowledgement o ancora quella del campo di End Of Frame del messaggio che gli altri nodi della rete stanno decodificando. Indipendentemente da quando l'errore si verifica l'ERROR Frame viola qualche regola di formato e come conseguenza anche gli altri nodi rilevano a loro volta un errore e trasmettono un ERROR Frame, i vari Error Flag si sovrappongono sul bus producendo un "echo" che ha come effetto quello di allungare la permanenza della sequenza di bit dominanti; l'Error Flag che ne deriva può essere composto da un numero di bit dominanti che può variare tra 6 e 12.
- Se il nodo è *Error Passive* al rilevamento dell'errore il campo Error Flag prodotto è composto da 6 bit recessivi; il frame complessivamente è costituito da 14 bit recessivi. Come conseguenza l'ERROR Frame trasmesso da un nodo "passivo" non ha effetto sugli altri moduli della rete (l'informazione di errore trasmessa non è considerata attendibile perché generata da una stazione CAN "passiva" quindi con qualche "problema"). Solo se il nodo *Error Passive* è in trasmissione ed esso stesso rileva l'errore, l'Error Flag di tipo passivo viola le regole di bit stuffing e quindi l'errore viene rilevato anche dagli altri nodi, solo in questo caso un ERROR Frame di tipo passivo ha effetto sulle altre stazioni della rete.

INTERFRAME Space : con questo termine non si indica un vero e proprio frame ma solo una sequenza di bit recessivi (minimo 3 che costituiscono l'*Intermission Field*) che separano un frame qualunque da un altro frame di tipo DATA o REMOTE. Successivamente alla trasmissione dell'*Intermission Field* la linea del bus rimane in stato recessivo (Bus Idle) fino alla trasmissione di un nuovo messaggio. Dopo la trasmissione dell'*Intermission Field* il bus è considerato disponibile.

[6]Successivamente si avrà modo di vedere che i nodi di una rete CAN possono trovarsi in tre distinti stati detti: *Error Active, Error Passive* e *Bus-Off* e si vedrà in quale maniera queste condizioni vengono raggiunte.

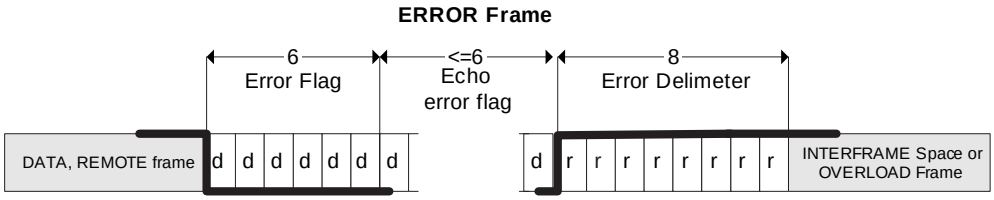


Figura 3.4: Struttura dell'ERROR Frame.

Un'eccezione a quanto appena descritto è mostrata in figura 3.5-B: nel caso in cui l'ultimo nodo trasmittente sia di tipo Error Passive all'*Intermission Frame* segue una sequenza di 8 bit recessivi detta *Suspend Transmission segment* che allunga i tempi minimi di trasmissione di un nuovo messaggio da parte dello stesso nodo, ciò consente ad altre interfacce anche con identificatori con priorità inferiore di iniziare una nuova trasmissione prima del nodo passivo, così facendo il protocollo evita che un modulo ad altissima priorità ma difettoso possa rischiare di monopolizzare la rete.

OVERLOAD Frame : questo frame ha la stessa struttura di un ERROR Active Frame tuttavia *può essere generato solo durante un INTERFRAME Space* e questa è anche l'unica maniera attraverso la quale i due messaggi possono essere distinti.

Rispetto ad un ERROR Frame, l'OVERLOAD *non incrementa i contatori TEC e REC e non causa la ritrasmissione dei messaggi*, il messaggio ricevuto prima di un OVERLOAD Frame è un messaggio valido a differenza di quello che precede un ERROR Frame.

L'OVERLOAD viene trasmesso al verificarsi di una delle seguenti condizioni:

1. il nodo per qualche motivo non è in grado di riabilitarsi in tempo per una nuova ricezione dopo che la precedente si è conclusa;
2. viene rilevato un bit dominante durante la trasmissione dell' *Intermission Frame*[7]. Se il bit dominante è l'ultimo (il terzo) viene interpretato come uno *Start of Frame*[8].

[7]Questa situazione non viene rilevata da alcuna delle condizioni di errore che sono successivamente descritte; non determina la generazione di un Form Error e quindi tantomeno l'invio di un ERROR Frame.

[8]Dato che la regola di bit stuffing viene applicata a partire dal bit di SOF, il sesto bit dell'Overload Flag viola lo stuffing e come consegua i nodi attivi genereranno un Error Active Flag.

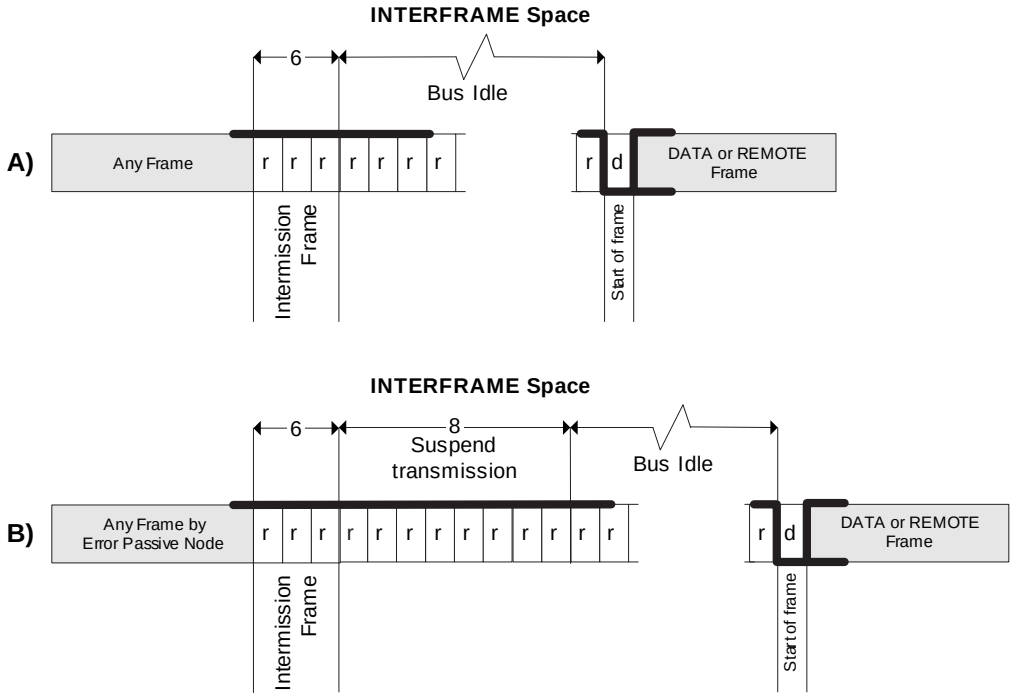


Figura 3.5: Struttura dell'INTERFRAME Space.

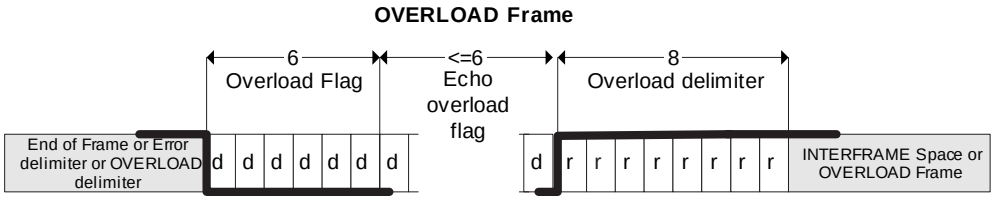


Figura 3.6: Struttura dell'OVERLOAD Frame.

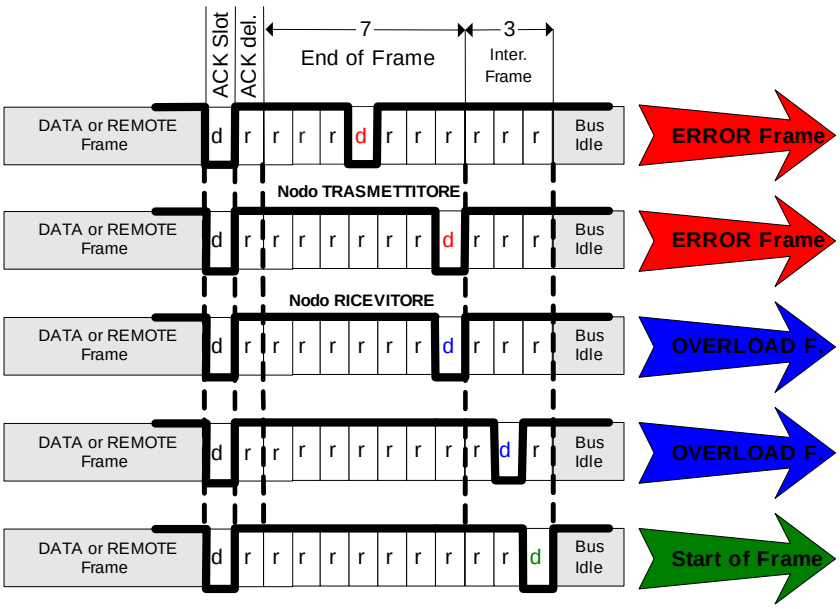


Figura 3.7: Comportamento del nodo CAN al rilevamento di errori nei campi di EOF e Intermission Space.

3. Viene generato anche se un nodo campiona un bit dominante come ultimo bit di un Error Delimiter o di un Overload Delimiter[9].
4. Se un nodo in ricezione rileva un bit dominate come ultimo bit di un *End of Frame* riconosce il messaggio come valido. Il bit dominante ricevuto è interpretato appartenente all'*Intermission Frame* e quindi il nodo si trova nella condizione descritta nel punto 1 (figura 3.7).

La struttura dell'OVERLOAD è analoga a quella dell'ERROR Frame e come tale la condizione di overload viene notificata a tutti i nodi della rete che a loro volta generano l'echo mostrato in figura 3.6. L'obbiettivo che il nodo generatore dell'OVERLOAD Frame raggiunge è quello di ritardare l'istante in cui il bus tornerà in condizione di idle e quindi sarà nuovamente disponibile a gestire altro traffico[10].

3.1.1.2 CSMA/CD

Il CAN è un protocollo CSMA/CD (Carrier Sense Multiple Access/Collision Detection), ciò significa che i nodi della rete devono monitorare il bus atten-

[9] Anche questa situazione non viene rilevata da alcuna condizione di errore.
[10] Un nodo può trasmettere al massimo due frame consecutivi di questo tipo.

dendo che questo si porti in stato di idle prima di tentare di trasmettere un messaggio (Carrier Sense). Quando il bus è in idle ogni nodo può tentare di iniviare le proprie informazioni impossessandosi del bus stesso (Multiple Access). Se due nodi iniziano la trasmissione nel solito momento questi rileveranno una "collisione" e si comporteranno di conseguenza (Collision Detection).

In modo particolare il protocollo CAN per decidere quale nodo deve avere la priorità nella trasmissione adotta un metodo di arbitraggio di tipo *bitwise* (a livello del singolo bit) non distruttivo, in altri termini il messaggio inviato dal nodo che "vince" l'arbitraggio non viene alterato e quindi non deve essere ritrasmesso con evidenti vantaggi: non ci sono tempi morti ed i messaggi ad alta priorità possono essere trasmessi senza alcun ritardo dovuto alla fase di contesa del bus.

3.1.1.3 Non-destructive bitwise arbitration

Per poter supportare un arbitraggio di tipo *non-destructive bitwise* un protocollo deve definire due stati del bus corrispondenti ai due livelli logici 0 e 1 in cui uno stato sia in grado logicamente e quindi anche elettricamente di sovrascrivere sempre l'altro; per questo motivo il livello logico che viene sovrascritto prende il nome di stato recessivo mentre l'altro è detto dominante.

Il protocollo CAN definisce il livello logico 0 (livello basso di tensione) come dominante (dominant bit = d) ed i livello logico 1 (livello alto di tensione) come recessivo (recessive bit = r). I due *bus state* così definiti sono gestiti dai nodi sul bus con il meccanismo del *Wired-AND* con la conseguenza che il bit dominante (0 logico) sovrascrive sempre quello recessivo (1 logico), solo se tutti i nodi della rete trasmettono un bit recessivo anche il bus si troverà in quello stato.

Con questo meccanismo di pilotaggio illustrato in figura 3.8 ogni nodo della rete che inizia a trasmettere il messaggio è in grado di riconoscere "leggendo[11]" lo stato del bus, se altri nodi con identificatore a più elevata priorità hanno contemporaneamente iniziato una trasmissione. Se un nodo durante la trasmissione del *Arbitration Field* invia un recessivo e legge il bus in stato dominante riconosce che c'è almeno un altro nodo che stà trasmettendo un messaggio a più elevata priorità e come conseguenza termina la trasmissione ed entra in modalità di "listening" (vedi figura 3.9). Un nodo che perde l'arbitraggio prova a ritrasmettere il proprio messaggio appena il bus si li-

[11]Mentre un nodo trasmesse è allo stesso tempo in grado anche di leggere lo stato del bus. E' attraverso questo meccanismo che un nodo *Producer* rileva un eventuale errore in fase di trasmissione (Bit Error).

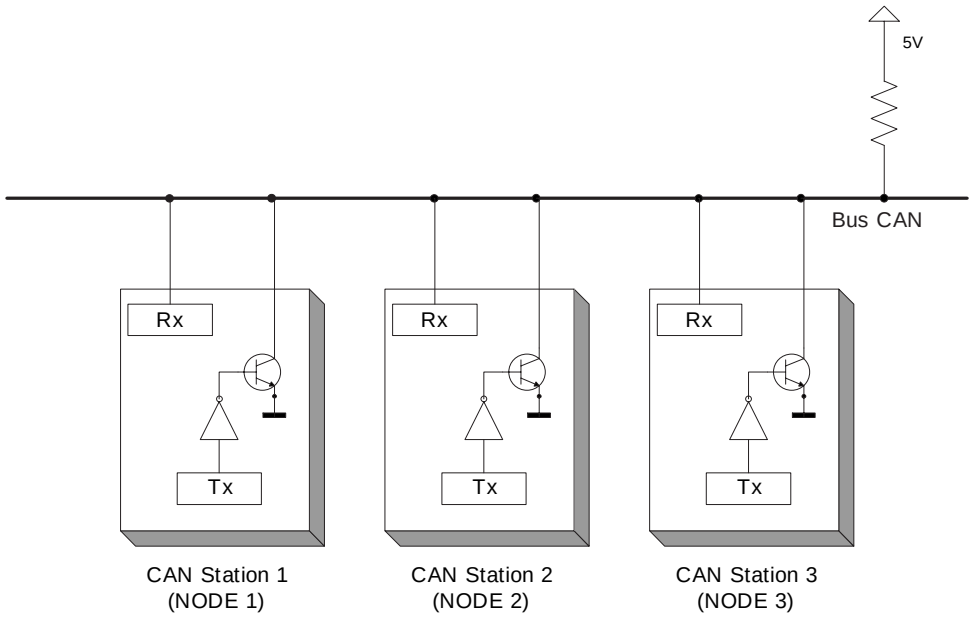


Figura 3.8: Meccanismo del Wired-AND.

bera nuovamente (dopo l'*Intermission Field* parte subito una nuova fase di arbitraggio del bus).

L'arbitraggio viene vinto dal noto che trasmette il messaggio a più elevata priorità e cioè quello con identificatore più piccolo[12].

> Il CAN è un protocollo message-based e questa sua caratteristica offre diversi vantaggi in molti campi applicativi come quelli automotive: un sensore per airbag può inviare informazioni alla centralina che gestisce il sistema di sicurezza dell'autovettura, questa a sua volta può inviare questa ed altre informazioni contemporaneamente ad altri nodi. Un altro vantaggio è il fatto che un nodo aggiuntivo può essere inserito nella rete senza il bisogno di riprogrammare le altre interfacce: il nodo legge i dati che transitano sul bus e decide se utilizzarli o meno. Per mezzo dei REMOTE Frame la stessa centralina può richiedere periodicamente ad altri nodi (sensore di pressione delle gomme, tensione di batteria ecc...), che a differenza del sensore degli airbag non trasmettono continuamente messaggi, l'aggiornamento dei dati. Il progettista può utilizzare questa caratteristica per minimizzare il traffico sulla rete limitandolo ai dati strettamente necessari.

[12]Il messaggio con ID=0 è quello a priorità massima.

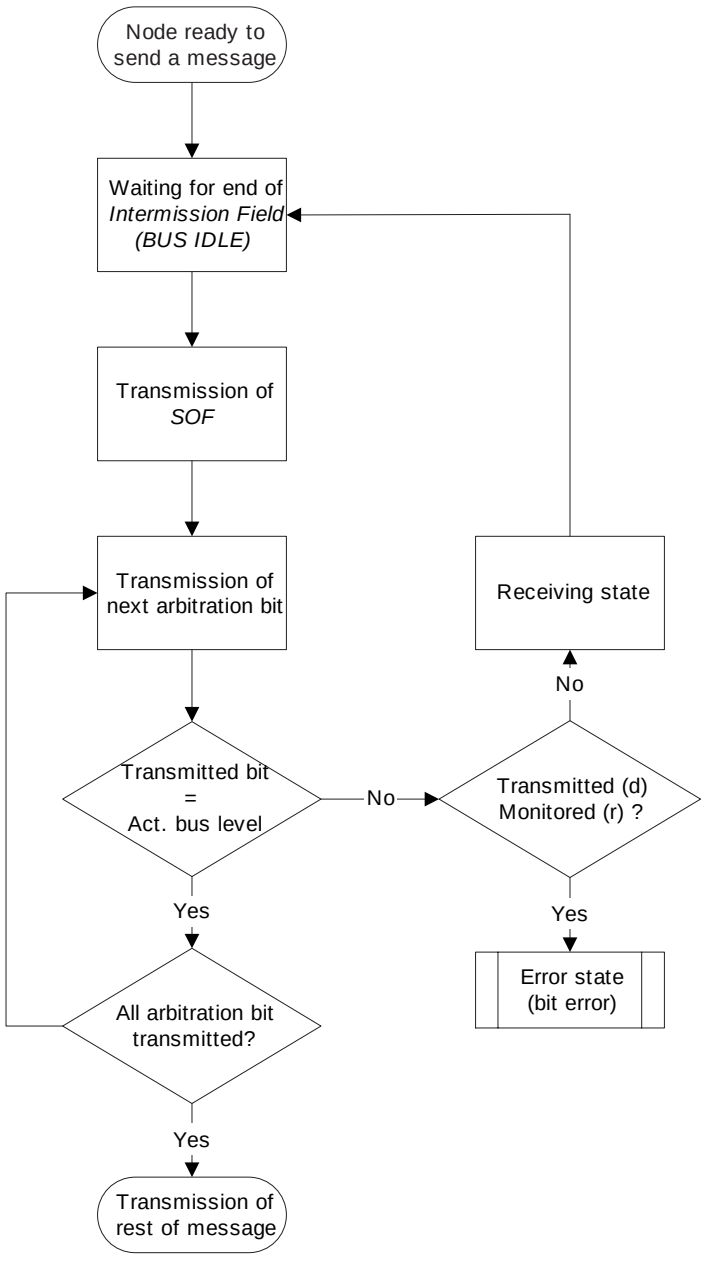


Figura 3.9: Diagramma a blocchi esplicativo delle operazioni eseguite da ogni nodo CAN durante la fase di arbitraggio del bus.

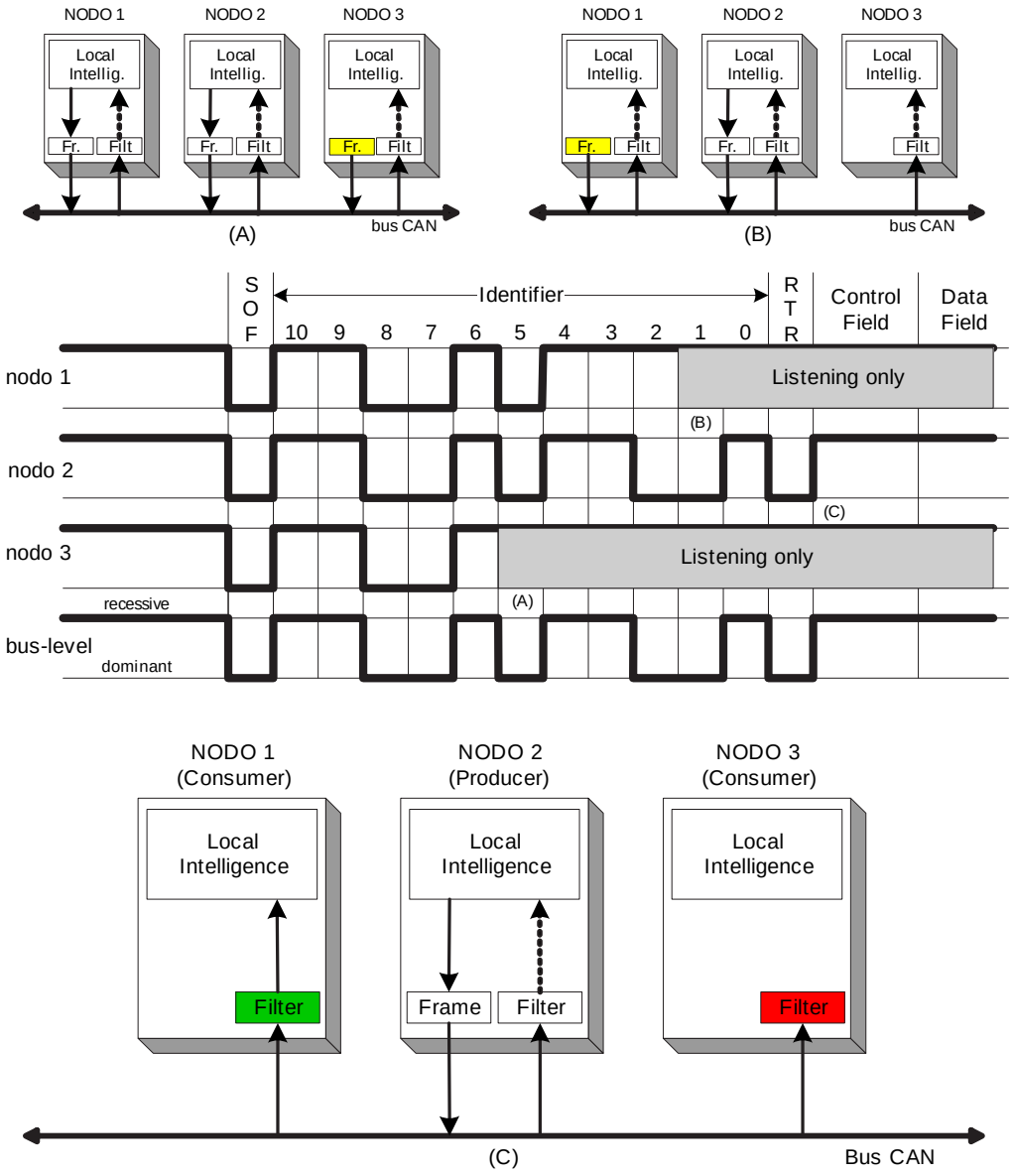


Figura 3.10: Meccanismo dell'arbitraggio *bitwise non distruttivo*: (A) I nodi 1, 2 e 3 iniziano contemporaneamente la trasmissione del messaggio, il nodo 3 rileva un bit dominante, sospende la trasmissione ed entra in ricezione. (B) Anche il nodo 1 entra in ricezione. (C) Il nodo 2 vince l'arbitraggio e *continua* l'invio del frame, il nodo 1 filtra l'ID e lo accetta, il nodo 3 lo rifiuta e non elaborerà i dati trasmessi da 2.

3.1.2 Sequenza CRC

Come detto il *CRC Field* è composto da 16 bit, i primi 15 costituiscono la sequenza CRC, l'ultimo è un bit recessivo chiamato *CRC Delimiter*.

Il *Cyclic Redundancy Code o codice polinomiale (CRC)* è una tecnica di codifica dei bit che consente di ottenere un elevato grado di protezione del messaggio spedito (è in grado di rilevare un'elevata percentuale di errori) usando una ridondanza non troppo elevata, dove per ridondanza si indica il numero di bit di controllo (*Check-bit*) da aggiungere al messaggio originale (nel caso del CAN i bit sono 15) per ottenere questa protezione. Il CRC è un codice di rilevamento degli errori e non di correzione a differenza di altri algoritmi come il *codice di Hamming*.

3.1.2.1 Concetti base

In apparenza si tratta di un meccanismo abbastanza complesso ma si basa su meccanismi algebrici (Teoria Algebrica dei Campi) piuttosto semplici da implementare via hardware e che di seguito si riepilogano:

- addizioni e sottrazioni sono eseguite *modulo 2* cioè sensa riporti per l'addizione e prestiti per la sottrazione. Somme e sottrazioni tra sequenze di bit si eseguono facendo uno XOR;
- la divisione viene effettuata come in binario ma con sottrazioni *modulo 2* (XOR).

L'algoritmo considera i bit in arrivo dalla rete come i coefficienti di un polinomio di grado $(K-1)$ fatto in questo modo:

$$M(x) = m(K-1)x^{(K-1)} + m(K-2)x^{(K-2)} + \cdots + m(2)x^2 + m(1)x + m(0)$$

dove:

K : è il numero di bit che compongono la parte del messaggio sottoposto a codifica CRC;

m(i) : è il coefficiente di grado i e vale 0 o 1, corrisponde al valore del bit in posizione i-ma della sequenza in arrivo.

Ad esempio una sequenza di bit:

101101

corrisponde al polinomio:

$$M(x) = x^5 + x^3 + x^2 + 1$$

Viene poi definito un secondo polinomio detto *generatore* $G(x)$ di grado $(R < K)$ e struttura prefissata, noto sia all'interfaccia trasmittente che al ricevitore; ad esempio potrebbe essere fatto in questo modo:

$$G(x) = x^4 + x + 1$$

corrispondente alla sequenza binaria:

$$10011$$

La tecnica CRC si preoccupa di calcolare una sequenza di R bit di controllo da accodare al messaggio originale ottenendo così una sequenza di $K + R$ bit detta *codeword* in modo tale che il polinomio corrispondete $T(x)$ sia divisibile per il polinomio generatore $G(x)$. Se uno o più errori di trasmissione modificano la parola di codice il polinomio corrispondente non sarà più divisibile, con elevata probabilità, per il polinomio generatore $G(x)$ e l'errore verrà rilevato.

3.1.2.2 Calcolo del CRC

Di seguito viene descritta la procedura di calcolo.

1. Il polinomio "originario" $M(x)$ viene moltiplicato per x^R, l'operazione equivale ad accodare alla sequenza del messaggio da codificare un numero R di bit a 0.

   $$101101\mathbf{0000}$$

2. Si divide $x^R M(x)$ per $G(x)$ come mostrato in figura 3.11.

3. Il resto della precedente divisione viene sottratto a $x^R M(x)$:

   $$1011010000 \; XOR \; 000000\mathbf{1110} = 1011011110$$

   ottenendo il polinomio $T(x)$ i cui coefficienti costituiscono la sequenza di bit da trasmettere:

   $$101101\mathbf{1110}$$

   dove i bit evidenziati sono la sequenza CRC.

Il ricevitore si costruisce il polinomio $T'(x)$ con i $K + R$ bit che riceve, lo divide per il generatore $G(x)$ ed accetta il messaggio solo se il resto della divisione è zero.

Il calcolo della sequenza CRC può essere implementato via software e/o hardware mediante operazioni/registri di shift e attraverso operazioni/porte XOR.

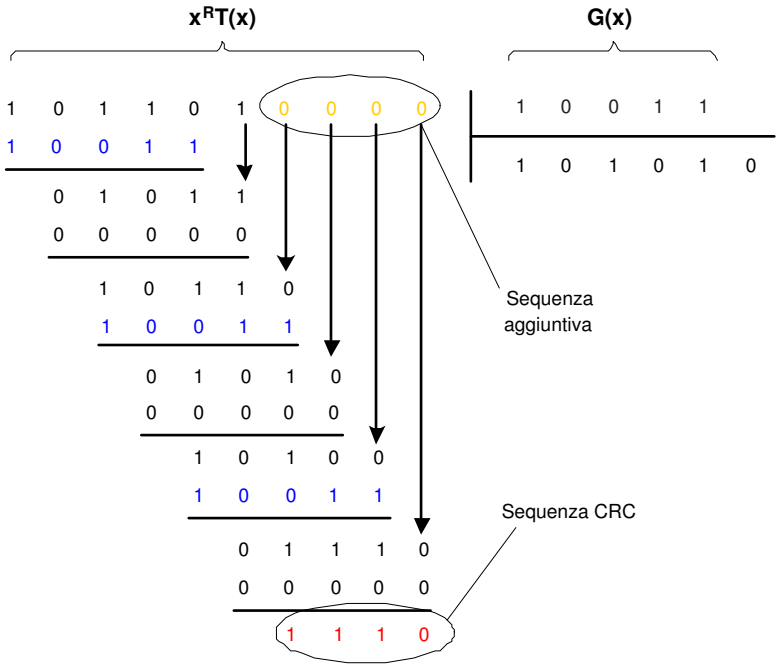


Figura 3.11: Calcolo del CRC: divisione binaria modulo 2 tra $x^R T(x)$ e $G(x)$.

3.1.2.3 Errori rilevati da CRC

Il polinomio che costruisce il ricevitore possiamo scriverlo anche in questo modo:

$$T'(x) = T(x) + E(x)$$

dove $E(x)$ è il polinomio di errore: ogni coefficiente non nullo di $E(x)$ corrisponde ad un bit invertito per errore nella trasmissione di $T(x)$. Se la divisione che esegue il ricevitore restituisce resto nullo ci sono due alternative possibili:

$E(x) = 0$: non ci sono errori;

$E(x)$ è multiplo di $G(x)$: la trasmissione ha generato degli errori ma questi non vengono rilevati dal nostro sistema di codifica.

Ne consegue che la scelta del polinomio generatore è critica, le sue caratteristiche determinano la capacità dell'algoritmo di rilevare certe tipologie di errori.

> Il generatore deve essere scelto in modo tale da non dividere esattamente il polinomio di errore. Si può dimostrare che un

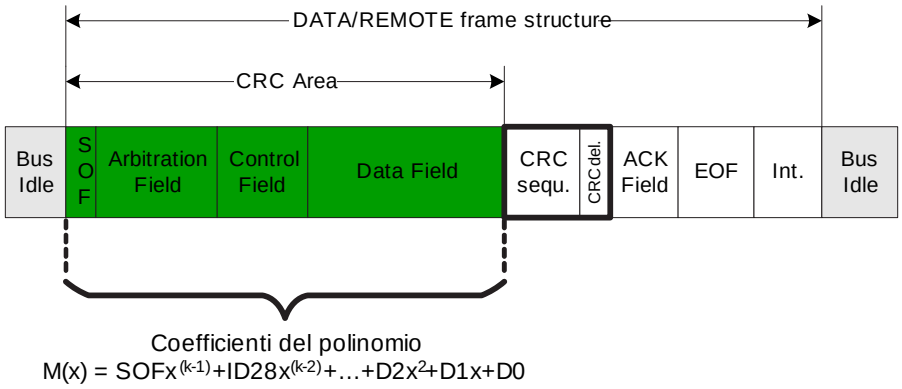


Figura 3.12: "CRC Area" e coefficienti del polinomio di calcolo $M(x)$.

polinomio $G(x)$ che ha due o più termini *rileva tutti gli errori singoli*, se ha grado R *rileva tutti i burst*[13] *composti al più da R errori* ecc.... Alcuni polinomi generatori sono diventati degli standard internazionali come a titolo esemplificativo il "CRC-16" sotto riportato:

$$G(x) = x^{16} + x^{15} + x^2 + 1$$

3.1.2.4 La sequenza CRC del protocollo CAN

Il polinomio $M(x)$ descritto in precedenza viene costruito dal trasmettitore utilizzando come coefficienti i bit dell'area evidenziata in figura 3.12 ed indicata con il termine "CRC Area"; esemplificando possiamo dire che:

$$m(K-1) = \textbf{SOF}\ ,\ m(K-2) = \textbf{ID28},\ m(K-2) = \textbf{ID27}\ \ldots$$

Come generatore il CAN utilizza il seguente polinomio:

$$G(x) = x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$$

che consente di rilevare:

- la presenza al massimo di 5 errori singoli casualmente distribuiti[14] nella "CRC Area";
- errori di tipo burst con lunghezza massima pari a 15 bit.

[13] L'errore di tipo *burst* è una sequenza di bit che inizia e termina con un errore anche se i bit tra un errore e l'altro sono esatti. La dimensione che viene assegnata e caratterizza il *burst* corrisponde alla lunghezza della sequenza.

[14] Tecnicamente si parla di una *distanza di Hamming* pari a 6.

Nel listato 3.1 è descritto in un linguaggio pseudopascal la procedura di calcolo della sequenza CRC. Allo scopo di implementare questa funzionalità viene impiegato uno shift register a 15 bit `CRC_RG(14:0)` e una variabile `NXTBIT` utilizzata per contenere il valore del bit successivo nello stream da codificare.

```
1 CRC_RG = 0                                \\init shift reg.
2 REPEAT
3     CRCNXT = NXTBIT EXOR CRC_RG(14)
4     CRC_RG(14:1) = CRC_RG(13:0)           \\shift left by...
5     CRC_RG(0) = 0                         \\...one position
6     IF CRCNXT THEN
7             CRC_RG(14:0) = CRC_RG(14:0) EXOR (4599 hex)
8     ENDIF
9 UNTIL (CRC SEQUENCE starts or ERROR condition)
```

Listato 3.1: Algoritmo di calcolo della sequenza CRC implementato nel protocollo CAN.

Il nodo ricevente dopo aver eseguito il destuffing del frame calcola anch'esso la sequenza CRC, la confronta con quella trasmessa ed in presenza di una qualche difformità rileva una condizione di errore di tipo *CRC Error*; successivamente al ricevimento dell'ACK Delimiter invia un ERROR Flag.

3.1.3 Gestione degli errori

Dato che il CAN è stato sviluppato inizialmente per l'uso in applicazioni automotive una delle sue principali caratteristiche è l'efficienza nella rilevazione e gestione degli errori, elemento critico in questo campo applicativo.

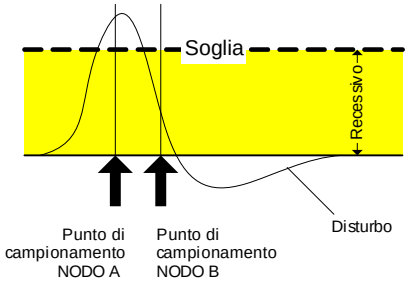
I nodi CAN hanno la capacità di rilevare un errore e passare in una modalità adeguata allo stato operativo in cui si trova l'interfaccia che al limite può addirittura autodisattivarsi (stato di *bus-off*); questa caratteristica prende il nome di *Fault Confinement*. Nessuna *CAN Station* difettosa può rischiare di monopolizzare la rete grazie al suo progressivo confinamento che subisce alla rilevazione degli errori[15]. Il nodo CAN è anche in grado di rilevare piccoli disturbi provocati ad esempio dalle tolleranze di funzionamento dell'oscillatore e modificare il proprio comportamento di conseguenza (risincronizzazione).

[15]Il nodo in condizioni normali si trova nello stato *Error Active*, alla rilevazione e generazione di un sufficiente numero di errori passa in modalità *Error Passive* ed al limite, in caso di gravi malfuzionamenti può addirittura essere disattivato (stato di *Bus-off*).

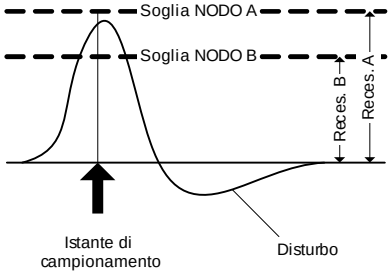
3.1.3.1 Cause di errore

Le cause che possono provocare un errore in fase di trasmissione e ricezione di un dato sul bus sono diverse; le principali sono di seguito riepilogate e descritte.

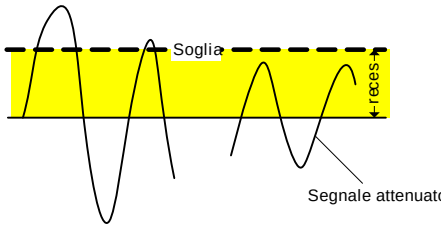
1. Ogni nodo può avere un differente istante di campionamento e quindi per effetto del disturbo può verificarsi che due nodi A e B leggano due livelli logici differenti.



2. Due diversi nodi possono avere soglie diverse di riconoscimento dei livelli logici.



3. Il segnale lungo la linea si attenua e quindi si può avere una degradazione eccessiva del dato.



3.1.3.2 Condizioni di errore

Come accennato in precedenza il CAN prevede 3 differenti modalità operative dette *Error-state* in cui il nodo si può trovare in funzione del numero di condizioni di errore che rileva e che genera.

Quando un nodo rileva una condizione di errore (*Local Error*) genera un ERROR Frame con le caratteristiche precedentemente descritte. Per sua costruzione il frame è tale da far rilevare un errore anche agli altri nodi della rete; l'ERROR Frame ha l'effetto di *globalizzare l'errore*[16]. Tutti i nodi della

[16]La globalizzazione è necessaria per garantire la consistenza dei dati trasmessi.

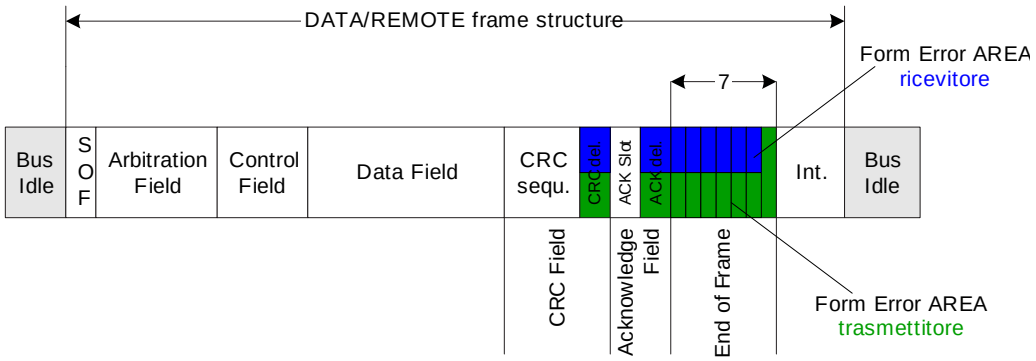


Figura 3.13: Sezioni del frame su cui il CAN verifica la presenza di Form-Error.

rete bollano il messaggio come "non valido", lo scartano ed inizia una nuova nuova fase di arbitraggio (i nodi provano a ritrasmettere il loro messaggio).

Nelle specifiche del protocollo sono descritte 5 condizioni di errore di seguito descritte.

CRC Error : i 15 bit del Cyclic Redundancy Check vengono calcolati dal nodo trasmettitore e inviati nel *CRC Field*. Tutti i nodi della rete ricevono il messaggio, calcolano il CRC e verificano che coincida con quello contenuto nel *CRC Field*, se ciò non si verifica viene prodotto un *CRC Error*. Vedi figura 3.12.

Acknowledge Error : nell'*Acknowledge Field* del messaggio, il trasmettitore verifica se il bit di *Acknowledge Slot* (inviato come recessivo) è dominante ed in caso affermativo significa che almeno un nodo della rete a ricevuto correttamente il messaggio altrimenti viene prodotto un *Ack Error*.

Form Error : se un nodo rileva un bit dominante in uno dei seguenti 4 segmenti: *End of Frame, Acknowledge Delimiter, CRC Delimiter*, significa che è stata violata la struttura del messaggio e conseguentemente si rileva un errore di forma[17] (*Form Error*). Vedi figura 3.13.

Bit Error : si verifica un *Bit Error* se un nodo trasmettitore invia un bit dominante e rileva un recessivo o viceversa durante la fase di monitoraggio della linea, confrontando quanto letto con il bit appena trasmesso. Nel caso in cui il nodo dopo aver inviato un recessivo legge un dominante

[17]Unica eccezione a questa regola si applica quando l'ultimo bit dell'*End of Frame* è dominante, questa situazione non viene rilevata come un errore ed al posto di un ERROR Frame il nodo genera un OVERLOAD Frame.

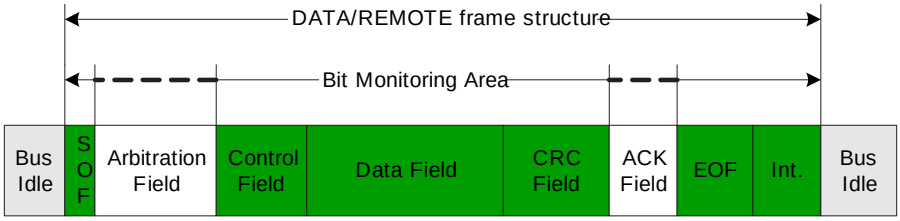


Figura 3.14: Sezioni del frame su cui il CAN verifica la presenza di Bit-Error.

durante l'*Arbitration Field* o l'*Acknowledge Slot*, non viene rilevato alcun *Bit Error* perché quanto descritto è caratteristico di una normale fase di arbitraggio e acknowledgement. I campi del frame sottoposti a questo controllo sono evidenziati in figura 3.14.

Stuff Error : come verrà successivamente descritto il protocollo impiega un metodo di trasmissione di tipo NRZ che richiede lo "stuffing" dei bit e per effetto del *bit-stuffing* non possono esistere più di 5 bit con la solita "polarità" nella sequenza che arriva al ricevitore. Se tra lo *Start of Frame* e il *CRC Delimiter* incluso, vengono trovati 6 bit consecutivi dello stesso valore la regola di *bit-stuffing* viene violata e viene riconosciuto uno *Stuff Error* (vedi figura 3.17).

3.1.3.3 Fault confinement

La rilevazione di un errore "locale" e la sua conseguente globalizzazione non ha solo l'effetto di scartare il messaggio trasmesso e dare il via ad una nuova ritrasmissione ma anche quello di alterare il valore contenuto nei registri denominati *REC (Receive Error Counter)* e *TEC (Transmit Error Counter)* dei nodi appartenenti alla rete. E' per mezzo di questi registri contatori che il protocollo CAN gestisce lo stato di *Fault confinement* dei nodi.

Questi contatori sono incrementati quando viene rilevato un errore e sono decrementati quando il nodo esegue una trasmissione o una ricezione in modo corretto. Le *Fault confinement rule* specificate nel protocollo elencano dettagliatamente le condizioni ed i valori con cui TEC e REC devono essere incrementati e decrementati.

Attraverso TEC e REC il protocollo distingue i nodi che funzionano correttamente (*Error Active node*) da quelli che presentano un malfunzionamenti temporanei (*Error Passive node*) e da quelli che invece sono non funzionanti (*Bus Off node*). In figura 3.15 vengono rappresentate schematicamente le condizioni di passaggio da uno stato all'altro.

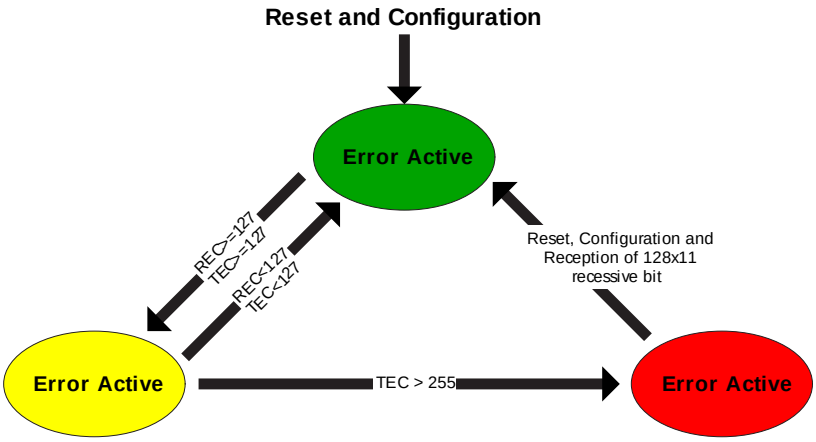


Figura 3.15: Regole di *fault confinement* dei nodi di una rete CAN.

Error Active Un nodo in questo stato può prendere parte attivamente alla comunicazione sul bus, in particolar modo alla rilevazione di una condizione di errore genera un ERROR Frame con un campo *Error Flag* di tipo attivo costituito da 6 bit dominanti (*Error Active Flag*). In questo stato in cui il nodo si porta al reset l'interfaccia ha attive tutte le sue funzionalità senza alcuna restrizione. Un nodo CAN si trova in questa modalità quando i contatori TEC e REC sono entrambi inferiori a 128.

Error Passive Un nodo diventa tale quando TEC o REC superano la soglia di 127 ed in questo stato alla rilevazione di un errore il nodo genera un ERROR Frame di tipo "passivo" caratterizzato dal fatto che l'*Error Flag* è composto da 6 bit recessivi (*Error Passive Flag*). Il frame generato non ha alcun effetto a meno che il nodo non sia abilitato in trasmissione, solo in tal caso anche i 6 bit recessivi violano le regole di bit-stuffing e l'errore viene rilevato anche dalle altre stazioni della rete.

Bus Off Quando entrambi i contatori sono superiori a 255 il nodo si scollega dal bus e và in *Bus Off*[18]. Ovviamente il nodo in queste condizioni non può più inviare o ricevere messaggi, dare l'acknowledge, trasmettere ERROR Frame e quant'altro. Questo stato può essere abbandonato solo per effetto di un reset software o hardware; il reset software viene ottenuto dopo che il nodo riceve 128x11 bit recessivi.

[18]I soli errori di ricezione non possono portare un nodo in stato di *Bus Off*.

3.1.4 Codifica dei messaggi

In questo paragrafo si vuole descrivere quale tecnica viene impiegata dal CAN nella codifica dei bit che vengono trasmessi lungo la linea e la motivazione di certi meccanismi come quello del *bit-stuffing*.

3.1.4.1 La codifica NRZ

Il protocollo CAN utilizza per la codifica dei bit il meccanismo di *Non-Return-to-Zero (NRZ)*: la linea viene pilotata con un valore alto di tensione per codificare il livello logico 1 e rimane in questo stato per un tempo pari al *bit-time* (durata di un bit) moltiplicato per il numero di bit a livello logico alto che devono essere trasmessi; la stessa cosa accade quando si deve trasmette uno o più bit a livello logico 0.

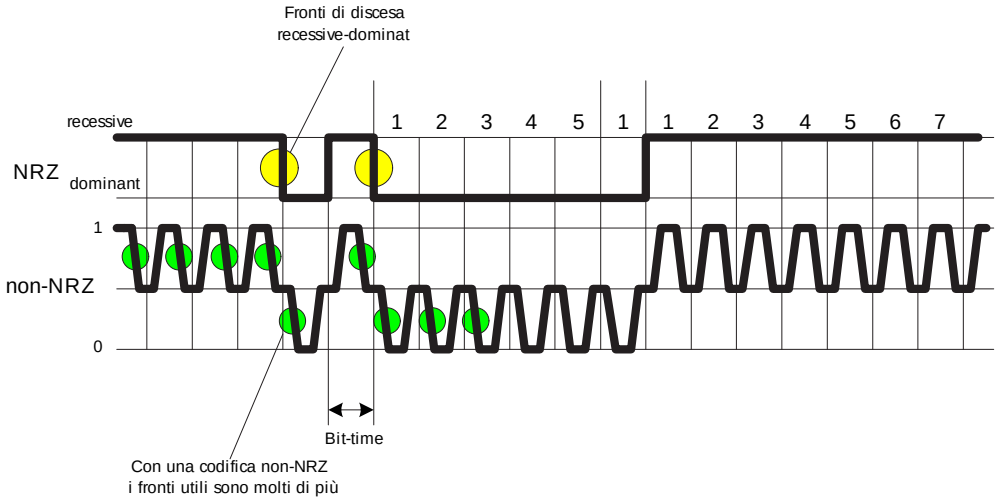


Figura 3.16: Codifica dei bit mediante l'uso della tecnica NRZ.

Come conseguenza sul bus il segnale presenta dei fronti di salita e discesa non in corrispondenza di ogni singolo bit trasmesso ma solo ogni volta che cambia lo stato del bus.

- La trasmissione dei frame avviene in modo sincrono ed i nodi della rete sfruttano il fronte di discesa del segnale che si genera al momento della trasmissione del bit di Start of Frame; su questo fronte avviene una prima sincronizzazione detta *Hard-synchronization*.
- Durante la trasmissione di un messaggio i nodi rischiano di perdere la sincronizzazione per effetto delle tolleranze del circuito di oscillazione

che genera il clock; i fronti di discesa del segnale sul bus (fronti generati dal passaggio da un bit recessivo ad uno dominante) vengono sfruttati dalle interfaccie per resincronizzarsi (*Soft-synchronization*). Il rischio di adottare una codifica NRZ è che il numero di questi fronti di discesa da utilizzare per la risincronizzazione possono essere molto pochi quando il messaggio è composto da lunghe sequenze di bit dello stesso valore e per questo motivo il frame viene sottoposto dal nodo trasmittente all'operazione di *bit-stuffing*.

- Il nodo si resincronizza anticipando o posticipando a seconda dei casi l'istante di campionamento del segnale con le modalità indicate in sezione 3.1.6.2.

3.1.4.2 Bit-stuffing

Le specifiche del protocollo impongono al nodo trasmittente di sottoporre ad operazione di *bit-stuffing* i campi di *Start of Frame, Arbitration, Control* e la *CRC sequence* dei DATA e REMOTE Frame[19].

Quando il trasmettitore rileva 5 bit consecutivi di identico valore nella sequenza da inviare ed appartente alla "bit-stuffing area" (compreso l'eventuale bit di stuff inserito in precedenza), automaticamente inserisce un bit di valore opposto nella sequenza, in questo modo introduce sul segnale trasmesso un fronte recessivo-dominante in più su cui i nodi si risincronizzeranno.

I ricevitori eseguono l'operazione inversa: cercano nella sequenza ricevuta 5 bit dello stesso valore e conseguentemente rimuovono dal frame il bit successivo (lo stuff-bit inserito dal trasmettitore); questa operazione è detta *destuffing* del frame. Nel caso in cui il ricevitore trovi più di 5 bit consecutivi con la stessa polarità tra l'arrivo dello *Start of Frame* e la fine della *CRC sequence*, genera uno *STUFF Error* e conseguentemente trasmette un ERROR Frame.

3.1.5 CAN physical layer

Come detto la linea del bus CAN ha due stati logici possibili: uno "recessivo", l'altro "dominante". Lo stato logico attuale del bus corrisponde all'operazione di AND fatta su i livelli logici che impongono sul bus i nodi che a esso sono collegati. La linea impiegata per realizzare un bus CAN deve essere in grado di gestire questi due stati.

[19]Le altre sezioni del frame, gli ERROR Frame e gli OVERLOAD Frame non sono sottoposti a bit-stuffing.

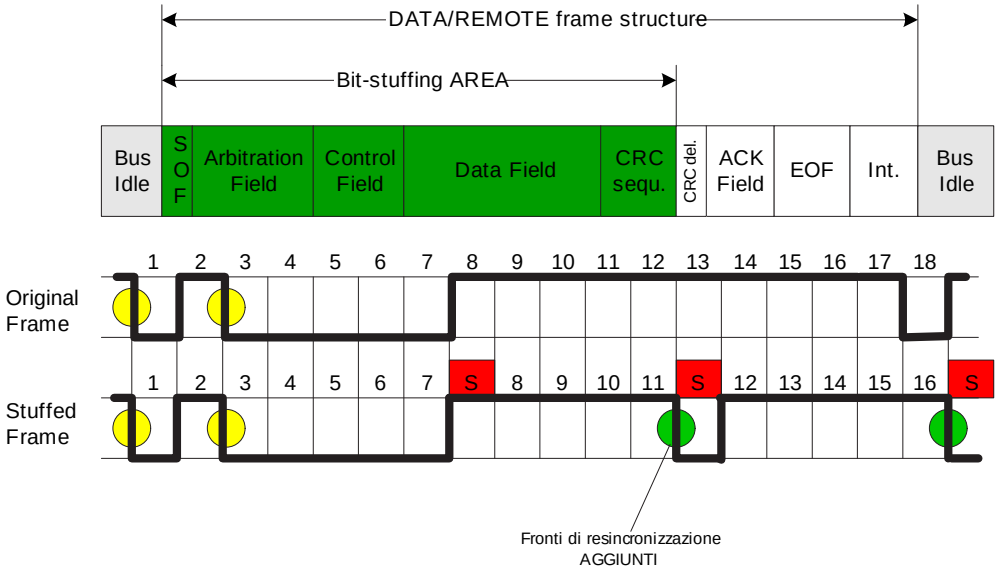


Figura 3.17: Metodo del bit-stuffing.

Nella configurazione più comune il bus è realizzato con una coppia intrecciata di fili terminati ad entrambi i capi con due resistori da 120Ω (ISO 11898). Le due linee indicate con CAN_H e CAN_L vengono pilotate dai nodi in modo differenziale e per effetto di ciò il CAN è praticamente insensibile alle interferenze elettromagnetiche[20] (le linee, in ogni tratto e su tutta la loro lunghezza, sono affette dallo stesso disturbo che quindi scompare sul segnale differenziale). Il circuito (parte del nodo) che si occupa di pilotare la linea e che potremmo impropriamente descrivere come un convertitore di livello è detto *CAN Transceiver*.

La configurazione appena descritta costituisce il *Physical Medium Attachment sublayer (PMA)* del protocollo e sostanzialmente descrive la struttura del bus e le caratteristiche dei driver di linea (i transceiver). Oltre alla configurazione appena descritta esistono altri tipi di implementazione che rispettano altrettanti standard alcuni dei quali (i più diffusi) sono di seguito riportati.

ISO 11898-2 : è uno standard nato per essere impiegato in reti CAN ad alta velocità come ad esempio per la realizzazione di network di classe

[20] Per limitare i disturbi provocati dalle *emissioni* di radiazioni elettromagnetiche le linee vengono spesso schermate.

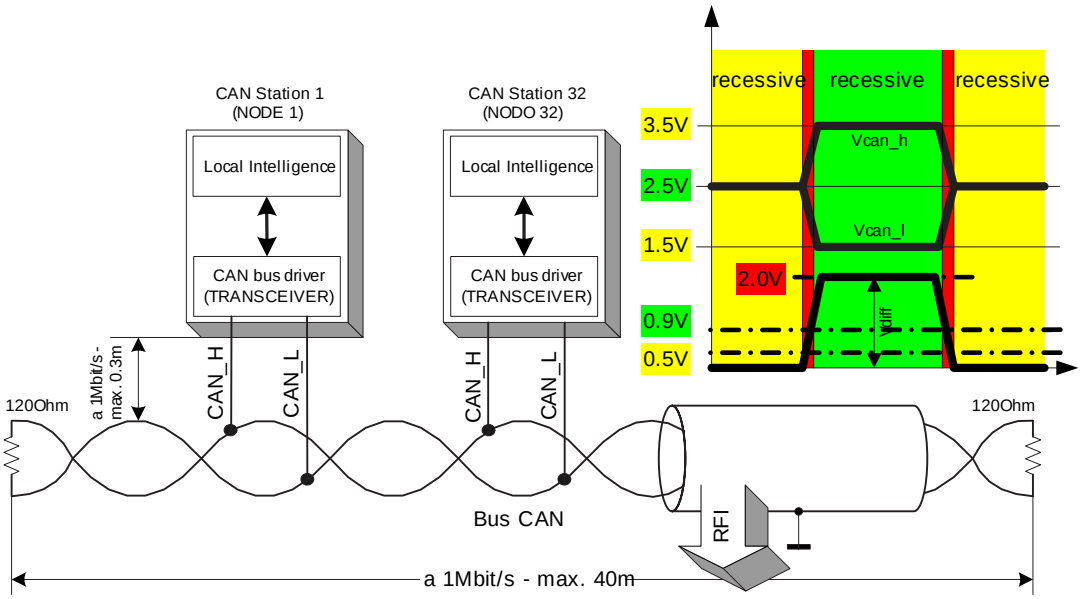


Figura 3.18: ISO11898-2 physical layer.

C[21]. Utilizza una linea a due fili intrecciati e le sue caratteristiche[22] sono sintetizzate riportate in figura 3.18.

ISO 11898-3 : lavora fino a bit-rate non superiori a 125Kbit/s e nasce per per applicazione di tipo *Fault- tolerant*. I Transceiver di tipo ISO 11898-3 sono in grado di lavorare anche in modalità single-wire in caso di errore e pilotano la linea nel rispetto della caratteristica rappresentata in figura 3.19.

ISO 11992 : è stato originariamente sviluppato per la distribuzione dei segnali negli autoarticolati dove sono frequenti disturbi HF a modo comune elevati (100V) e pesanti offset del riferimento di massa (6V). Lo standard presenta capacità di tipo fault-tolerance come il precedente ed i suoi transceiver pilotano la linea nel modo descritto in figura 3.19.

[21] In campo automotive i bus vengono classificati in tre categorie a seconda delle differenti velocità di trasmissione. Classe "A" per reti low-speed con bit-rate fino a 10Kbit/s; Classe "B" per reti low- speed con velocità leggermente superiori (fino a 125Kbit/s) ed infine i bus di classe "C" che si riferiscono a reti high-speed che raggiungono al massimo bit-rate di 1Mbit/s.

[22] La limitata capacità di pilotare più di una trentina di nodi deriva dalle caratteristiche elettriche del transceiver previsto dalle specifiche. Tratte più lunghe sono gestibili solo facendo ricorso a dei ripetitori di segnale.

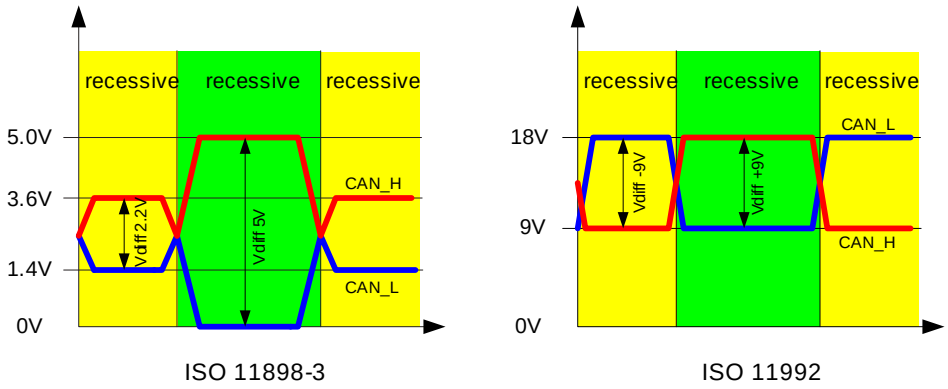


Figura 3.19: ISO 11898-3 e ISO 11992 physical layer.

3.1.6 Bit timing

Il protocollo CAN da la possibilità all'utente di impostare il bitrate della rete, l'istante di campionamento del bit (il *Nominal Sample point*) ed il numero di volte che il bit deve essere campionato; con queste caratteristiche la rete può essere adattata ad ogni specifica esigenza, infatti c'è tenere sempre presente che i parametri di bit timing sono legati a diversi fattori caratterizzanti il bus a livello fisico come il ritardo di propagazione del segnale sulla linea e le tolleranze degli oscillatori locali[23].

3.1.6.1 Il bit time di una rete CAN

Il *bit time* corrisponde al tempo per cui un nodo trasmittente deve mantenere il bus ad livello logico costante affinché questo questo vengo riconosciuto da un qualunque nodo ricevitore. Da questa definizione possiamo ricavare immediatamente il baudrate della rete indicando con t_{bit} il valore di *bit time*:

$$Nominal\ Bitrate = \frac{1}{t_{bit}} \quad [bit/s]$$

Il protocollo CAN suddivide ogni *bit time* in quattro segmenti ognuno dei quali è composto da un multiplo intero del *Time Quantum* (t_q). Il *Time Quantum* è un riferimento temporale fisso ricavato dal periodo di oscillazione del segnale di clock locale del nodo e che viene determinato moltiplicando questo periodo di oscillazione (t_{osc}) per un valore intero compreso tra 1 e

[23] Per tutta questa serie di motivi è fondamentale per il progettista manipolare i parametri di bit-timing ed è quindi di fondamentale importanza che anche CAN*analyser* sia in grado di fornire all'operatore questa possibilità.

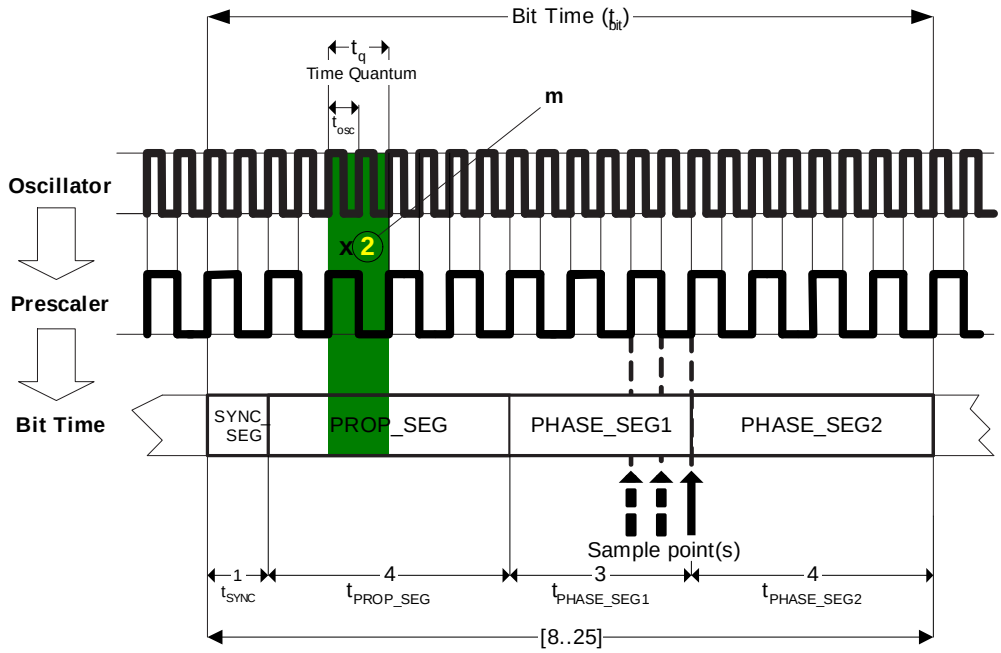


Figura 3.20: Schematizzazione del processo di determinazione del *bit time*.

32[24]. Quindi possiamo scrivere che:

$$t_q = mt_{osc} = \frac{m}{f_{osc}}$$

dove m è il valore del prescaler[25] e f_{osc} è la frequenza dell'oscillatore locale.

> Il baudrate di una rete CAN è determinato una volta che viene stabilito il valore del *Time Quantum* ed il numero di Time Quanta che compongono i quattro segmenti nei quali viene suddiviso il *bit time*.

[24]Ogni nodo gestisce logicamente il bus per mezzo di un *CAN controller*, questo integrato dispone normalmente di un circuito denominato *programmable prescaler* che consente di ottenere un segnale con periodo pari o superiore a quello di ingresso. Il valore per cui il t_{osc} viene "moltiplicato" è programmabile a scelta dell'utente scrivendo un valore opportuno in un registro del *prescaler* stesso.

[25]Le specifiche impongono che il *bit time* può essere composto da un numero variabile di Time Quanta compreso tra 8 e 25.

3.1.6.2 I quattro segmenti del bit time

I quattro segmenti nei quali viene suddiviso il *bit time* sono indicati in figura 3.20. Possiamo quindi andare a scrivere che:

$$t_{bit} = t_{SYNC_SEG} + t_{PROP_SEG} + t_{PHASE_SEG1} + t_{PHASE_SEG2}$$

Synchronization Segment Il SYNC_SEG è il primo segmento che costituisce il *bit time*, la sua durata è fissa e pari ad un Time Quanta ed all'interno di questo periodo il nodo CAN attende l'eventuale fronte di variazione del segnale.

Propagation Segment Il PROP_SEG esiste per compensare il ritardo nella propagazione del segnale tra un nodo e l'altro della rete ed è definito nella seguente maniera:

$$t_{propagation} = 2(t_{cable} + t_{controller} + t_{optocoupler} + t_{transceiver})$$

dove t_{cable} è il ritardo di propagazione prodotto dalla linea, $t_{controller}$ è il tempo di risposta del CAN controller (*Input comparator delay*), con $t_{optocoupler}$ si indica il ritardo introdotto dai fotoaccoppiatori di isolamento (se presenti) ed infine $t_{transceiver}$ è il tempo di risposta del transceiver (*Output driver delay*). Quindi t_{PROP_SEG} tiene conto non solo del ritardo della linea ma considera anche i ritardi introdotti da tutti gli altri elementi costituenti il nodo. PROP_SEG può essere programmato con un valore compreso tra 1 e 8 t_q.

Phase Buffer Segment 1/2 Entrambi i segmenti sono impiegati per compensare l'errore introdotto dal bus sull'istante di campionamento (3.1.4.1). La durata di PHASE_SEG1 e PHASE_SEG2, entro certi limiti fissati dal valore del *Resynchronization Jump Width (SJW)*[26] , può rispettivamente essere allungata o accorciata per recuperare la sincronizzazione perduta. PHASE_SEG1 e PHASE_SEG2 possono assumere valori compresi tra 1 e 8 Time Quanta.

3.1.6.3 Programmazione dei bit-time segment

Le specifiche del protocollo limitano i valori (in termini di t_q) che possono essere assegnati ai singoli segmenti nel rispetto delle seguenti regole:

1. PROP_SEG + PHASE_SEG1 $\geq$ PHASE_SEG2
2. PROP_SEG2 $>$ SJW

[26] SJW può assumere un valore intero compreso tra 1 e 4 Time Quanta.

3. PHASE_SEG2 = max(PHASE_SEG1, IPT)
4. IPT ≤ 2
5. 8 ≤ (SYNC_SEG+PROP_SEG+PHASE_SEG1+PHASE_SEG2) ≤ 25

Il parametro IPT *Information Processing Time* è definito come il tempo richiesto dalla logica del nodo per determinare il livello logico da assegnare al segnale campionato. L'IPT[27] inizia all'istante di campionamento ed è misurato in termini di Time Quanta.

3.1.6.4 Istante di campionamento

Il *Nominal Sample point* è il punto all'interno del *bit time* nel quale il livello logico del bus viene letto ed interpretato. In una rete CAN l'istante di campionamento è fissato alla fine del *Phase Buffer Segment 1*; unica eccezione a questa regola è il caso in cui il nodo sia configurato per campionare il bit non una ma tre volte. In quest'ultimo caso il bit è sempre campionato alla fine di PHASE_SEG1 ma lo è anche altre due volte $\frac{1}{2}\ t_q$ ed t_q secondi prima del punto di campionamento "tradizionale": il risultato del campionamento è il livello logico che si è presentato più volte (almeno 2).

3.1.6.5 Hard Synchronization

Il protocollo CAN è un protocollo asincrono nel senso che non esiste un segnale di clock comune ai nodi della rete e che questi possono usare per sincronizzarsi; ogni nodo è dotato di un oscillatore locale che viene sfruttato per generare la frequenza di campionamento. Affinché la rete funzioni correttamente tutti i nodi devono essere programmati in modo tale da avere lo stesso *bit time* e deve essere previsto un meccanismo che consenta ai nodi di sincronizzarsi come mostrato in figura 3.21 (*Hard Synchronization*).

Il processo di *Hard Synchronization* avviene solo sul primo fronte recessivo-dominante che si verifica durante una fase di bus idle, in altri termini avviene all'inizio della trasmissione del bit di *Start of Frame (SOF).*

Alla ricezione del fronte recessivo-dominante prodotto dal SOF il nodo "resetta" i contatori con cui gestisce il bit timing (*il* bit time *riparte con la*

[27] Questo coefficiente caratterizza il CAN controller. Di fatto il progettista una volta scelto il controllore non è più in grado di mettere mano al valore dell'IPT del proprio nodo. Nel caso dell'ATMEL T89C51CC01 utilizzato nella realizzazione della CAN*interface* di CAN*analyser*, IPT=1.

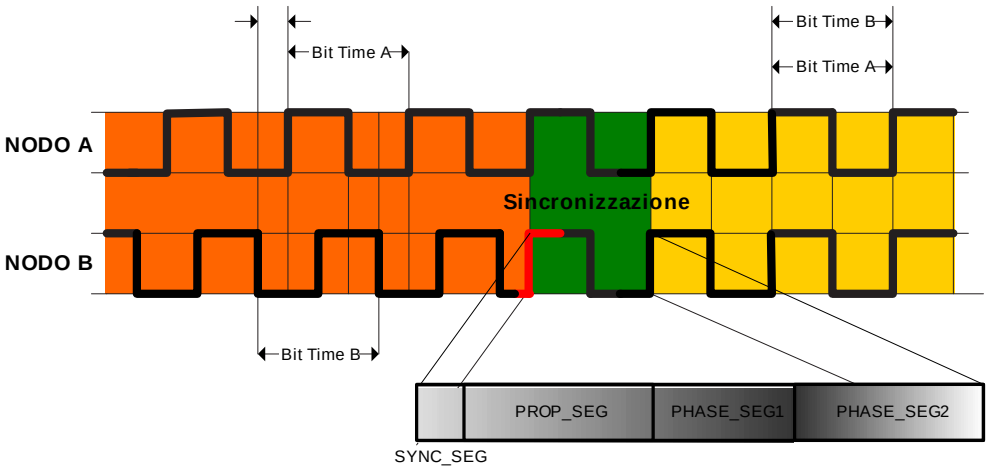


Figura 3.21: Esemplificazione della procedura di sincronizzazione tra due nodi di una generica rete.

fine del SYNC_SEG precedente). La *Hard Synchronization* obbliga il fronte che ha causato la sincronizzazione a "cadere" all'interno del *Synchronization Segment* del nuovo bit. A questo punto la stazione ricevente a recuperato interamente l'errore di fase (3.1.6.6) e si è sincronizzata con il nodo in trasmissione.

Questo tipo di sincronizzazione avviene una sola volta durante la ricezione di un intero messaggio e non è l'unico meccanismo implementato dai nodi. Per effetto del rumore e delle tolleranze degli oscillatori locali, i nodi rischiano di perdere la sincronizzazione raggiunta alla ricezione del SOF durante la trasmissione del corpo del frame, il protocollo prevede un meccanismo detto di *Soft Synchronization* che consente di recuperare (entro certi limiti) questo "disallineamento".

3.1.6.6 Risincronizzazione

Per mezzo della *Soft Synchronization* i nodi riescono a mantenere la sincronizzazione stabilita all'inizio della trasmissione del frame. L'implementazione di questo meccanismo e di quello di *Hard Synchronization* richiede al CAN controller di disporre di un *Digital Phase Lock Loop circuit (DPLL)*.

Il DPLL confronta la posizione del fronte recessivo-dominante sul bus con quella attesa all'interno del SYNC_SEG ed in funzione del valore rilevato aggiusta il *bit time*. L'errore di fase ("e") rilevato è dato dalla differenza tra l'istante di arrivo del fronte e la fine del SYNC_SEG misurato in Time Quanta ed è definito in questo modo:

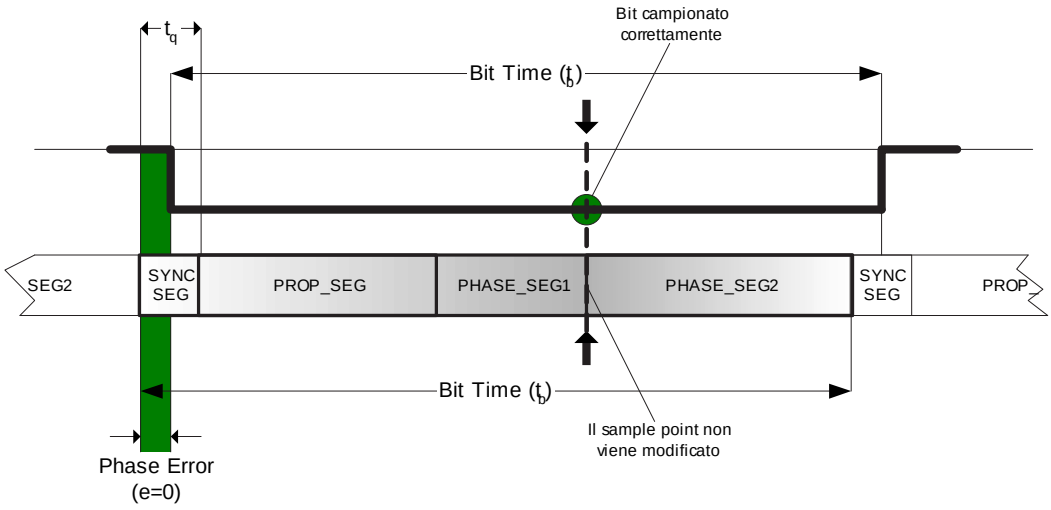


Figura 3.22: Sincronizzazione dei nodi: errore di fase nullo.

e $= 0$: il fronte di risincronizzazione cade all'interno del SYNC_SEG; non c'è bisogno di alcuna operazione di aggiustamento (figura 3.22). L'errore di fase non è nullo ma è certamente inferiore a un Time Quanta.

e > 0 : il fronte cade dopo la fine del SYNC_SEG e prima del punto di campionamento. La sincronizzazione viene recuperata allungato di "$e\ t_q$" secondi il PHASE_SEG1 (figura 3.23).

e < 0 : il fronte cade dopo il punto di campionamento del bit precedente, questa volta si sottraggono "$e\ t_q$" secondi dal PHASE_SEG2 (figura 3.24).

A seguire sono sintetizzate le regole di sincronizzazione descritte nel protocollo dalla versione 1.2 in avanti.

1. Solo i fronti recessivo-dominanti vengono sfruttati per la risincronizzazione.

2. Durante un *bit time* un nodo può risincronizzarsi solo una volta.

3. I fronti sono rilevati andando a campionare il livello logico del bus ogni Time Quantum e confrontando questo valore con quello acquisito durante il precedente *Nominal Sample Point*.

4. Un nodo trasmettitore non si risincronizza alla rilevazione di un errore di fase positivo (e > 0). In questo modo il trasmettitore non rischia di

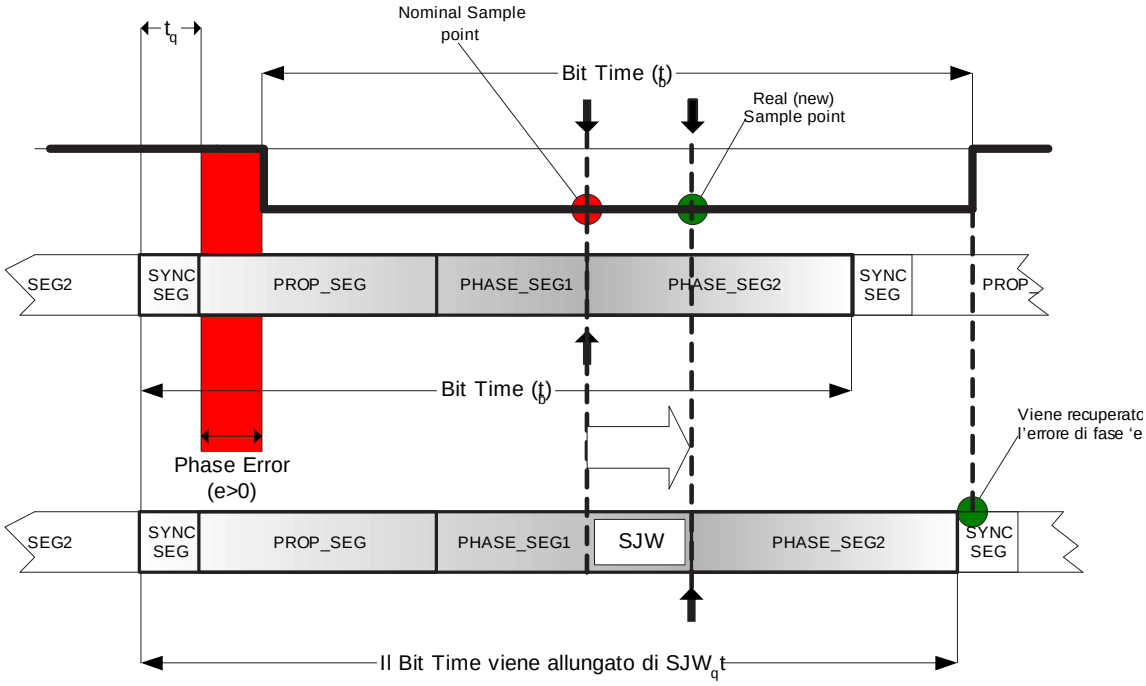


Figura 3.23: Sincronizzazione dei nodi: errore di fase positivo.

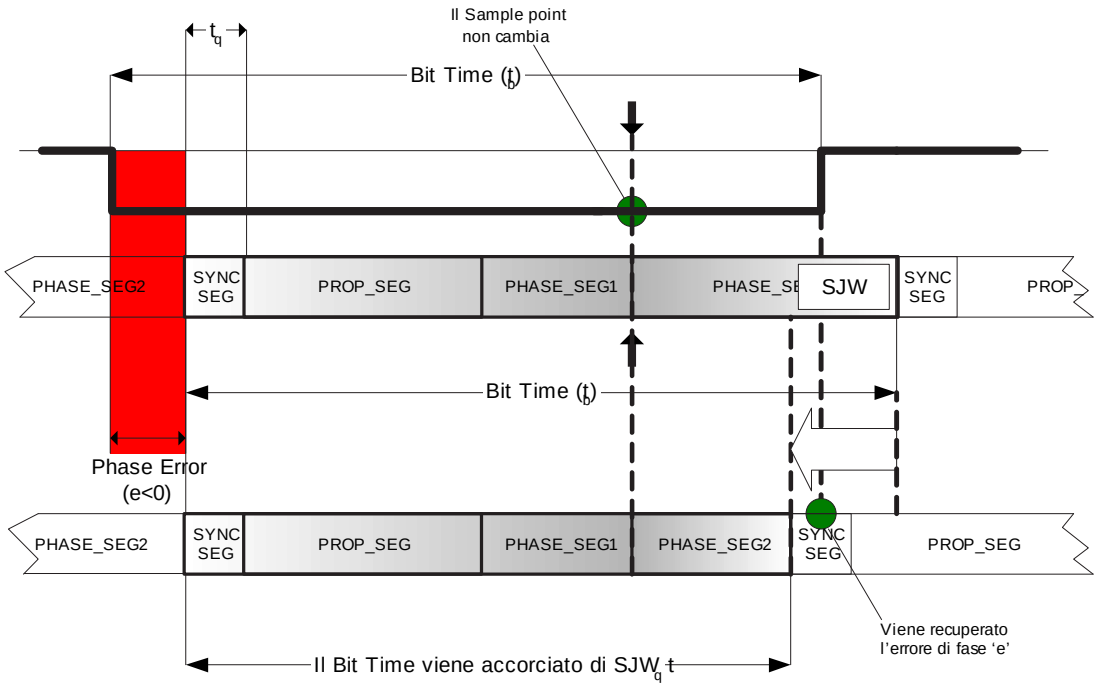


Figura 3.24: Sincronizzazione dei nodi: errore di fase negativo.

risincronizzarsi sui fronti generati da esso stesso generati e che vengono rilevati successivamente per effetto del ritardo di propagazione.

5. PHASE_SEG1 può essere allungato e PHASE_SEG2 accorciato al massimo di SJW Time Quanta.

   Ogni nodo implementa un meccanismo di risincronizzazione per compensare lo shift di fase che rileva mentre riceve un frame.

3.1.7 Impostazione dei parametri

Baudrate, lunghezza del bus e le tolleranze dell'oscillatore sono tre parametri fortemente legati che il progettista può gestire attraverso i parametri di bit timing. La determinazione di questi parametri non è un processo arbitrario, il progettista deve conoscere come i diversi elementi sono influenzati dal bit timing e come manipolarli per ottimizzare le prestazioni della rete [5].

3.1.7.1 Il Propagation Delay

Come visto in precedenza il protocollo CAN implementa una tecnica di arbitraggio del bus di tipo *bitwise non-destructive*, l'implementazione di questo meccanismo richiede che *i nodi siano in grado di campionare il segnale sul bus all'interno dello stesso* bit time *nel quale il bit viene trasmesso.* Ad esempio se due nodi ai due estremi della rete iniziano contemporaneamente a trasmettere inizia una fase di arbitraggio per il controllo del bus; l'arbitraggio avviene correttamente solo se i nodi sono in grado di rilevare lo stato logico del bus all'interno dello stesso *bit time*. Per effetto del ritardo di propagazione c'è il rischio che il *bit time* cada al di là del punto di campionamento con il risultato di ottenere un arbitraggio non corretto come mostrato in figura 3.25. Questo rischio a parità di ritardo introdotto è tanto più probabile quanto maggiore è il baudrate della rete essendo il t_{sample_point} indicato in figura sempre più piccolo. *Si può concludere che il baudrare di una rete limita l'estensione della rete stessa (la lunghezza del bus).*

Per le considerazioni svolte appare evidente l'importanza di valutare questa quantità allo scopo di compensarne gli effetti impostando adeguatamente i parametri di bit timing della rete. Come visto il propagatione delay può essere calcolato andando a sommare i vari ritardi che i singoli elementi introducono:

$$t_{propagation} = 2(t_{cable} + t_{controller} + t_{optocoupler} + t_{transceiver})$$

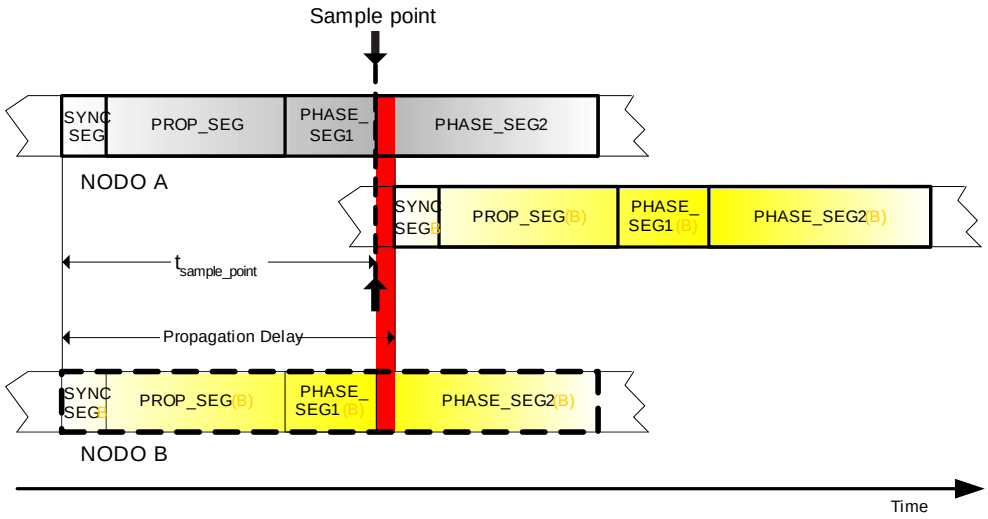


Figura 3.25: Esemplificazione di un arbitraggio errato per effetto del *Propagation Delay*.

tutti questi componenti di ritardo devono essere considerati due volte (figura 3.26) per compensare anche il propagation delay del segnale di risposta durante le fasi di *Hard Synchronization*, di *Acknowledgement* (risposta al bit di Ack Slot) e di *Arbitration* (trasmissione dell'identificatore).

Valori tipici[28] per i ritardi dei singoli componenti oscillano entro i limiti riportati in tabella 3.1. Da questi dati è abbastanza evidente che la lunghezza

Componente	Min Delay	Max Delay (worst case)
Controller	50ns	62ns
Optocoupler	40ns	140ns
Transceiver	120ns	250ns
Cable	5ns/m	5.5ns/m

Tabella 3.1: Tabella riepilogativa dei ritardi introdotti sul segnale dai singoli componenti di un nodo CAN.

del bus normalmente ha un peso enorme sul ritardo di propagazione: sono sufficienti 50 metri di linea per determinare un ritardo di circa 250ns pari al ritardo introdotto dal peggior transceiver. *Lunghezza del bus e propagation delay sono parametri fortemente legati l'uno all'altro.*

[28] I dati riportati nella colonna *Min Delay* sono quelli previsti dalle specifiche CAN High-Speed codificate nello standard ISO11898-2.

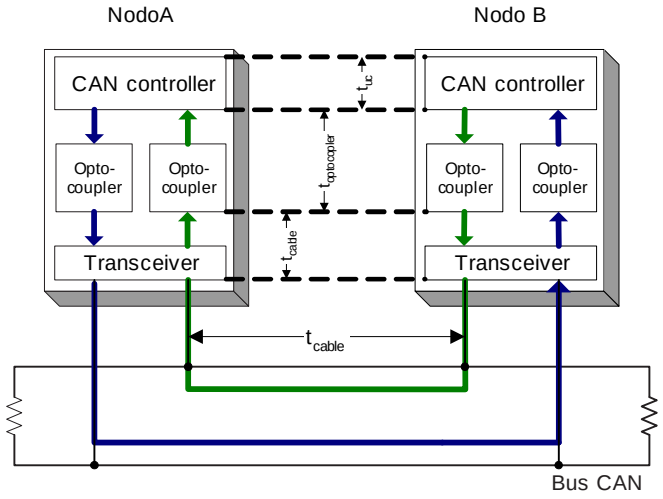


Figura 3.26: Ritardo introdotto sulla propagazione del segnale *Propagation Delay*.

In conclusione il valore di PROP_SEG può essere ricavato dalla semplice relazione:

$$PROP_SEG > \frac{t_{propagation}}{t_q}$$

3.1.7.2 Oscillator Tolerance

Ogni nodo CAN si costruisce il proprio *bit time* dalla frequenza f_{osc} dell'oscillatore locale; la non idealità di quest'ultimo è una delle cause che rendono necessario il meccanismo di risincronizzazione dei nodi infatti la posizione del punto di campionamento reale differisce da quella ideale (nominale).

Lo shift di fase può essere recuperato dal nodo CAN per mezzo della risincronizzazione allungando il PHASE_SEG1 od accorciando PHASE_SEG2 al massimo di "$SJW\ t_q$" secondi. Vediamo in quale maniera si può valutare il valore adeguato di SJW affinché con la *Soft Synchronization* il nodo sia in grado di recuperare interamente l'errore di fase.

L'errore di fase introdotto dagli oscillatori si accumula tra una risincronizzazione e l'altra, cioè tra un fronte recessivo-dominante ed il successivo. La regola di bit-stuffing garantisce che non più di 5 bit dello stesso valore possono essere trasmessi consecutivamente e quindi tra due risincronizzazioni non possono essere inviati più di 10 bit; il valore di SJW deve essere tale da riuscire a recuperare interamente l'errore di fase che si accumula in 10 t_b secondi e quindi:

$$SJW > \frac{t_{SJWmax}}{t_q} = \frac{(2\Delta f)10t_b}{t_q}$$

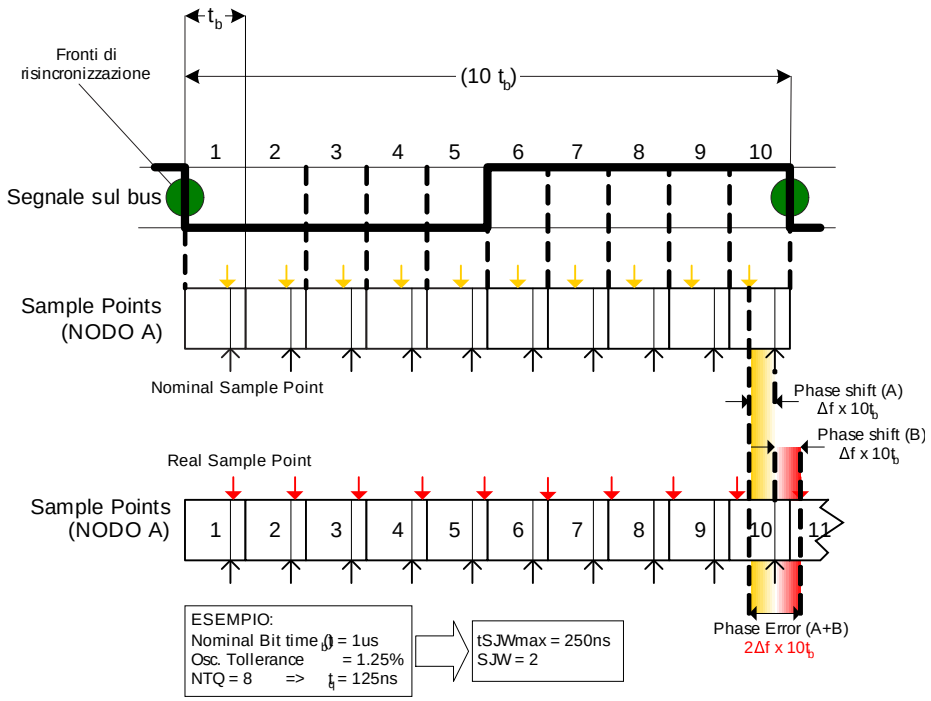


Figura 3.27: Valutazione dello shift di fase tra due nodi nel caso peggiore (worst case).

dove con Δf si è indicata la tolleranza di oscillazione in termini percentuali e t_{SJWmax} è l'errore di fase massimo possibile. Scrivendo il Time Quantum in funzione del *bit time* e del numero di Time Quanta (NTQ) si ottiene un espressione ancora più immediata:

$$SJW > (2\Delta f)10NTQ$$

In figura 3.27 si è esemplificato il caso peggiore a cui si riferisce l'espressione sopra: quello in cui entrambi i nodi hanno una tolleranza massima di Δf e il punto di campionamento reale viene anticipato rispetto a quello nominale nel nodo A e posticipato nel nodo B ($2\Delta f$).

3.1.7.3 Tolleranza massima ammissibile

Quanto appena descritto si riferisce alle valutazioni che deve fare il progettista in fase di programmazione del bit timing dei nodi della rete partendo dai dati a sua disposizione e quindi con un valore Δf noto e che caratterizza le interfacce.

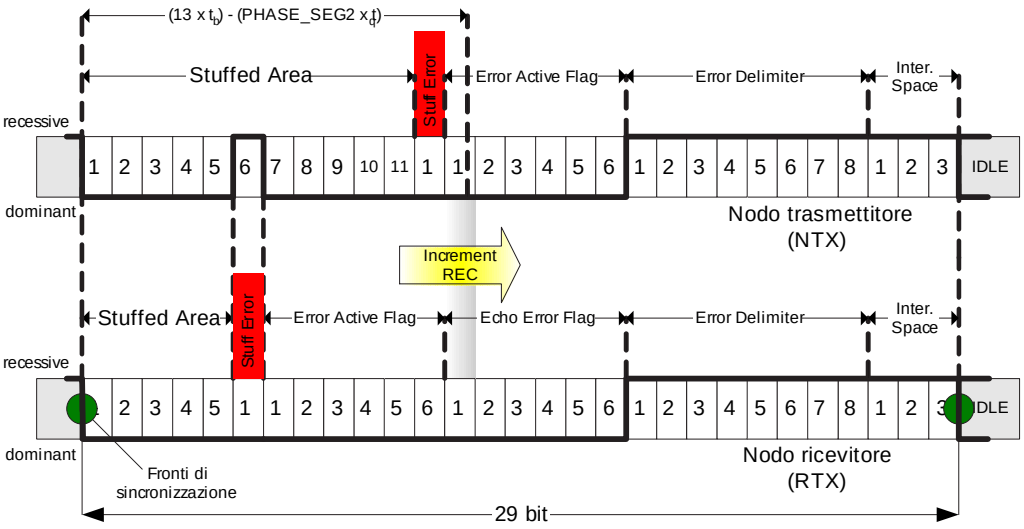


Figura 3.28: Local Error in presenza di almeno due nodi Error Active.

In fase di progettazione del nodo le specifiche impongono un valore massimo di tolleranza del 1.58%, valore che può essere mantenuto a baudrate abbastanza bassi (fino a 125Kbit/s) impiegato come oscillatore locale un semplice risonatore ceramico mentre a frequenze di lavoro più elevate l'impiego di un oscillatore al quarzo è d'obbligo.

Questo limite teorico può essere valutato andando a determinare la massima distanza tra due fronti di risincronizzazione sulla base di quanto detto in precedenza. Di seguito vengono descritte le casistiche di una qualche rilevanza per la valutazione del Δf massimo.

Local Error in presenza di almeno due nodi Error Active La situazioni in cui ci poniamo è quella in cui il nodo in ricezione (NRX) non riconosce il bit di stuff dominante inviato dal nodo trasmettitore (NTX) (figura 3.28); al bit successivo NRX genera un Error Active Flag che allunga la sequenza dei bit dominanti, questa si allunga ulteriormente per effetto dell'Error Active Flag inviato anche dal nodo attivo NTX che genera un echo. Prima che il bus torni in idle e quindi risulti nuovamente disponibile alla trasmissione di un nuovo frame (invio di un nuovo SOF e quindi una nuova sincronizzazione) devono essere trasmessi l'*Error Delimiter* e l'*Itermission Space*. In accordo alle regole di *Fault confinement* il registro REC del nodo NRX viene incrementato durante il primo bit che segue la fine dell'*Error Flag*. *Affinché la rete funzioni correttamente il nodo ricevitore deve essere in grado*

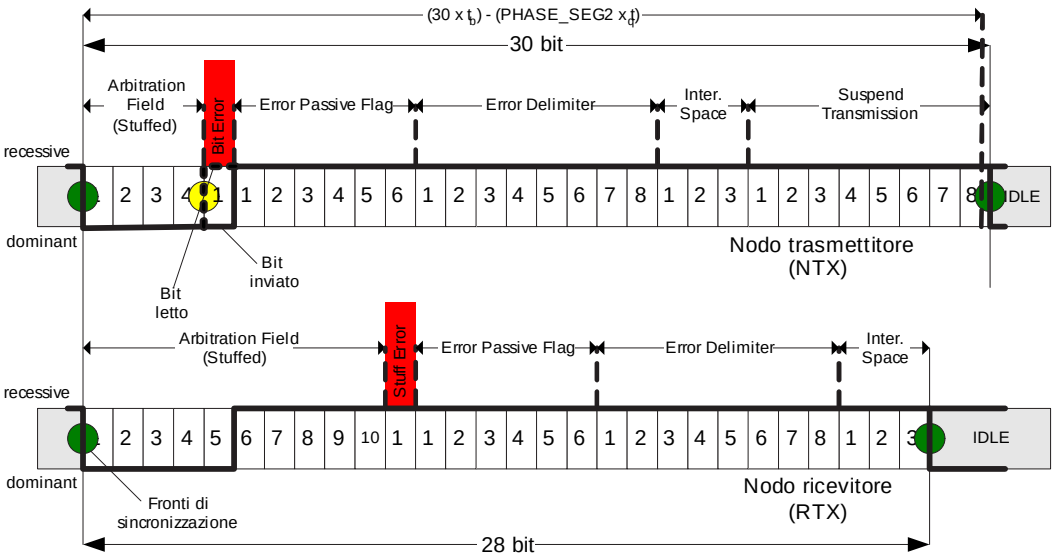


Figura 3.29: Local Error in presenza di nodi tutti passivi.

di discriminare una sequenza di 13 bit dominanti e questo è un primo limite.

Local Error in presenza di nodi tutti passivi In questo caso si suppone che tutti i nodi della rete siano Error Passive e che il nodo trasmittente durante la fase di arbitraggio del bus rilevi un *Bit Error* (invia un dominante ma nello stesso *bit time* rileva un recessivo). A seguito dell'errore NTX invia un Error Passive Flag composto da 6 bit recessivi che dal lato ricevitore producono uno *Stuff Error* cui segue un Error Passive Frame (Flag + Delimiter) e l'*Intermission Space*. In questa situazione tra due fronti di sincronizzazione (l'ultima risincronizzazione e l'invio di un bit di SOF) NRX riceve 28 bit.

Il trasmettitore successivamente all'*Intermission Space* essendo Error Passive si autosospende dal bus per 8 *bit time* (*Suspend Transmission*; per il trasmettitore il tempo di attesa per una nuova sincronizzazione sale a "30 t_b" secondi. (figura 3.29)

Tenendo presenti i casi sopra riportati possiamo concludere che affinché un nodo sia in grado di campionare correttamente il primo bit dopo l'invio di un Error Active Flag (per la corretta rilevazione e gestione dell'errore) la tolleranza massima è limitata dalla seguente espressione:

$$(2\Delta f)(13t_b - PHASE_SEG2\ t_q) \leq SJW\ t_q \tag{3.1}$$

Per la corretta sincronizzazione nella stuffing area del frame DATA o REMOTE, deve essere rispettata la condizione seguente:

$$(2\Delta f)10t_b \leq SJW\ t_q \tag{3.2}$$

Affinché il nodo sia in grado di risincronizzarsi fino a che non arriva uno *Start of Frame* "che rimette le cose apposto", nel caso peggiore, deve recuperare l'errore di fase accumulato su 30 bit quindi:

$$(2\Delta f)(30t_b - PHASE_SEG2\ t_q) \leq SJW\ t_q \tag{3.3}$$

Come già mostrato nel paragrafo 3.1.7.2 la maggiore tolleranza agli shift di fase viene ottenuta dal nodo impostando i parametri di bit timing in questa maniera:

$$SJW = PHASE_SEG1 = PHASE_SEG2 = 4 \tag{3.4}$$

$$SYNC_SEG = PROP_SEG = 1 \tag{3.5}$$

e quindi:

$$t_b = 10t_q \tag{3.6}$$

Il protocollo CAN inizialmente[29] non prevedeva che l'eventuale ricezione di un bit dominante come ultimo bit (il terzo) dell'*Intermission Space* venisse riconosciuto come uno *Start of Frame* e quindi non fosse fonte di errore (3.1.1.1).

Per il protocollo CAN "standard" il limite per l'*Oscillator Tolerance* lo si ricava dalla relazione 3.8 imponendo le precedenti condizioni sul bit timing:

$$(2\Delta f)(30t_b - 0.4t_b) \leq 0.4t_b \tag{3.7}$$

da cui risolvendo per Δf si ottiene che:

$$\Delta f \leq 0.675\%$$

La conseguenza del fatto che la versione "avanzata" del protocollo consente al nodo di "sopportare" un errore sull'ultimo bit dell'*Intermission Space* ha l'effetto di ridurre il numero massimo di bit compresi tra due fronti di sincronizzazione ed aumentare la capacità dei nodi e della rete a far fronte all'errore di fase.

Per la disugualianza 3.8 tra un fronte recessivo-dominante e lo *Start of Frame* ci sono al massimo 30 bit, dato che il nodo "accetta" anche un dominante come terzo bit dell'*Intermission Space* ciò equivale a dire che il nodo

[29] Ci si riferisce alla versione 1.1 del protocollo denominata "standard"; le versioni dalla 1.2 in avanti sono dette protocolli "enhanced". La prima versione in assoluto è stata la 1.0 che non è mai stata implementata.

Figura 3.30: .

è in grado di sopportare un errore di fase di un *bit time* come mostrato in figura 3.30. Dalla stessa figura si deduce che la 3.8 deve essere sostituita dalla nuova condizione limite seguente:

$$(2\Delta f)(33t_b - PHASE_SEG2\ t_q) \leq t_b + SJW\ t_q \tag{3.8}$$

Adesso le tre relazioni da confrontare applicando le condizioni per il calcolo della massima tolleranza sono la 3.2, 3.1 e l'ultima. Il risultato che si ottiene applicando la 3.1 è quello più limitante e corrisponde all'1.58% indicato dalle specifiche come la massima tolleranza ammissibile.

3.1.7.4 Considerazioni sul Baudrate

Partendo dal fatto che una volta caratterizzata fisicamente la rete CAN (definite le caratteristiche dei transceiver, della logica dei nodi, del cavo con cui si realizza la linea e la sua lunghezza) il parametro $t_{propagation}$ è certamente determinabile ed è elemento invariabile e caratterizzante del bus, il bit timing deve essere costruito per venire incontro alle performance che si voglio ottenere dalla rete.

Vediamo in quale modo è possibile impostare questi parametri in due casi estremi: nell'ipotesi di progettare una rete che sia in grado di avere massima tolleranza allo shift di fase (*Maximum Oscillator Tolerance*) ed il caso in cui questo elemento passa in secondo piano rispetto alla massima velocità di trasmissione (*Maximum Bit Rate*).

Maximum Oscillator Tolerance Dato che lo shift di fase viene recuperato durante la risincronizzazione, la condizione di massima tolleranza si ottiene massimizzando la durata del *Resynchronization Jump Width* (t_{SJW}); essendo:

$$t_{SJW} = SJW\ t_q \quad [s]$$

la condizione precedente viene raggiunta in corrispondenza del valore più elevato possibile per SJW ($SJW_{MAX} = 4$) e per t_q (come mostrato in figura 3.31[30] il valore di t_q viene massimizzato dalla condizione $t_q = t_{propagation}$).

[30]Considerando che PHASE_SEG1 e PHASE_SEG2 come minimo devono essere composti da SJW Time Quanta, la condizione di massimo baudrate nel rispetto della condizione di *Maximum Oscillator Tolerance* viene ottenuta per PHASE_SEG1 = PHASE_SEG2 = SJW.

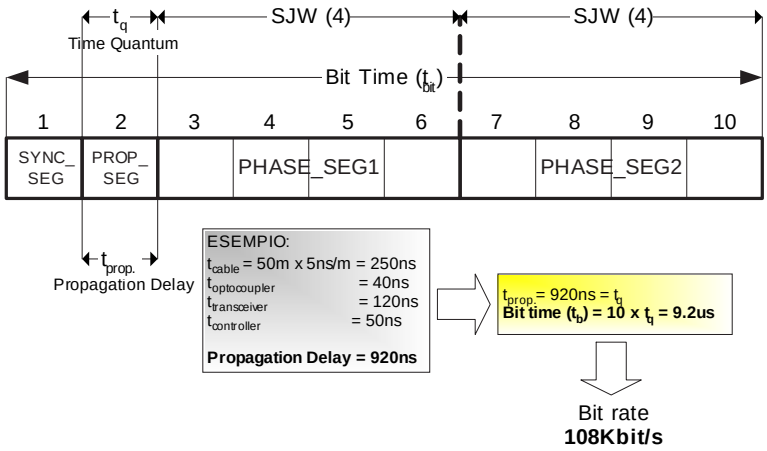


Figura 3.31: Costruzione del bit time per ottenere la condizione di *Maximum Oscillator Tolerance*.

Maximum Bit Rate Massimizzare il baudrate significa ridurre il *bit time* cioè, ridurre il numero di Time Quanta nei quali viene suddiviso un bit e ridurre la durata degli stessi. Tenendo sempre presente che il ritardo di propagazione è una quantità determinata, questa condizione si raggiunge imponendo:

$$PHASE_SEG1_{min} = PHASE_SEG2_{min} = 1 = SJW$$

e

$$t_q = \frac{t_{propagation}}{PROP_SEG_{MAX}} = \frac{t_{propagation}}{8}$$

La struttura del bit timing che ne consegue è riportata in figura 3.32.
Può a questo punto essere di un certo interesse andare a vedere in quale maniera il valore del baudrate risulta essere influenzato dalla lunghezza del bus in entrambi i casi estremi appena proposti. Come mostrato in figura 3.33 e nella tabella 3.2, il baudrate, la lunghezza della linea (Propagation Delay) e l'Oscillator Tolerance sono variabili profondamente legate l'una all'altra; ad esempio se si desidera avere una elevata tolleranza di fase la lunghezza del bus è inevitabilmente limitata, così come tolleranza e baudrate sono quantità inversamente proporzionali.
In conclusione la massima lunghezza che può raggiungere la linea in una rete CAN è essenzialmente funzione dei seguenti effetti fisici:

- il tempo di propagazione del segnale lungo la linea;
- le differenze tra i bit time dei nodi dovute alle tolleranze degli oscillatori locali;

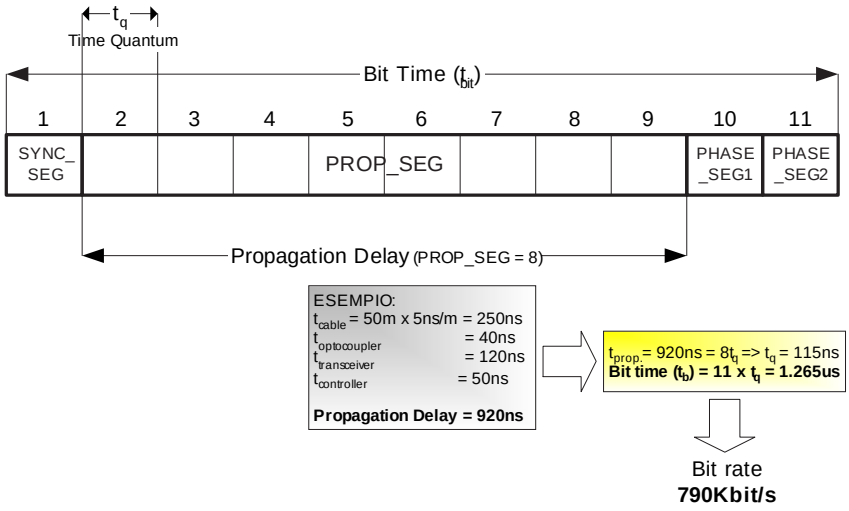


Figura 3.32: Bit timing per ottenere *Maximum Bit Rate*.

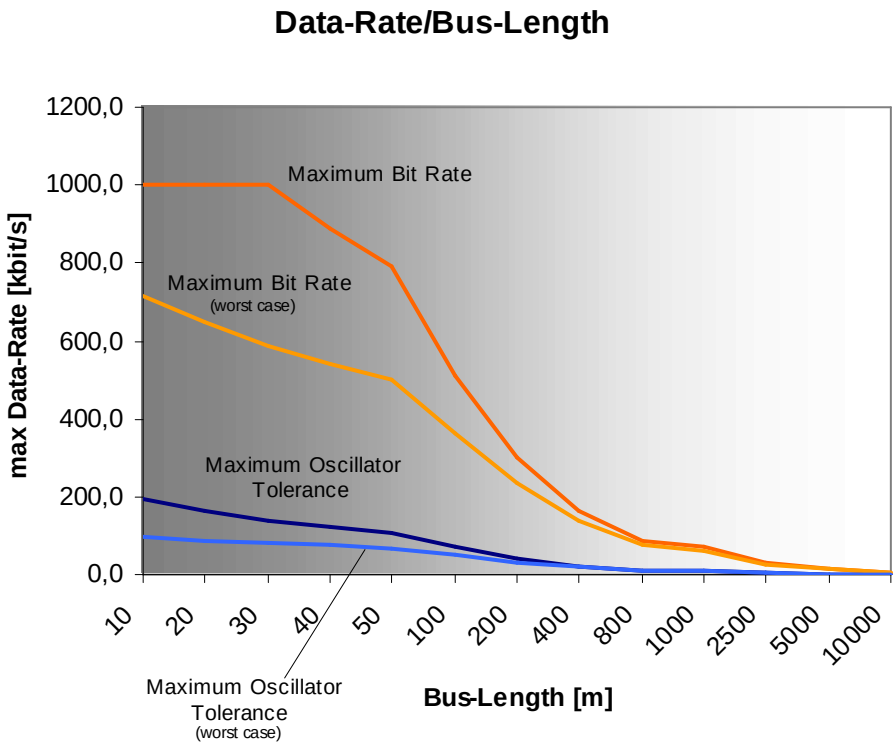


Figura 3.33: Variazione del bitrate in funzione della lunghezza del bus.

Bus Length [m]	Max Rate [Kbit/s]	Max Rate (worst case)
10	1000	717
30	1000	647
50	790	500
100	512	363
200	300	234
400	165	137
800	86	75
1000	70	61
2500	29	26
5000	14	13

Tabella 3.2: Tabella riepilogativa dei valori del baudrate espressi in funzione della lunghezza della linea.

- le cadute di tensione lungo la linea per effetto delle resistenze serie del cavo e per la resistenza di ingresso al bus.

In appendice C sono descritti i registri che il micro ATMEL T89C51CC01 (utilizzato come CAN controller per l'interfaccia CAN*interface*) sfrutta per definire la configurazione di bit timing della rete.

3.2 Informazioni sul sistema di sviluppo utilizzato

3.2.1 Introduzione

Il cuore dell'interfaccia CAN realizzata e chiamata CAN*interface* è il microcontrollore ATMEL T89C51CC01 che è attualmente uno dei tre μC appartenenti alla famiglia ATMEL CANary; ad essa appartengono integrati che combinano alle classiche funzionalità dei dispositivi della famiglia 80C51 con tecnologia Flash, pieno supporto degli standard CAN 2.0A e 2.0B.

La scelta di utilizzare questo modello di μC dotato di CAN controller integrato è stata legittimata dalla disponibilità presso il Laboratorio di Testing dell'Università di un sistema completo di sviluppo per questo tipo integrati e dal fatto che si tratta di un oggetto particolarmente performante.

3.2.2 L'ambiente di sviluppo

Il sistema utilizzato per la progettazione e realizzazione di CAN*analyser* (parte software ed hardware) è composto da:

- due application board "ATMEL CAN-DEMOBOARD1" che corredate del software "ATMEL FLIP" versione 1.8.2 consente di programmare gli integrati della serie CANary e di implementare un bus CAN del tutto funzionante; ognuna delle due board è a tutti gli effetti un nodo CAN in grado di essere connesso ad una rete di tipo High-Speed.

  Caratteristiche tecniche [6]:

  - è dotata di una CAN Extension Board che incrementa le connessioni disponibili del sistema;
  - dispone di un display LCD (2 linee di 16 caratteri);
  - supporta sockets PLCC44, PLCC68 e DIL24;
  - supporto per In-System Programming (ISP);
  - switch per attivazione C51 Flash boot loader dopo il reset;
  - connettore RS232 per la programmazione;
  - connettore DSub conforme CiA DS-102 per la connessione della board ad reti CAN rispondenti a queste specifiche.

- L'ambiente di sviluppo software integrato "μVision2" di Keil Software [7]. Si tratta di un IDE (Integrated Development Environment) dedicato allo sviluppo di applicazioni con microprocessori delle famiglie C51, C251 e C16x/ST10; in un'unica interfaccia utente combina: gestione del progetto, editazione del codice sorgente, compilazione e debugging.

  μVision2 comprende i seguenti moduli:

  - "C51 Compiler" e "A51 Assembler", per la compilazione e/o assemblaggio dei file sorgente. Entrambi sono ottimizzati per fornire pieno supporto alle caratteristiche dei microprocessori.
  - "LIB51 Library Manager", per la creazione e gestione di librerie.
  - "BL51 Linker", per il collegamento dei file oggetto prodotti dai precedenti applicativi in un unico modulo.
  - "OH51 Object-Hex converter": genera un file in formato Intel HEX con cui attraverso il programma "ATMEL Flip" si può programmare il micro.

  - μVision2 Debugger: sfrutta l'oggetto prodotto dal linker per il debug del codice sorgente e simula il μC in tutti i suoi aspetti. Le caratteristiche degli integrati sono configurate automaticamente alla selezione del dispositivo[31] nel momento della creazione del progetto.

  La Keil Software ha sviluppato per questo ambiente anche alcune interessi librerie:

  - "ISD-51" [8], è una libreria che consente di eseguire il debug del codice in modo In-System. Attraverso la porta seriale del dispositivo C51 e la funzione di interrupt fornita dalla libreria "μVision2 Debugger" interagisce con l'applicazione utente che gira direttamente sul target.
  - Come attraverso le librerie "ISD-51" ma con caratteristiche molto più sofisticate, anche il programma "Monitor-51" [9] consente di connettere il target al debugger fornendo così una potente interfaccia di test per l'hardware.
  - Alcune applicazioni basate su micro richiedono che certe operazioni vengano eseguite simulataneamente (mulitasking) mentre altre devono essere eseguite in un certo periodo di tempo (real-time control); la loro implementazione è lasciata al programmatore. Keil Software attraverso "RTX51" ha realizzato un potente real-time operating system (RTOS) attraverso cui le operazioni precedenti vengono organizzate in processi indipendenti (task). L'utilizzo di questo kernel per la gestione delle risorse (CPU, memoria ecc...) rende più semplice lo sviluppo di applicazioni molto complesse; è sufficiente includere l'header e collegare la libreria RTX51 per poter gestire i task in modo semplice e diretto.

- L'applicativo CAN*software* che lavora dal lato PC, dialoga con CAN*interface* e visualizza i dati è stato sviluppato utilizzando l'ambiente di programmazione ad oggetti dedicato a sistemi Microsoft Windows e sviluppato dalla stessa Microsoft: "Visual C++".

[31] Con la selezione del T89C51CC01 viene ad esempio attivata la periferica "CAN Controller". In base alla mia esperienza non sempre i risultati prodotti dal debugger sono del tutto attendibili ed in particolare la gestione di alcuni registri della periferica CAN.

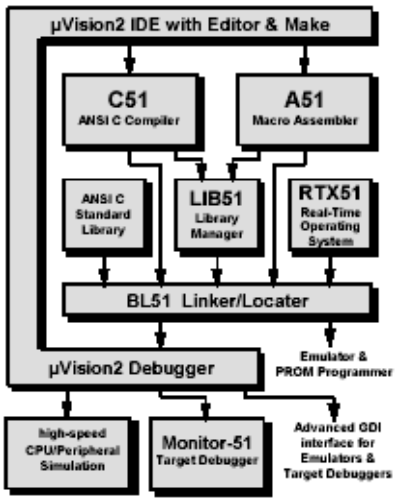


Figura 3.34: Schema funzionale del ciclo di sviluppo del software per mezzo del software μVision2.

3.2.3 Implementazione della rete CAN di test

Come mostrato in figura 3.35 le due demoboard sono state utilizzate per implementare una rete CAN funzionante da utilizzare per il test dell'analizzatore e dove il CAN*analyser* stesso costituisce il terzo modulo di questo sistema. Le due demoboard vengono programmate per simulare del traffico sul bus, traffico che lo strumento deve essere in grado di acquisire ed elaborare.

L'analizzatore quando viene connesso al bus diventa a tutti gli effetti un nodo della rete anche se si tratta di un nodo un pò particolare: quando è in fase di acquisizione dati non interagisce con il bus ed in particolare non fornisce ack di risposta agli altri moduli.

A questo punto si può comprendere per quale motivo si è dovuta realizzare una terza interfaccia con cui implementare la CAN*interface* anziché sfruttare una delle due demoboard disponibili come supporto per l'hardware del nostro strumento.

Quando un nodo trasmette un messaggio verifica che nel campo di Acknowledge del frame il bit di Slot inviato come recessivo ritorni dominante; *almeno uno dei nodi della rete deve riconoscere come valido il messaggio* (figura 3.36). Se l'ack si produce un *ACK-Error* e quindi la trasmissione di un ERROR Frame. Se la rete di test venisse implementata per mezzo di solo due nodi con uno dei quali utilizzato come strumento analizizzatore di traffico, la demoboard che genera i messaggi di test non riceverebbe mai l'ack

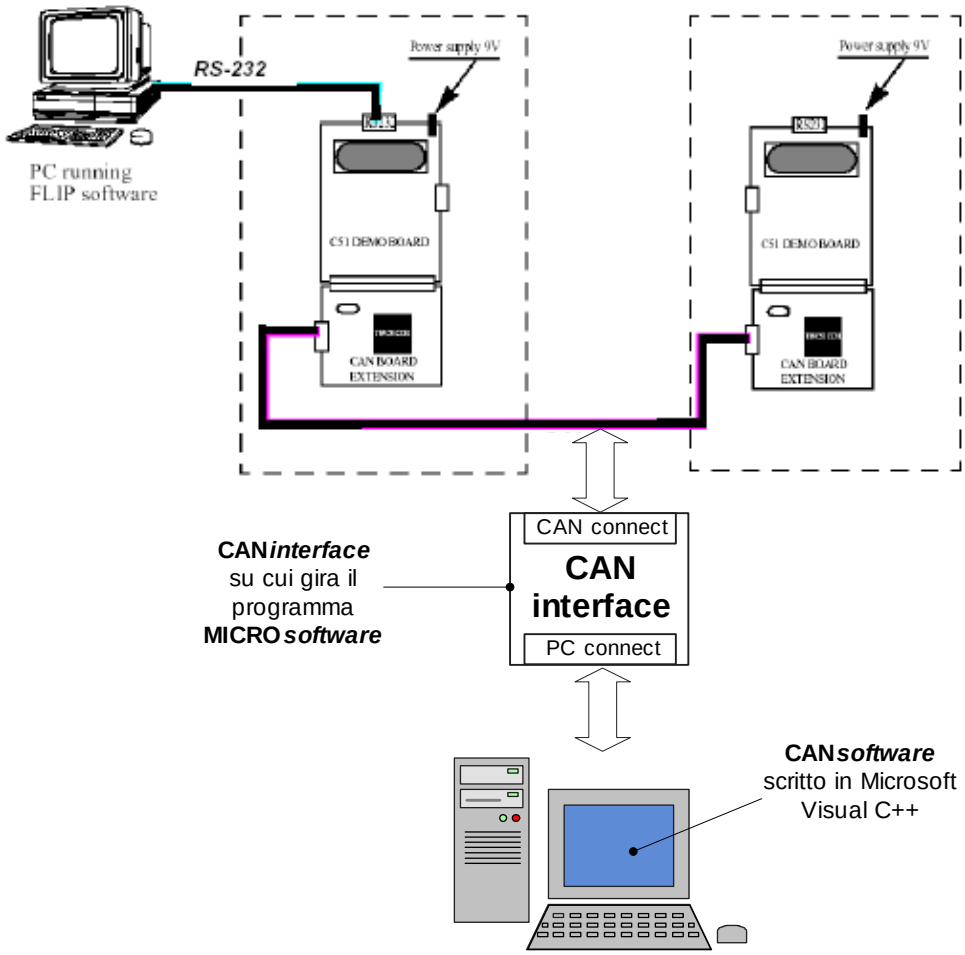


Figura 3.35: Rete CAN realizzata per il test del sistema.

ai frame che invia (non ci sono nodi che possono rispondere). Per le regole di Fault Confinement[32] il nodo continuerebbe ad inviare il solito frame sul bus all'infinito.

In conclusione con soli due nodi non siamo in grado di generare sulla rete un traffico significativo per il test dell'analzzatore (si avrebbe a disposizione un unico frame trasmesso in modo ciclico); da qui l'esigenza di realizzare un supporto hardware ex-novo per il nostro CAN*analyser*.

[32] Al reset il nodo è in stato di "Error Active" (TEC e REC ≤ 127). L'ACK Error che rileva ad ogni nuova trasmissione incrementa di 8 il contatore TEC; quando TEC supera il valore 127 il nodo diventa Error PASSIVE, a questo punto TEC non viene più incrementato ed il nodo non raggiungerà mai la condizione di Bus-off e continuerà a trasmettere il solito frame. [10]

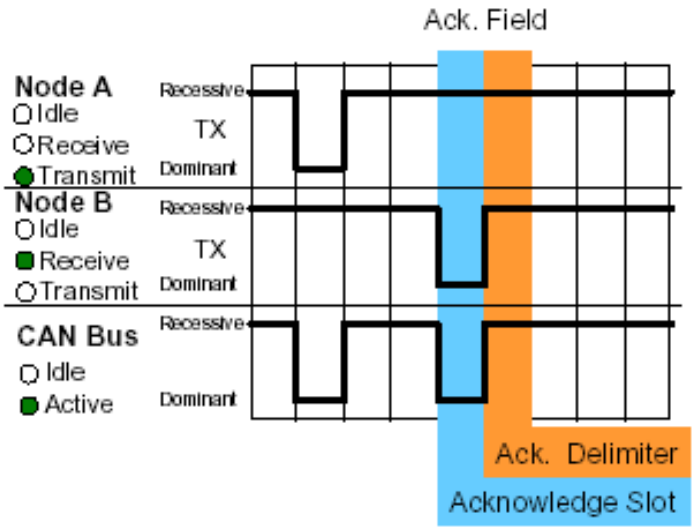


Figura 3.36: Processo di acknowledgment check del messaggio in una rete CAN.

Bibliografia

[1] Robert Bosch Gmbh. *CAN specification*, 2.0 edition, 1991.

[2] CAN in Automation Users Group (CiA). www.can-cia.org.

[3] CAN in Automation (CiA). *CAN Physical Layer*. www.can-cia.org.

[4] CAN in Automation (CiA). *CAN Data Link Layer*. www.can-cia.org.

[5] Motorola Ltd. *Bosch Controller Area Network (CAN) Version 2.0*, third edition. www.motorola.com.

[6] ATMEL Corporation. *CAN Extension Board - Demo Kit User Guide*, 2003. www.atmel.com.

[7] Keil Elektronik Gmbh e Keil Software Inc. *Getting Stanted with uVision2*, 2001. www.keil.com.

[8] Keil Elektronik Gmbh e Keil Software Inc. *Release Notes for ISD51*, 2001. www.keil.com.

[9] Keil Elektronik Gmbh e Keil Software Inc. *Installing and Using the Keil Monitor-51*, 2000. www.keil.com.

[10] Robert Bosch GmbH. Fault confinement. In *CAN Specification version 2.0*, pages 24–25, 1991. www.can.bosch.com.

Capitolo 4

CAN*interface*

L'analizzatore di traffico CAN*analyser* è composto da diversi elementi tra cui l'hardware di interfaccia tra il Personal Computer e la rete CAN la cui descrizione è oggetto di questo capitolo; nella prima parte si motivano le scelte progettuali seguite nella sua realizzazione di CAN*interface* mentre nella seconda parte si approfondiscono i limiti dell'architettura e si forniscono alcune indicazioni per migliorare le prestazioni dell'interfaccia.

4.1 L'interfaccia hardware CAN*interface*

Sulla base di quanto già descritto nella sezione 2.4 le scelte da fare in fase di progettazione dell'interfaccia hardware riguardano:

- CPU,
- CAN controller,
- transceiver,
- memoria esterna (SRAM e/o EEPROM),
- protocollo e circuito di comunicazione con il PC.

CPU e CAN controller sono funzionalità integrate nel chip ATMEL T89C51CC01, come transceiver si è scelto un integrato che fosse in grado supportare lo standard ad alta velocità ISO 11898-2 per consentirci di testare l'analizzatore su un bus con un baudrate di 1Mbit/s. Relativamente agli altri due elementi si è cercato di conciliare esigenze differenti: da un lato la necessità di semplificare al massimo e ridurre al minimo i tempi di sviluppo del progetto, dall'altro fare scelte che consentono di raggiungere il risultato

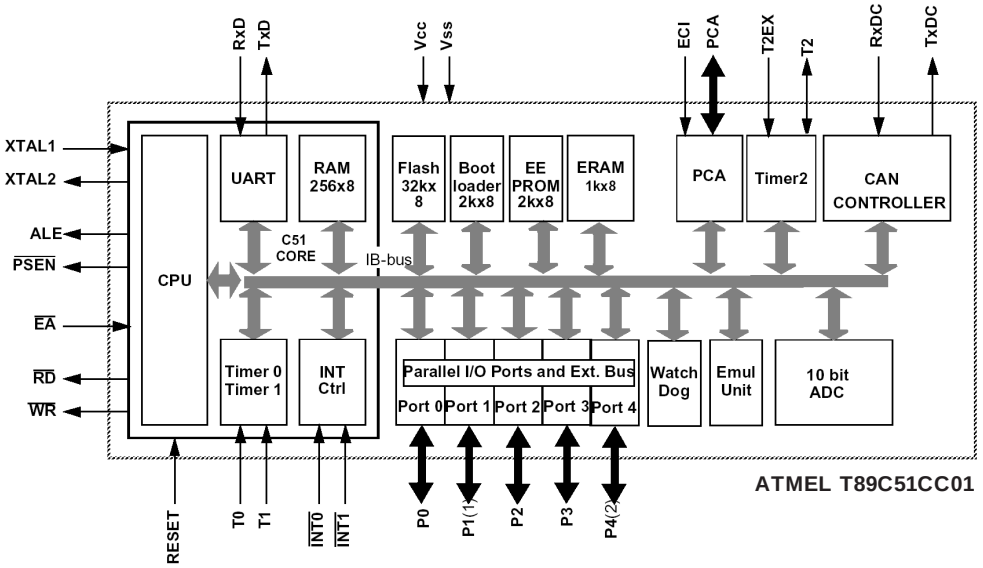


Figura 4.1: ATMEL T89C51CC01: schema funzionale a blocchi.

prefissato cioè l'acquisizione real-time dei dati. La memoria integrata nel microcontrollore (RAM per le variabili, EEPROM per codice e dati) è risultata più che sufficiente ai nostri scopi mentre come protocollo di comunicazione con il Personal Computer la scelta si è orientata sull'RS232 come unica soluzione immediatamente e facilmente implementabile. La scelta del protocollo di comunicazione con il PC ed in modo particolare l'RS232 costituisce un collo di bottiglia al flusso delle informazioni tra interfaccia e Computer; la RS232 per effetto della limitata banda che mette a disposizione del sistema costituisce un fattore limitante delle performance di CAN*analyser*; l'ultima sezione di questo capitolo è stata dedicata allo studio di come sia possibile, senza stravolgere più di tanto l'architettura attuale superare questo limite.

4.1.1 Il microcontrollore ATMEL T89C51CC01

Il T89C51CC01 è uno dei tre integrati appartenenti alla famiglia CANary della ATMEL, in tabella 4.1 sono elencate le periferiche disponibili in questi μC mentre di seguito sono descrivere alcune delle funzionalità di questi integrati limitando l'attenzione a quelle che sono state sfruttate nella realizzazione della CAN*interface*[1].

[1]Altre funzionalità che però che sono comunque di un certo interesse sono: il supporto per la Time Trigger Communication (utilizzabile per l'implementazione di reti TTCAN) e l'Automatic Reply Mode dell'interfaccia seriale.

		C51CC02	**C51CC01**	C51CC03
Memory	Flash Code	16 KB	**32 KB**	64 KB
	Flash Boot	2 KB	**2 KB**	2 KB
	EEPROM	2 KB	**2 KB**	2 KB
	RAM	0.5 KB	**1.2 KB**	2.2 KB
CAN Controller	Mess. Objects	4	**15**	15
Self-programming Capability	CAN	Yes	**Yes**	Yes
	UART	Yes	**Yes**	Yes
On-chip Peripherals	16-bit Timers	3	**3**	3
	PCA Channels	2	**5**	5
	10-bit ADC	8 ch.	**8 ch.**	8 ch.
	21-bit Watchdog	Yes	**Yes**	Yes
	Double Pointer	Yes	**Yes**	Yes
Available	Max. Frequency	40 MHz	**40 MHz**	40 MHz
	I/O Pins	20	**34**	34
	Power Supply (V)	2.7 - 5.5	**2.7 - 5.5**	2.7 - 5.5

Tabella 4.1: Caratteristiche dei chip ATMEL della serie CANary a confronto.

SFR La presenza di uno *Special Function Register* è una caratteristica comune ai micro della famiglia C51; attraverso i registri di che compongono l'SFR vengono gestite la quasi totalità delle funzioni del processore. Nel caso specifico del T89C51CC01 34 di questi registri sono utilizzati per la gestione del CAN controller integrato.

Mailbox Una mailbox o "canale" è un'area di memoria RAM che viene gestita dal CAN controller per memorizzare le informazioni relative ai messaggi che il micro deve inviare o a ricevuto. Come si vede in figura 4.2 ogni canale occupa 20 byte: identificatore e maschera (4+4), dati (8), timestamp (2), registro di configurazione (1) e registro di stato (1); l'ampia area di memoria che ne deriva (nel T89C51CC01 abbiamo 15 canali e quindi 300 byte di cui 120 sono riservati al campo DATA dei frame) viene gestita in modo efficiente attraverso un sistema di paginazione per mezzo di alcuni registri di SFR[2]. Ognuna di queste mailbox, indipendentemente dalle altre, può essere configurata in uno dei seguenti modi:

[2]CANPAGE è il registro utilizzato per selezionare uno dei 15 message object, attraverso il campo INDEX dello stesso registro si seleziona uno degli 8 byte di cui è composto il messaggio che viene caricato nel registro CANMSG. I registri "Message Object Control" (CANCONCH), "Message Object Status" (CANSTCH), "Message Time Stamp" (CANTIMH e CANTIML), "ID Tag" e "ID Mask" diventano disponibili per l'oggetto selezionato.

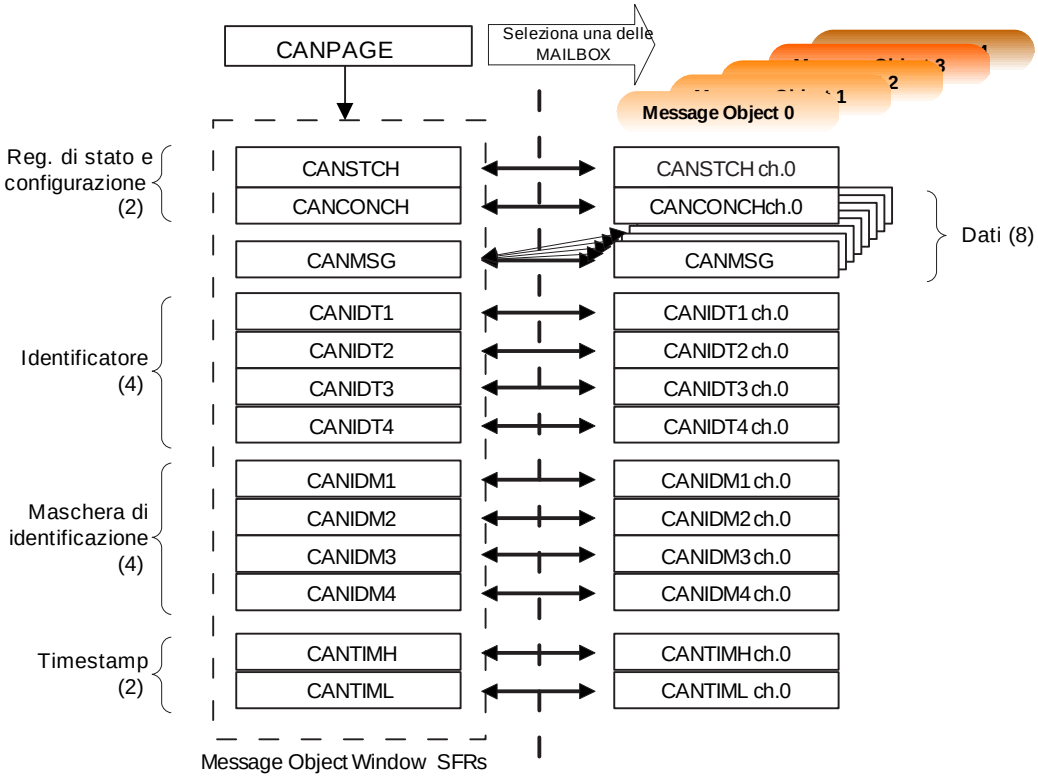


Figura 4.2: ATMEL T89C51CC01: mailbox ed gestione della memoria tramite registri SFR.

- in trasmissione,
- in ricezione,
- come buffer di ricezione,
- disabilitata.

Acceptance Filter Attraverso il contenuto dei registri ID Tag e ID Mask il micro costruisce il filtro attraverso cui vengono selezionati i frame da caricare nei canali di ricezione. L'ID Tag quando è assegnato ad una mailbox configurata in trasmissione costituisce l'identificatore del frame.

Quando uno o più canali vengono configurati come "buffer di ricezione" devono avere lo stesso *acceptance filter*; l'ultimo frame ricevuto viene sempre memorizzato nella prima mailbox libera facente parte del buffer[3] (quella con indice più piccolo tra quelle vuote). Quando il buffer in

[3]Questa caratteristica viene sfruttata quando si cerca di leggere dal bus cercando di

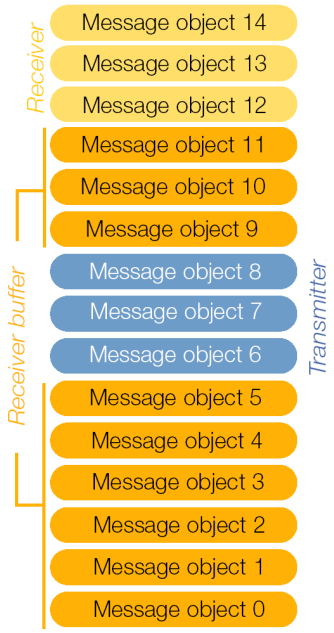


Figura 4.3: Esempio di configurazione di un CAN buffer.

ricezione è pieno e viene riconosciuto un nuovo frame il flag OVERBUF (OVER BUFfer) del registro CANGIT (CAN General INterrupt) viene settato e se il micro è stato adeguatemante configurato viene generata un'interruzione di programma. Ogni volta che un canale di ricezione viene caricato con un messaggio viene anche automaticamente disabilitato, se si verifica un overflow significa che tutti i canali che costituiscono il buffer risultano disabilitati; i frame che arrivano al controller da quel momento in avanti vengono "persi[4]".

CAN Interrupt La generazione di una interruzione di programma può essere attivata al verificarsi di una delle seguenti condizioni:

- corretta ricezione di un messaggio;
- avvenuta trasmissione del messaggio;
- buffer di ricezione pieno;

perdere meno messaggi possibili o quando si gestisce il trasferimento in burst di grosse quantità di informazioni; a tale proposito si ricorda che il campo DATA dei frame CAN è in grado di contenere al massimo 8 byte, il trasferimento di pacchetti di dimensione maggiore richiede una specifica implementazione da parte del progettista.

[4]Si è fatta l'ipotesi che tutti i canali utilizzati in ricezione siano configurati come buffer.

- al verificarsi di una delle condizioni di errore previste dal protocollo: *Bit Error, Stuff Error, Ack Error, Form Error* e *CRC Error*.

Autobaud e Listening Mode Quando questa modalità è attiva il CAN controller è in "ascolto" sul bus ed in questa condizione non viene generato alcun acknowledge alla ricezione di un messaggio, non vengono gestiti i contatori TEC e REC, il CAN controller non può inviare messaggi di alcun tipo mentre continuano ad essere aggiornati i flag di errore; è piuttosto facile comprendere perché si parli di "Listening Mode".

Il fatto che gli errori continuino ad essere rilevati mentre gli error counter sono "congelati" rende possibile implementare via software routine che sono in grado di determinare procedendo per tentativi una configurazione di bit timing adeguata al bitrate incognito della rete (*good configuration find*), in questo modo il programmatore ha la possibilità di fornire al nodo funzionalità di *Autobaud* e di *Hot-plugging* (il nodo può essere collegato ad una rete con bitrate sconosciuto mentre i nodi del bus continuano a scambiarsi messaggi).

UART Caratteristica comune ai micro delle famiglie 80C51 e 80C52 è la disponibilità di una porta di comunicazione seriale di tipo full duplex in cui ricezione e trasmissione possono avvenire contemporaneamente anche a differenti bitrate. Il circuito di gestione della porta seriale è una *Universal Asynchronous Receiver/Transmitter*[5] (UART) in grado di lavorare in quattro distinte modalità[6] (0, 1, 2 e 3) e dotata di un buffer di ricezione che consente di iniziare l'acquisizione di un nuovo dato ancora prima che il precedente sia stato letto dalla CPU. L'accesso ai registri di trasmissione e ricezione avviene per mezzo di un unico registro di SFR chiamato SBUF (Serial BUFfer register). Altre funzionalità disponibili sono:

- "Framing Error Detection": quando questa funzione è attiva ogni frame ricevuto viene analizzato alla ricerca di un bit di stop, se per qualche motivo questo bit non viene trovato (per effetto del

[5]Gestisce la serializzazione dei byte da trasmettere e la ricostruzione del dato ricevuto per mezzo di opportuni shift register.

[6]Le modalità 1, 2 e 3 sono asincrone e si distingono per il numero dei bit che compongono la parte dati della trama e per la frequenza di lavoro fissa o programmabile; i modi 2 e 3 consentono anche la gestione di comunicazioni multiprocessore. In modalità 0 trasmissione e ricezione avvengono in modo sincrono attraverso un unica linea collegata al pin RxD del micro.

rumore o altro) viene settato il *Framing Error bit* (FE) nel registro SCON (Serial CONtrol register).

- "Automatic Address Recognition": si tratta di una funzione attiva solo se viene abilitata la comunicazione multiprocessore. Alla ricezione di un nuovo frame il dato viene confrontato con l'indirizzo costruito a partire dai registri SADEN (Slave Address Mask register) e SADDR (Slave ADDREss register), solo se questo controllo da esito positivo il bit RI (*Receive Interrupt flag*) di SCON viene settato[7].

In-System Programming L'ISP consente per mezzo del *bootloader* che fornisce tutte le API necessarie di programmare e riprogrammare il microcontrollore senza rimuoverlo dal sistema in cui è inserito e senza la necessità di preprogrammarlo con un'applicazione dedicata a tale scopo. La stessa ATMEL fornisce il suo T89C51CC01 in due versioni: nella prima il bootloader precaricato[8] fornisce pieno supporto alla riprogrammazione tramite UART, la seconda versione implementa questa funzionalità tramite bus CAN.

In figura 4.4 è riportato lo schema di collegamento del microcontrollore nell'interfaccia CAN*interface* e nei paragrafi seguenti si dettagliano le funzionalità svolte dai singoli componenti (per conoscere più in dettaglio il significato dei singoli pin del T89C51CC01 si può fare riferimento a quanto riportato in appendice A o per una descrizione più dettagliata ci si può riferire direttamente al datasheet [7]).

4.1.1.1 Il circuito di generazione del clock

Il core del μC, in modo del tutto compatibile con gli altri processori della serie C51, richiede 12 periodi di clock per ciclo macchina; l'ATMEL T89C51CC01 supporta un'ulteriore modalità di funzionamento indicata con X2[9] che attraverso un moltiplicatore di frequenza collocato tra l'ingresso XTAL1 e l'in-

[7] Questo controllo viene eseguito via hardware e quindi risulta estremamente rapido. Per mezzo di questo meccanismo solo la CPU correttamente indirizzata viene interrotta dalla ricezione del frame, gli altri slave non vengono "disturbati" grazie anche al fatto che la verifica è eseguita da un circuito dedicato e quindi non ruba cicli di clock all'esecuzione del codice che gira sulle CPU non indirizzate.

[8] Nei microcontrollori della serie CANary esiste un'area di memoria (FM1) dedicata a contenere il bootloader fornito dalla ATMEL e detto "standard" per distinguerlo dall' "user" bootloader realizzato dall'utente.

[9] Questa funzionalità viene gestita via software per mezzo del bit omonimo *X2* che si trova nel registro CKCON ClocK CONtrol register oppure via hardware attraverso il fuse bit X2B.

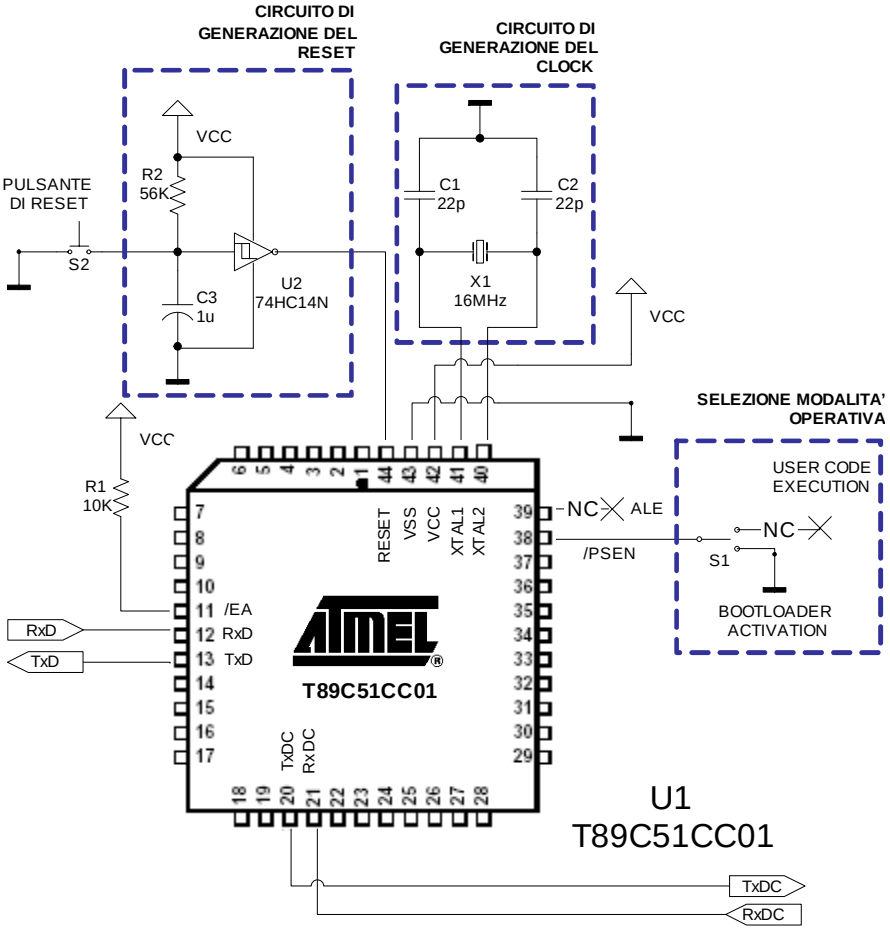


Figura 4.4: CAN*interface*: schema di collegamento del microcontrollore ATMEL T89C51CC01.

gresso del circuito di clock del core consente di raddoppiare la frequenza di lavoro impostata esternamente: *attivata la modalità X2 è come se il micro lavorasse con 6 periodi di clock per istruzione* (prendendo come riferimento il clock esterno).

Le singole periferiche del micro: CAN controller, WatchDog, Programmable Counter Array, i Timer e l'interfaccia seriale, possono essere singolarmente abilitate a lavorare in questa modalità per mezzo dei relativi bit nel registro di configurazione CKCON: "CANX2", "WDX2", "PCAX2", "SIX2", "T2X2", "T1X2" e "T0X2".

Per far lavorare l'interfaccia seriale ad un bitrate di 115.2 Kbit/s (il massimo che supporta la UART del PC) si è dovuto sfruttare il TIMER2 del micro come generatore del baudrate e la formula che ci consente di calcolarlo a partire dai valori con cui vengono caricati i registri RCAP2H e RCAP2L è la seguente:

$$Baudrate_{mode\ 1,3} = \frac{F_{osc}\ 2^{\overline{T2X2}}}{32\ [65536 - (RCAP2H, RCAP2L)]} \quad [bit/s]$$

dove:

F_{osc} : è la frequenza dell'oscillatore locale;

$\overline{T2X2}$: vale 1 se il TIMER2 ha attivata la modalità X2 viceversa vale 0.

RCAP2H, RCAP2L : è il contenuto rispettivamente della parte alta e bassa dei registri con cui è caricato il contatore del TIMER2. Nella formula precedente $(RCAP2H, RCAP2L)$ indicano l'intero non segnato a 16 bit corrispondente all'unione dei due byte.

Sfruttando la modalità X2 ed una frequenza di oscillazione di 16MHz si riesce ad ottenere un baudrate prossimo al valore richiesto (115.2Kbit/s) con un errore percentuale che si è rilevato tale da non pregiudicare la comunicazione[10].

In figura 4.5 è riportato il dettaglio del circuito di clock tramite cui si imposta la frequenza di lavoro del microcontrollore e delle sue periferiche.

XTAL1 e XTAL2 sono rispettivamente l'ingresso e l'uscita di un invertitore a stadio singolo (vedi appendice A) che può essere sfruttato per realizzare un oscillatore aggiungendo esternamente all'integrato un limitato numero di componenti tra cui un quarzo che viene utilizzato per controllare la reazione positiva con cui si generano le oscillazioni.

[10] Il baudrate generato utilizzando TIMER2 è pari a 111.1Kbit/s da cui segue un errore relativo percentuale del 3.55%.

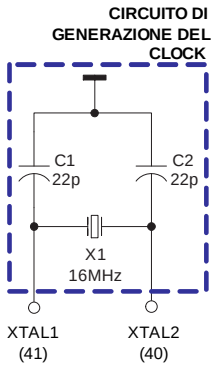


Figura 4.5: CAN*interface*: schema del circuito di generazione del clock.

4.1.1.2 Il circuito di reset

Il reset viene ottenuto mantenendo il relativo pin del μC (44) a livello logico alto per almeno 2 cicli macchina (24 periodi della frequenza di clock[11]) *mentre l'oscillatore è in funzione* e con la tensione di alimentazione VCC *stabile in un range operativo di valori*; la CPU come conseguenza esegue un ciclo interno di reset[12] che viene ripetuto finché il pin 44 non viene riportato a livello logico basso.

La gestione del reset[13] è richiesta quando il sistema viene alimentato ed ogni qual volta sia necessario passare dalla fase di programmazione del micro a quella di esecuzione del codice applicativo e viceversa.

In figura 4.6 è riportato lo schema utilizzato per il reset in CAN*interface*. Secondo quanto descritto in appendice A dove è stato riportato lo schema del circuito interno al micro posto sull'ingresso del pin di RESET, sarebbe sufficiente un condensatore collegato tra VCC ed il pin (44) per garantire una gestione corretta di questa funzionalità; l'utilizzo del Trigger di Schmitt presente nel nostro circuito garantisce maggiore sicurezza nella gestione dei "rimbalsi" del segnale che vengono prodotti ogni qual volta l'operatore agisce sul pulsante S2 per resettare l'interfaccia.

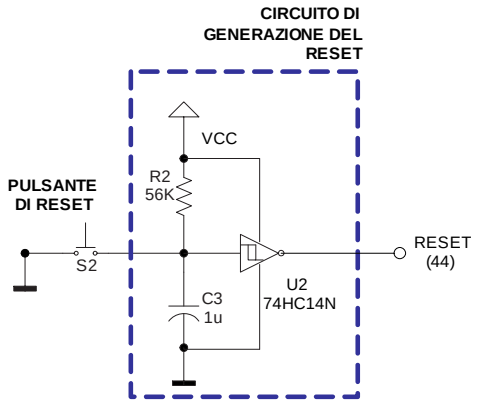


Figura 4.6: CAN*interface*: schema del circuito di generazione del reset.

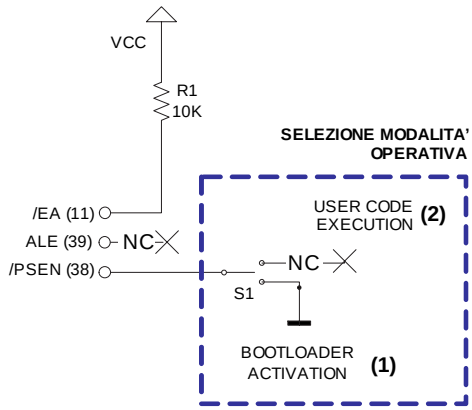


Figura 4.7: CAN*interface*: circuito di gestione delle modalità di programmazione/esecuzione.

4.1.1.3 Attivazione della modalità di programmazione

La funzione ISP che il μC fornisce, consente con pochissimo sforzo aggiuntivo in termini di progetto e di componenti elettronici (vedi figura 4.7) di sfruttare la CAN*interface* anche per programmare l'integrato tramite seriale UART sfruttando il bootloader standard fornito dalla ATMEL e il programma "FLIP". Se il deviatore S1 è in posizione:

[11] I periodi sono 12 se in fase di programmazione viene settato il fuse bit X2B che attiva la modalità X2 via hardware di default al reset del micro.

[12] Il reset non ha effetto sulla memoria RAM interna, una volta alimentato il micro il contenuto della RAM è indeterminato (casuale).

[13] Una sequenza di reset mal gestita può determinare una errata inizializzazione ed i registri di sistema come il Program Counter possono non essere correttamente caricati determinando un comportamento imprevedibile in fase di esecuzione del codice utente.

1 viene selezionata la *modalità di programmazione*: al reset successivo il microcontrollore manda automaticamente in esecuzione il bootloader "standard" caricato nell'area di memoria FM1. Nel nostro caso il μC utilizzato dispone di un bootloader di tipo UART che consente la programmazione del chip attraverso l'interfaccia seriale.

2 viene selezionata la *modalità operativa* nella quale al successivo reset del sistema viene mandato in esecuzione il codice utente.

In appendice B.2 si descrive in modo più approfondito la funzione di ISP e come questa possa essere attivata via hardware al reset del sistema.

4.1.2 La memoria

Il T89C51CC01 mette a disposizione dell'utente una memoria RAM interna per la gestione dei dati (Data Memory) divisa per metodologie di indirizzamento in 3 distinti segmenti che definiscono un'area totale di 1.2Kbyte:

- 128 byte ad accesso diretto;
- 128 byte ad accesso indiretto;
- 1024 byte di memoria estesa ad accesso indiretto tramite istruzione dedicata.

In figura 4.8 sono indicati i tre segmenti in cui è suddivisa la RAM interna ed è riportata anche la RAM esterna (XRAM) che il micro è in grado di gestire. In modo particolare i primi 128 byte sono organizzati in 4 banchi di 8 registri (R0..R7), i successivi 16 bit costituiscono la "bit addressable memory space[14]", questa organizzazione consente una più efficiente gestione dello spazio disponibile e consente di sfruttare istruzioni ad accesso diretto con cui velocizzare l'esecuzione del codice.

Il T89C51CC01 fornisce on-chip anche una EEPROM di 2Kbyte dedicata ai dati (EEPROM Data Memory), i tempi di accesso a quest'area di memoria sono però molto alti e ciò la rende praticamente inutile come spazio in cui salvare temporaneamente dati che il processore deve elaborare o che nel caso del nostro analizzatore vengono acquisiti in tempo reale dal bus[15].

[14]Si tratta di un'area che l'utente può sfruttare in modo estremamente efficiente per mezzo delle istruzioni di tipo "single bit" del C51. Le istruzioni "single bit" compiono operazioni su singoli bit di dato e sono progettate per avere accesso diretto a quest'area di memoria; la stessa operazione eseguita tramite OR o AND su l'intero byte richiede molto più tempo di esecuzione.

[15]La EEPROM è invece adatta ad altri impieghi come la memorizzazione dei parametri di autoconfigurazione del sistema e più in generale di tutti i dati che non devono essere manipolati frequentemente.

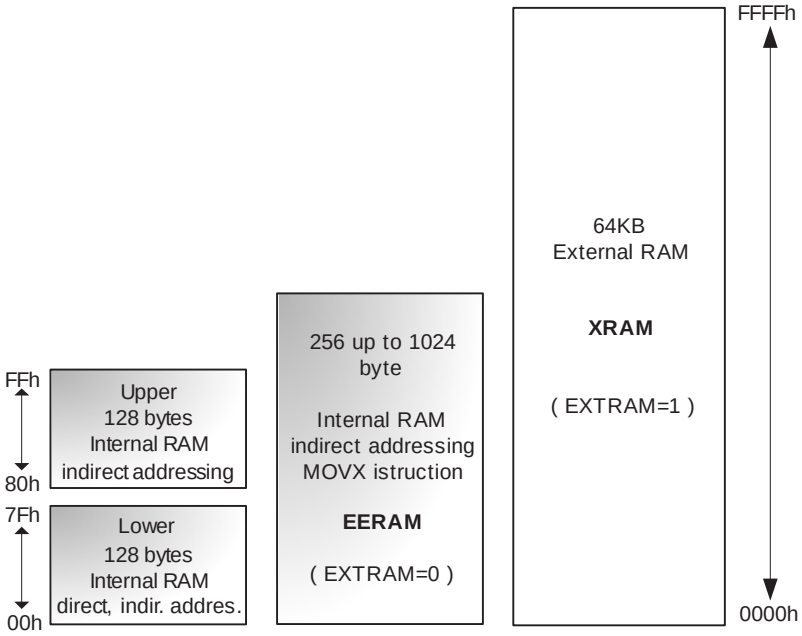


Figura 4.8: ATMEL T89C51CC01: organizzazione della memoria RAM.

In ultima analisi il T89C51CC01 dispone di 32Kbyte di memoria "program/code" indicata con FM0 di dimensione più che sufficiente per contenere MICRO*software* e una flash memory di 2Kbyte indicata con FM1 dedicata a contenere il bootloader "standard" e le API (Application Programming Interface) di supporto all'ISP. Una descrizione più approfondita sull'organizzazione e gestione della flash memory del micro è fornita in appendice B.1.

4.1.3 Il transceiver Philips PCA82C250

Il transceiver Philips PCA82C250 fornisce l'interfaccia tra il controller CAN e la linea di trasmissione; perfettamente compatibile con lo standard ISO 11898-2 è adatto ad essere impiegato per la realizzazione di connessioni con reti CAN ad alta velocità.

Oltre alle caratteristiche sinteticamente riportate in tabella 4.2 [8] il transceiver scelto implementa le seguenti funzionalità:

- con riferimento allo schema di figura 4.9, le linee "CANH" e "CANL" sono protette da fenomeni elettrici transitori particolarmente frequenti quando l'integrato viene impiegato in applicazioni automotive.

	PCA82C250
Max data rate	1 Mbit/s
Short circuit (V)	-8 +18
Transient (V)	-150 +100
ESD (kV)	2
Thermal Protection	Yes
Slope Control	Variable
CMR (V)	-7 +12
Delay (ns)	< 170
Fan Out	> 110
Supply Voltage (V)	5
Supply Current (mA)	< 70
Stand-by current (μA)	< 170
Rev. Leakage current (mA)	2

Tabella 4.2: Philips PCA82C250: elenco delle caratteristiche tecniche del transceiver.

- Attraverso il pin "Rs" (8) è possibile selezionare tre differenti modi operativi:
  - in modalità *High-speed* i transistori che pilotano le linee CAN in uscita (in trasmissione) vengono semplicemente accesi e spenti nel modo più veloce possibile cioè senza limitare i tempi di discesa e salita dei segnali;
  - quando è attivo lo *Slope-control mode* i tempi di salita e discesa sono controllati per mezzo della resistenza connessa al pin 8, in questo modo è possibile ridurre e tenere sotto controllo i problemi legati alle interferenze a radiofrequenza (RFI).
  - Mantenendo il pin 8 ad un valore "alto" di tensione il transceiver viene mandato in *Standby mode* dove lo stadio di trasmissione viene spento mentre quello di ricezione funziona in uno stato di minima dissipazione di corrente (non può essere spento del tutto visto che viene utilizzato per riattivare la modalità operativa "normale").

Il PCA82C250 in CAN*interface* viene sfruttato in modalità ad alta velocità ed a tale scopo il pin Rs mostrato in figura 4.10 è connesso a massa. Il CAN controller è collegato con il transceiver attraverso le due linee seriali corrispondenti ai pin 1 (TXD) e 4 (RXD) che vengono cortocircuitati con le

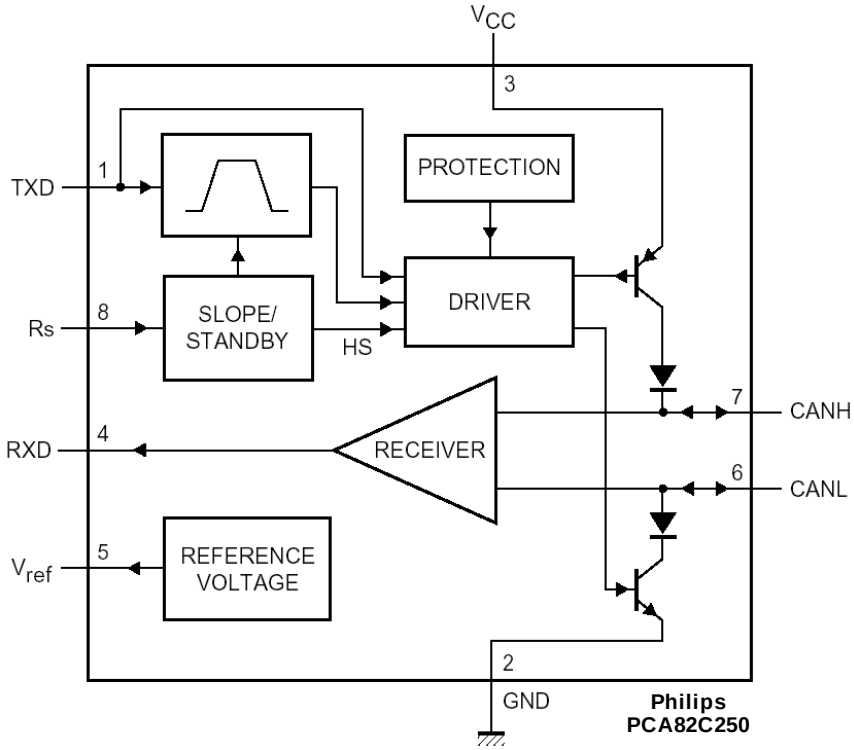


Figura 4.9: Philips PCA82C250: schema funzionale a blocchi.

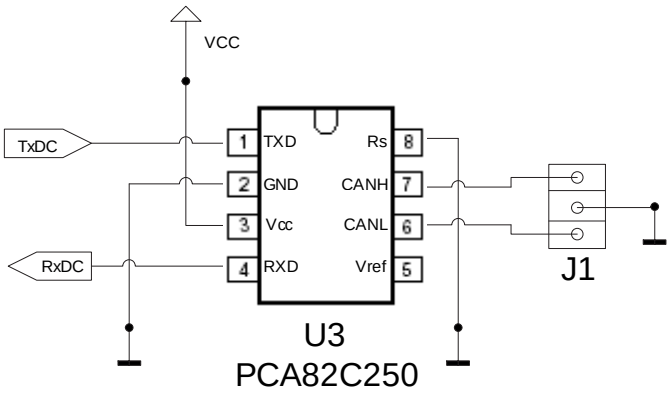


Figura 4.10: CAN*interface*: schema di collegamento del transceiver Philips PCA82C250.

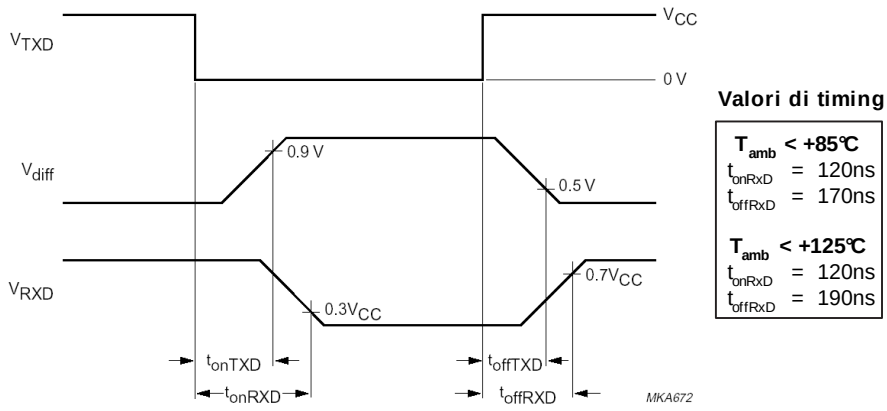


Figura 4.11: Philips PCA82C250: diagramma di temporizzazione del transceiver.

linee dedicate dal controller al collegamento con il transceiver: TxDC e RxDC. La connessione dal lato CAN avviene attraverso i due terminali CANH e CANL per mezzo dei quali la linea viene pilotata con una tensione differenziale conforme alle specifiche ISO di riferimento; l'interfaccia si connette fisicamente alla rete CAN sfruttando il connettore J1 di figura.

Dal punto di vista dei requisiti di bit-timing della rete CAN, in modalità *High-speed* il transceiver introduce un ritardo effettivo sull'anello di ritorno del segnale (vedi 3.1.7.1) inferiore a 145ns (sono 155ns a temperature di utilizzo superiori agli 85°C) come si ricava dall'espressione sottostante e dai valori riportati in figura 4.11.

$$t_{loop.eff} = max\{\frac{t_{onRxD} + t_{offRxD}}{2}, t_{onRxD}\} \quad [ns]$$

4.1.4 L'interfaccia di collegamento RS232 con il PC

Il microcontrollore T89C51CC01 come moltissimi μC oggi disponibili sul mercato ha integrata una "Serial Communication Interface" (SCI)[16] e questa UART viene sfruttata da CAN*interface* per collegarsi al PC.

[16]In fase di progettazione dell'integrato scegliere una SCI per far comunicare il chip con le periferiche esterne è una soluzione comunemente impiegata per ridurre il numero dei pin. Normalmente la SCI è realizzata con solo due linee: una per la ricezione ed una per la trasmissione (2 pin). La complessità di realizzazione fisica dell'integrato viene molto ridotta rispetto al caso in cui si scelga di implementare un'interfaccia parallela (8 pin, più un'eventuale segnale di Strobe).

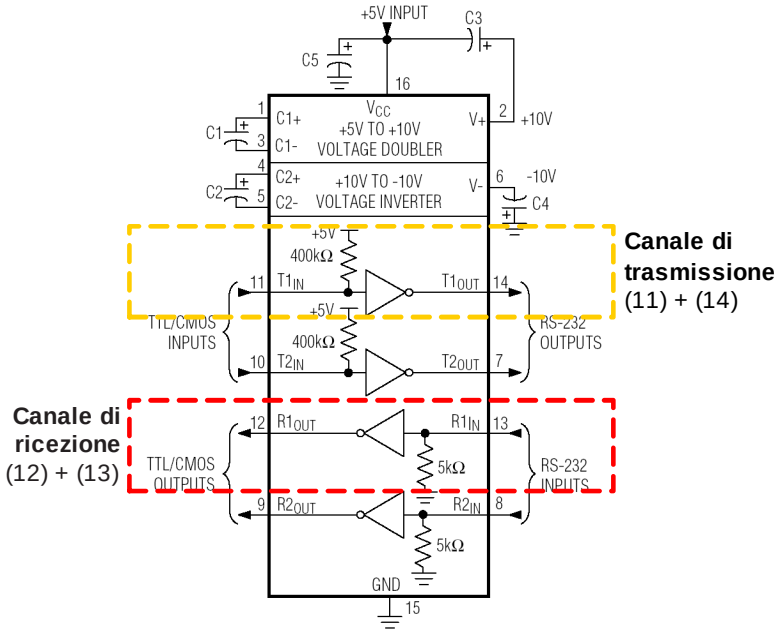


Figura 4.12: Maxim MAX232: schema funzionale a blocchi.

Il primo passo per connettere il μC alla porta COM del Personal Computer è quello di trasformare i livelli logici CMOS nei livelli di tensione compatibili con lo standard RS232[17], questo è il compito che svolge il convertitore di livello MAX232 prodotto dalla Maxim Semiconductor [9].

Nel MAX232 (figura 4.12) trovano posto due ricevitori e due trasmettitori ma soprattutto due pompe di carica che consentono di ottenere dall'alimentazione a 5V del chip due tensioni a $+10V$ e $-10V$ indispensabili per poter generare livelli compatibili con quelli richiesti dallo standard[18]; dovendo fornire solo i 5V richiesti dall'integrato il circuito di alimentazione diventa molto semplice.

Il costruttore dichiara che il chip è perfettamente compatibile con l'ultimo standard EIA 232-E, ciò ci mette al riparo da eventuali incompatibilità elettriche con la porta seriale di comunicazione del computer. L'importanza di utilizzare un buon convertitore è legata soprattutto al fatto che i valori di conversione generati hanno influenza sulla lunghezza massima del cavo di

[17]L'ultima revisione di questo standard è l'EIA 232-E, questa ulteriore revisione (E) dello standard originario non modifica le caratteristiche principali del protocollo di comunicazione.

[18]Lo standard RS232 codifica in trasmissione il livello logico 0 detto "Space State" tra $+5$ e $+15V$ ed il livello logico 1 detto "Mark State" tra -5 e $-15V$; in ricezione gli intervalli di tensioni diventano rispettivamente di $+5$, $+25V$ e -5 , $-25V$.

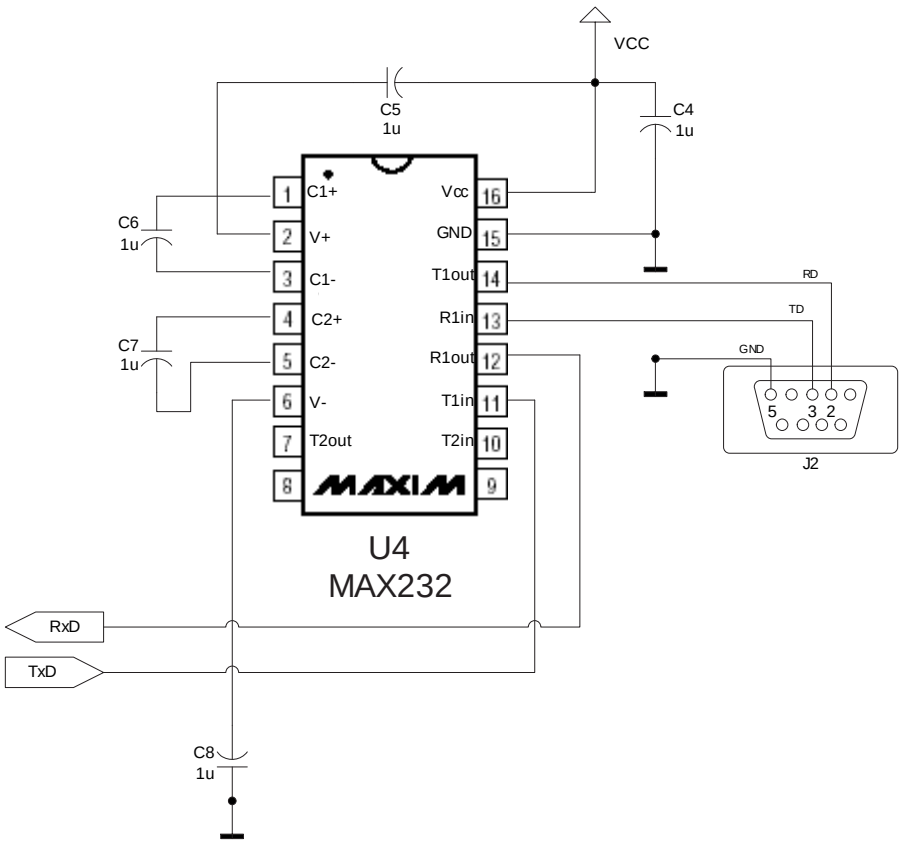


Figura 4.13: CAN*interface*: schema di collegamento del Maxim MAX232.

collegamento e sulle velocità di connessione permesse [19], in sostanza influenza fortemente la solidità del canale di comunicazione.

La CAN*interface* sfrutta solo uno (figura 4.12) dei due canali disponibili, lo schema utilizzato è riportato in figura 4.13 dove si è riportato anche il connettore DSub9 indicato con J2 utilizzato per la connessione dell'interfaccia al PC[20].

[19] Sono documentati casi in cui dispositivi identici che si distinguono solo per i valori di tensione in uscita dalla loro interfaccia seriale (ad esempio valori di "mark voltage" con differenze minime: $-9,3V$ e $-11,5V$), possono mostrare o meno difficoltà di connessione con la stessa periferica esterna.

[20] Per la connessione non deve essere utilizzato il classico cavo seriale "Null Modem" a coppie invertite (i segnali TD e RD sono internamente invertiti) ma un cavetto che deve fungere da semplice prolunga cioé i pin TD(3), RD(2) e GND(5) dal lato CAN*interface* devono essere collegati ai rispettivi TD, RD e GND della porta COM. L'inversione dei segnali è stata eseguita direttamente sulla scheda come è possibile notare dalla figura.

4.1.5 Il circuito di alimentazione

Il protocollo CAN è nato e stà raccogliendo sempre più ampia diffusione nel campo dell'automotive è quindi ovvio pensare a questo settore come primario campo di impiego di un analizzatore di traffico come CAN*analyser*, si è conseguentemente pensato di sfruttare la batteria dell'autovettura come fonte di alimentazione dell'interfaccia.

Elemento centrale del circuito in figura 4.14 è il regolatore di tensione ST L7805 [10] da cui dopo opportuno filtraggio si ricava una tensione continua e regolata di 5V che viene impiegata per alimentare l'intero sistema: microcontrollore, convertitore di livello, transceiver e trigger di Schmitt.

Il circuito di alimentazione e tutto ciò che sta a valle di esso sono componenti elettronici sensibili alle escursioni della tensione di alimentazione e devono quindi essere protetti dai transitori più "importanti" (agli altri ci pensa il regolatore), a tale scopo la tensione di batteria viene filtrata da un circuito di protezione realizzato con un diodo D1 che protegge dall'inversione dei morsetti e dal diodo zener DZ1 che limita alla propria tensione di breakdown (V_{BR} 33.3V) il valore massimo in ingresso al circuito di regolazione.

Sia l'alimentazione che i i segnali provenienti dal bus CAN presente nell'autoveicolo possono essere prelevati entrambi molto semplicemente attraverso il connettore di diagnostica (On-Board Diagnostic connector) conforme alle specifiche OBD-II presente su tutte le vetture di nuova produzione. Le connessioni da realizzare sono riportate in figura 4.14 (connettore J3) mentre informazioni più generali sullo standard OBD-II sono state riassunte in appendice D.

Con quest'ultimo circuito si è completa la descrizione dell'architettura di CAN*interface*, lo schema complessivo è riportato in figura 4.15.

4.2 CAN2USB*interface*: una nuova interfaccia seriale

Come accennato all'inizio di questo capitolo e come si vedrà ancora meglio più avanti l'utilizzo di una linea di comunicazione seriale RS232 limita pesantemente le prestazioni di CAN*analyser*.

Come superare i problemi di banda del protocollo RS232 senza stravolgere l'architettura hardware di CAN*interface* ed il codice di CAN*software* e MICRO*software*?

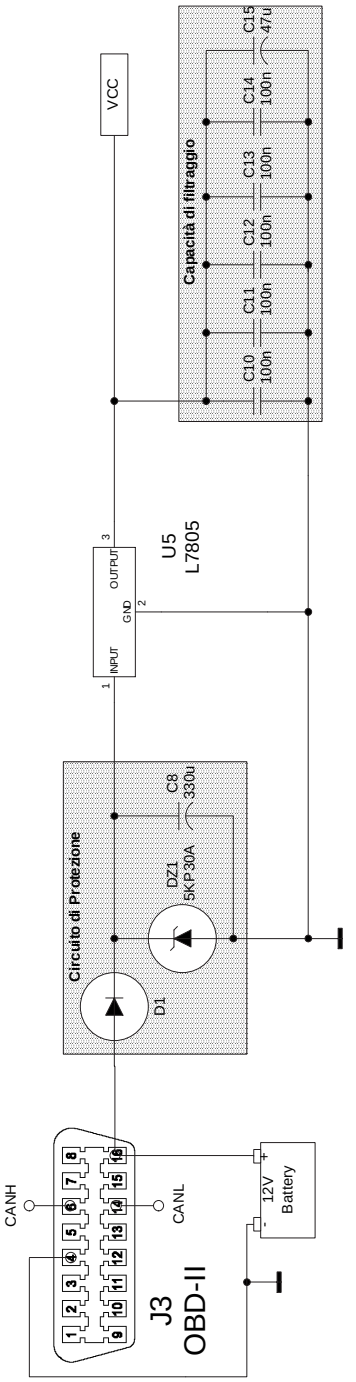


Figura 4.14: CAN*interface*: circuito di alimentazione.

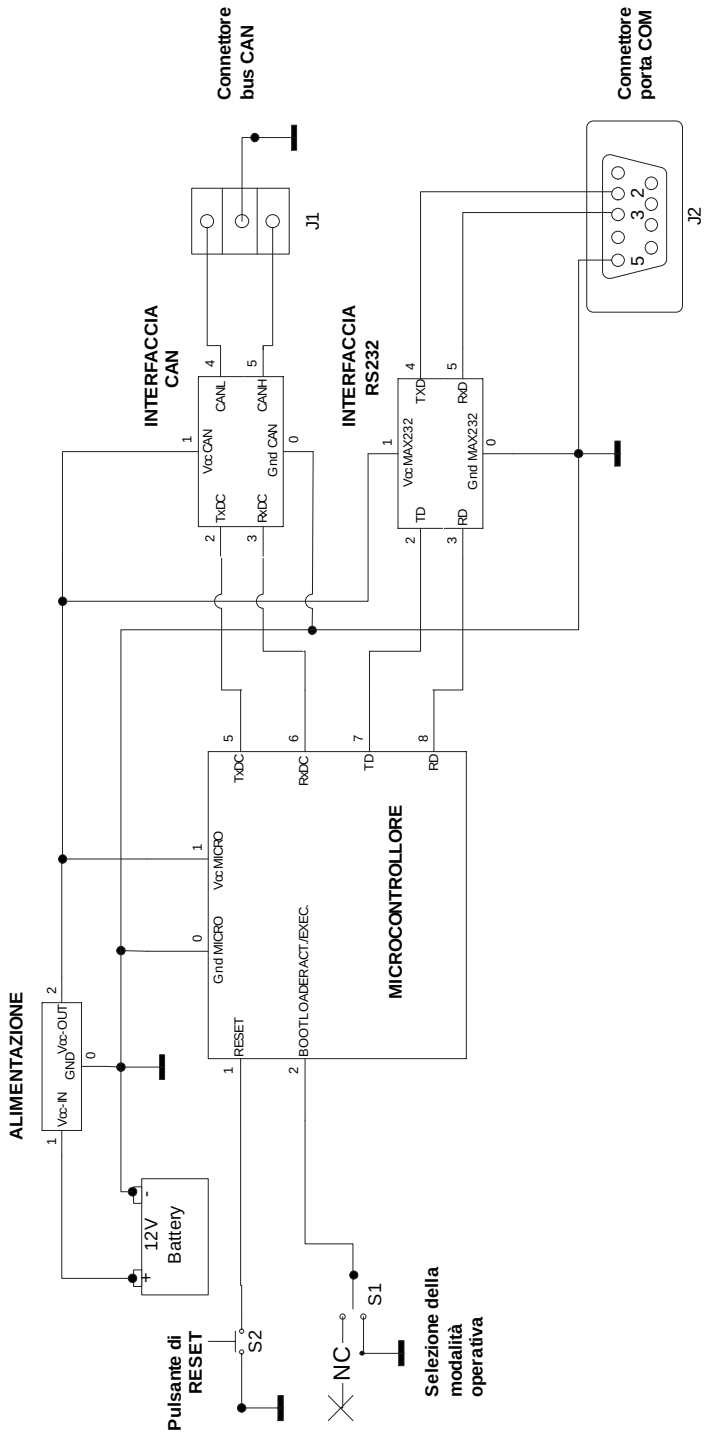


Figura 4.15: CAN*interface*: schema completo.

	RS-232	**USB1.1**	**USB2**
Speed	115.2Kbit/s	12Mbit/s	480Mbit/s
Relative Speed	Very Slow	Medium	Fast
Hot Inserition	No	Yes	Yes
Power	No	Yes 5V, 500mA	Yes 5V, 500mA
Mechanical Interface	Complex 9pin	Simplex[21]	Simplex[21]
Implementation	Simplex	Complex	Complex
Handshake	Yes/No	Yes	Yes

Tabella 4.3: Tabella comparativa caratteristiche RS-232, USB.

La soluzione elaborata consiste nell'utilizzare un convertitore USB-RS232 come quello che si potrebbe progettare impiegando l'integrato FT8U232BM [11] prodotto dalla *Future Technology Devices Intl. Ltd. (FTDI)*

4.2.1 USB e RS232 a confronto

Nella tabella 4.3 sono riepilogate e confrontate le caratteristiche principali dei due protocolli; l'interfaccia USB abbina ad una banda passante molto più ampia dell'RS232 la predisposizione alla funzione di hot-plugging ed il vantaggio di poter alimentare direttamente dal bus i dispositivi ad essa collegati senza quindi la necessità di prevedere plug esterni e cavi addizionali, per contro la gestione del protocollo è molto più complessa.

4.2.2 Caratteristiche del convertitore FT8U232BM

L'FT8U232BM appartiene alla seconda generazione di convertitori USB-UART prodotti dalla Future Technology; di tutto ciò che riguarda il protocollo USB e quindi: "numeration", "device descriptor", "endpoint", "token" e "message pipe" si occupa l'integrato ed i driver che vengono forniti dalla stessa casa produttrice[22] per semplificare la produzione delle applicazioni lato PC. Di seguito sono riportate alcune delle caratteristiche di rilievo di questo integrato e che risultano di particolare interesse nello sviluppo del-

[22]Nel caso in cui il progettista voglia impostare i propri PID (Product ID), VID (Vendor ID), "product string" o un "Serial Identification Number", l'FT8U232BM può essere interfacciato con una EEPROM seriale esterna dalla quale recupera queste informazioni che vengono poi gestite nel rispetto dello standard USB.

la nuova interfaccia a cui ci riferiremo da qui in avanti con il termine di CAN2USB*interface*.

- Supporto del protocollo RS232 fino ad 1Mbit/s;
- supporto del protocollo RS422/RS485 fino a 3Mbit/s;
- completa gestione dell'UART con possibilità di selezionare: 7/8 bit per i dati, 1/2 bit di stop e il bit di parità che può essere impostato in modalità : "Odd", "Even", "Mark", "Space", "None";
- buffer di ricezione FIFO di 384byte ed un buffer di trasmissione FIFO da 128byte;
- gestione della modalità di alimentazione tramite bus di tipo High-Power (massimo 500mA);
- compatibile con gli standard USB1.1 e USB2.0.

4.2.3 Come modificare CAN*interface*

Per quanto detto l'FT8U232BM non richiede alcuna programmazione, si occupa in tutto e per tutto della gestione del protocollo lasciando libero il progettista di curare le funzionalità del sistema in cui il convertitore viene inserito.

4.2.3.1 Modifiche hardware

Le modifiche da effettuare sono estremamente limitate: bisogna sostituire il convertitore di livello MAX232 (U4) con l'FT8U232BM aggiungendo i pochi componenti esterni richiesti dall'integrato e rimpiazzare il connettore DSub9 (J2) con un plug di tipo B conforme allo standard USB. La grande flessibilità e diffusione di di questo integrato della Future Technology a reso disponibili in commercio dei moduli che integrano sullo stesso PCB oltre all'FT8U232BM anche i componenti elettronici esterni richiesti per il suo funzionamento, il connettore USB ed eventualmente anche il socket per la EEPROM seriale. In figura 4.16 si è riportato a titolo di esempio lo schema del circuito "USB-MOD1" prodotto dalla *Raven Pty Ltd.* [12] e di cui riportiamo di seguito le principali caratteristiche.

- Non richiede componenti passivi esterni;
- integra un connettore USB di tipo B;

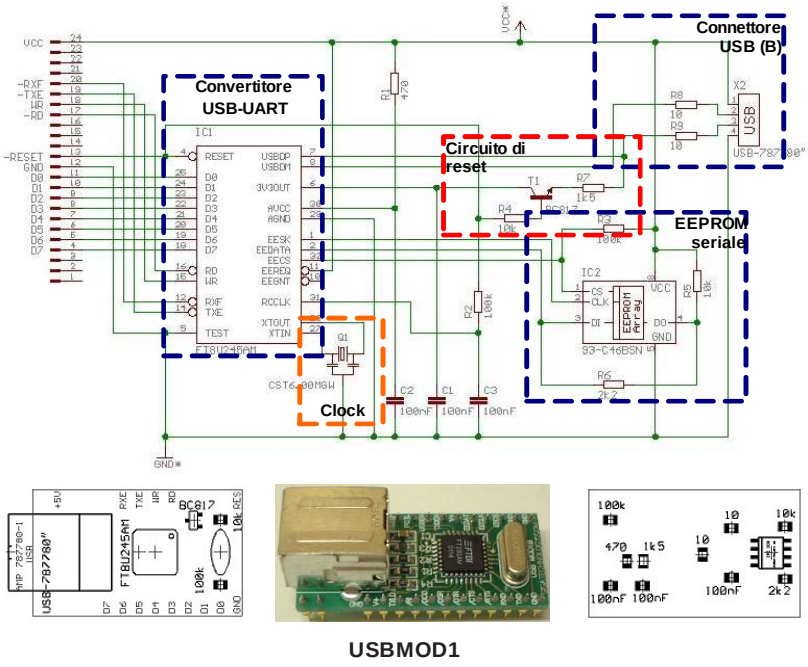


Figura 4.16: USBMOD1: schema del modulo prodotto dalla "Raven".

- integra i componenti necessari al collegamento di una EEPROM esterna utilizzata per l'operazione di "enumeration";
- il modulo di alimenta dalla porta USB;
- 450mA sono disponibili per l'alimentazione dei dispositivi collegati alla porta;
- può essere inserito in un socket DIL da 32 pin

Sfruttando uno di questi circuiti "già pronti" i tempi di sviluppo delle modifiche hardware da apportare a CAN*interface* diventano praticamente nulli, la nuova interfaccia CAN2USB*interface* è riportata in figura 4.17.

Il progetto si complica di poco se si vuole sfruttare la possibilità di alimentare l'intera interfaccia tramite la porta USB; in questa ipotesi verrebbe ad essere eliminato dallo schema originario anche il circuito di alimentazione che prevedeva l'utilizzo di un regolatore di tensione per stabilizzare e regolare la tensione di batteria e la serie di capacità inserite per ottenere una continua adeguata all'alimentazione dell'elettronica a valle. In tabella 4.4 si fornisce una descrizione sintetica dei pin del modulo che devono essere gestiti per la connessione a CAN*interface* nell'ipotesi di voler realizzare una configurazione USB di tipo "Bus Powered".

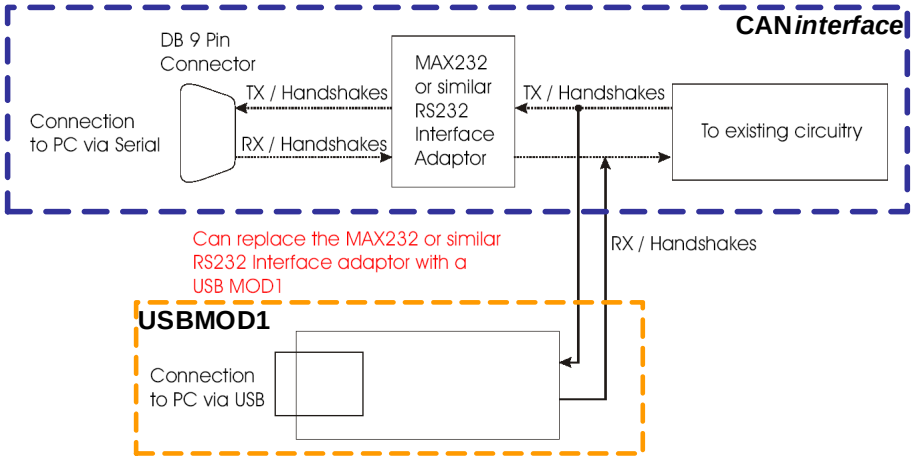


Figura 4.17: CAN2USB*interface*: schema funzionale.

Pin	Name	Type	Description
5	GND	PWR	Device - Ground supply pin
6	V+	PWR	Device - +4.4V to +5.25V Power supply pin
14	RXD	I	UART - Receive data input
15	TXD	O	UART - Transmit data output
16	GND	PWR	Device - Ground supply pin
17	GND	PWR	Device - Ground supply pin
18	GND	PWR	Device - Ground supply pin
22	V+	PWR	Device - +4.4V to +5.25V Power supply pin
24	USBEN	O	USB ENable - è posto a livello logico alto dopo che è terminata la fase di configurazione del dispositivo via USB
25	PCTL	I	Se tenuto a livello logico basso abilita la configurazione "Bus Powered" del bus USB (i device collegati alla porta possono prelevare l'alimentazione dal bus) in caso contrario deve essere tenuto alto
26	SLEEP	O	Và basso durante la fase in cui l'USB è in "Suspend Mode"; in questa condizione la corrente erogata attraverso il bus è limitata
28	GND	PWR	Device - Ground supply pin

Tabella 4.4: MODUSB1: elenco dei pin più significativi.

4.2.3.2 Modifiche software

Dal lato PC le modifiche da apportare all'applicativo CAN*software* sono altrettano elementari se non addirittura più immediate. FTDI ha realizzato per rendere ancora più semplice lo sviluppo di soluzioni software ed hardware compatibili con il proprio integrato, un driver denominato "Virtual COM Port driver" o più semplicemente VCP disponibile per tutte le piattaforme Windows attuali, che consente di trattare la porta USB del pc come una semplice porta COM aggiuntiva[23]. Una volta installato il VCP driver CAN*software* sarà in grado di accedere all'interfaccia CAN2USB*interface* tramite la porta USB del personal nella stessa maniera in cui accede alla porta COM chiamando le "Windows VCOMM API"; è compito del "VCP driver" trasdurre le istruzioni inviate alla porta COM virtuale in chiamate a funzione in grado di gestire in modo equivalente la porta USB.

L'unica modifica da apportare al software consiste nel selezionare la porta COM creata dal VCP driver ed impostare i nuovi parametri di inizializzazione ed in modo particolare il nuovo baudrate (al massimo 1Mbit/s). L'uso della porta virtuale è una soluzione che permette di mantenere il nuovo hardware con interfaccia USB perfettamente compatibile con il software attualmente sviluppato.

[23]FTDI fornisce in alternativa al driver VCP la possibilità di accedere all'interfaccia USB direttamente tramite librerie dedicate (DLL) per Windows chiamate "D2XX Direct Drivers" anch'esse fornite gratuitamente da FTDI e scaricabili dal suo sito internet.

Bibliografia

[1] TEMIC Semiconductors. *Hardware Description of the C51 Family Products*, 1997.

[2] Fairchild Semiconductor. *MM74HC14 Hex Inverting Schmitt Trigger datasheet*, 1999. www.fairchildsemi.com.

[3] Philips Semiconductors. *Application Note PCA82C250/251 CAN Transceiver (AN96116)*. www.semiconductors.philips.com.

[4] ATMEL Corporation. *Frequently Asked Questions (FAQ) on T89C51CC01 CAN microcontrollers*, 2000. www.atmel.com.

[5] ATMEL Cooporation. *In-System Programming - T89C51CC01 UART bootloader description*, 2002. www.atmel.com.

[6] MICROSEMI. *Transient voltage suppressor*, 2003. www.microsemi.com.

[7] ATMEL Corporation. *Atmel T89C51CC01 Datasheet*, 2002. www.atmel.com.

[8] Philips Semiconductors. *PCA82C250 CAN controller interface datasheet*, 1997. www.semiconductors.philips.com.

[9] Maxim Integrated Products. *Maxim +5v-Powered, Multichannel RS-232 Drivers/Receivers datasheet*, 2003. www.maxim-ic.com.

[10] STMicroelectronics. *L8000 series Positive Voltage Regolator datasheet*, 1999. www.stm.com.

[11] Future Technology Devices Intl. Ltd. *FT232BM USB UART (USB-Serial) I.C. datasheet*, 2003. www.ftdi.com.

[12] Ravar Pty Ltd. *USB MOD1 User's manual*. www.ravar.net.

Capitolo 5

Il software di CAN*analyser*

In questo capitolo si vuole mostrare in quale maniera è stato implementato il software di CAN*analyser*; nella prima parte si descrive il codice con cui è stato programmato il micro, si discute in quale maniera si è arrivati a fare certe scelte e le prestazioni che si sono ottenute; stesso tipo di analisi viene condotta su CAN*software* nella seconda parte del capitolo.

5.1 MICRO*software*

All'applicativo che viene fatto girare sul microcontrollore ATMEL T89C51CC01 è stato dato il nome di MICRO*software* e ad esso ci riferiamo in questa prima parte del capitolo.

CAN*analyser* è un analizzatore di traffico dove l'interfaccia CAN*interface* si occupa del collegamento fisico dello strumento al bus e della decodifica delle informazioni che transitano su di esso (il traffico della rete); il ruolo di MICRO*software* è quello di stabilire un canale di comunicazione stabile con il Personal Computer dell'operatore, formattare adeguatamente i dati raccolti dall'interfaccia, integrarli di altre informazioni necessarie per la gestione di particolari funzionalità ed inviarli al PC perché CAN*software* possa visualizzarli e maniporarli.

In conclusione MICRO*software* si occupa di gestire la comunicazione tra l'interfaccia hardware ed il PC.

Tra MICRO*software* e CAN*software* c'è un'intenso e continuo scambio di informazioni è dunque di primaria importanza descrivere in quale maniera avviene l'handshake tra questi due elementi dell'analizzatore. In figura 5.1 si sono schematizzate le principali operazioni svolte dalle due applicazioni

ed in corrispondenza di alcune di esse si è riportata la comunicazione basata su l'invio di comandi e dati; prendendo la figura come riferimento il microcontrollore esegue sequenzialmente le seguenti operazioni:

Inizializzazione COM vengono inizializzati: i parametri generici di configurazione del μC, le strutture dati utilizzate da MICRO*software* ed i parametri di gestione dell'interfaccia seriale; a questo punto CAN*interface* è in grado di ricevere ed inviare comandi.

Verifica connessione MICRO*software* rimane in attesa del comando "cmdActiveMonitor" con cui CAN*software* fornisce il consenso ad iniziare una nuova acquisizione e contestualmente controlla che l'interfaccia sia presente e correttamente connessa con il Personal Computer (attende la ricezione del comando "cmdAckToAM").

Attesa parametri MICRO*software* si aspetta di ricevere da CAN*software* 14 byte attraverso i quali sono stati codificati i parametri di configurazione del CAN controller (sono parametri che possono essere modificati dall'operatore ad ogni connessione). Nel caso in cui venga fatta richiesta di "autobaud" i parametri che MICRO*software* calcola autoconfigurando il CAN controller vengono trasmessi a CAN*software*.

Acquisizione dati Una volta configurato anche il CAN controller l'interfaccia è in grado di decodificare e memorizzare nelle mailbox libere a disposizione i frame che transitano sulla rete; acquisizione, formattazione dei frame ed invio dei messaggi al PC si arrestano solo al ricevimento del comando di disconnessione "cmdDisconnect".

Reinizializzazione Ricevuto il comando di arresto MICRO*software* reinizializza il sistema e rimane in attesa di ricevere istruzioni.

5.1.1 Struttura ed elenco dei comandi

I byte che PC e interfaccia si scambiano per comunicare impostazioni e stati del sistema sono chiamati "comandi", attraverso di essi viene implementato una sorta di handshake software per la gestione del flusso di dati in transito tra i due sottosistemi.

5.1.1.1 I comandi inviati da CAN*software*

I comandi che vengono inviati dal PC al micro servono per impostare la modalità operativa dell'analizzatore e fissare i parametri di funzionamento

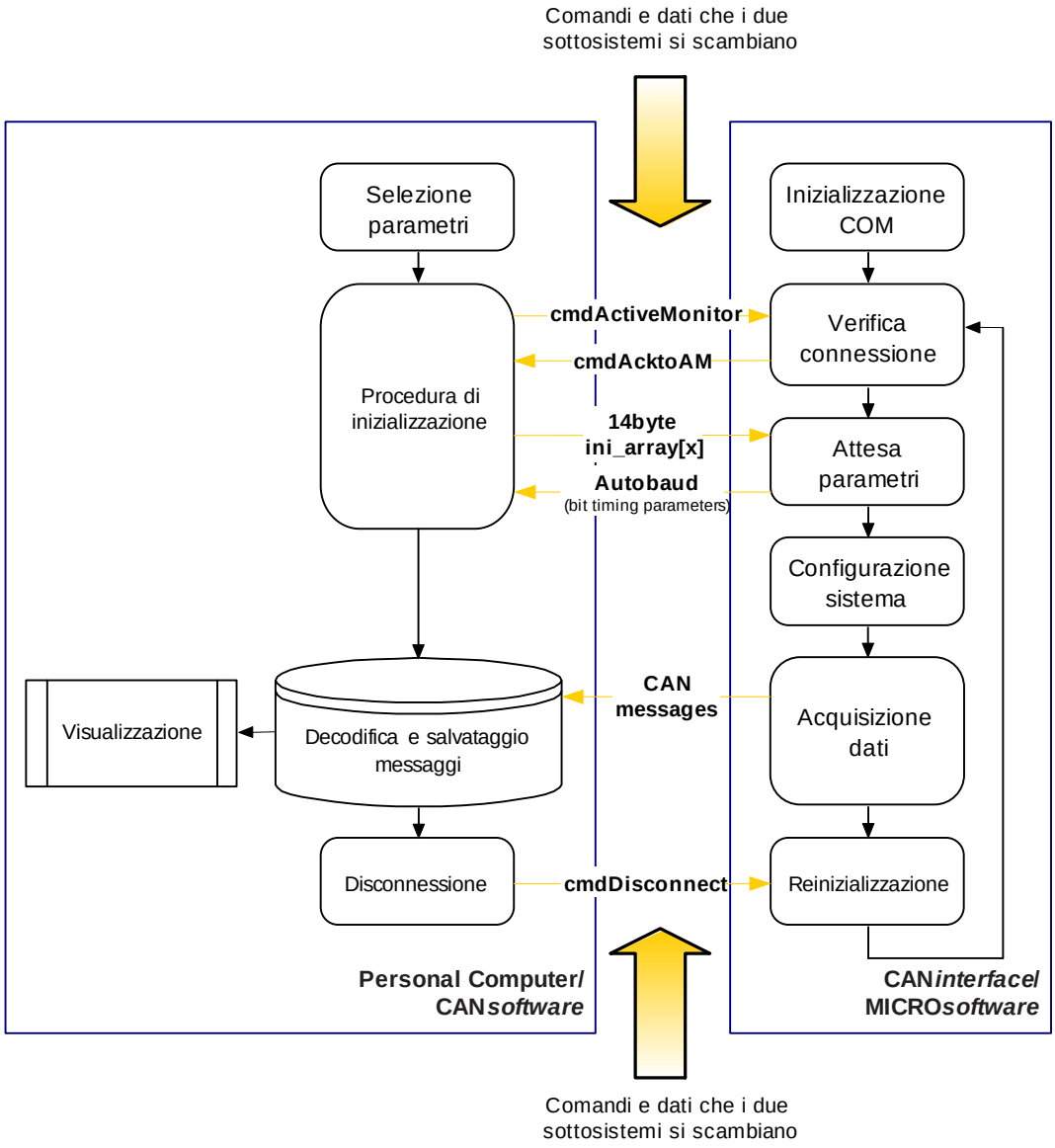


Figura 5.1: MICRO*software*: descrizione dell'handshake tra CAN*software* e MICRO*software*.

Mnemonic	Description	Bin	Hex
cmdActiveMonitor	Attiva acquisizione dati	0000 0001	01h
cmdDisconnect	Disconnessione	0000 0010	02h
cmdActiveSim	Attiva simulazione nodo virtuale	0000 0011	03h
ini_array[0]	CANIDM1 (ID Mask register 1)		
ini_array[1]	CANIDM2 (ID Mask register 2)		
ini_array[2]	CANIDM3 (ID Mask register 3)		
ini_array[3]	CANIDM4 (ID Mask register 4)		
ini_array[4]	CANIDT1 (ID Tag register 1)		
ini_array[5]	CANIDT2 (ID Tag register 2)		
ini_array[6]	CANIDT3 (ID Tag register 3)		
ini_array[7]	CANIDT4 (ID Tag register 4)		
ini_array[8]	*vedi tabella 5.2*	000x xxxx	
ini_array[9]	Bit timing Prescaler BRP	00xx xxxx	00-3Fh
ini_array[10]	Bit timing Re-sync. SJW	0000 00xx	00-03h
ini_array[11]	Bit timing Prop. time PRS	0000 0xxx	00-07h
ini_array[12]	Bit timing Phase seg.1 PHS1	0000 0xxx	00-07h
ini_array[13]	Bit timing Phase seg.2 PHS2	0000 0xxx	00-07h

Tabella 5.1: CAN*software*: comandi inviati.

del CAN controller; per semplificare l'analisi sono stati suddivisi in quattro gruppi e riportati in tabella 5.1.

I primi 3 comandi (I° gruppo) impostano la modalità operativa di CAN*interface*. L'operatore per mezzo di CAN*software* può decidere di monitorare i dati sulla linea, utilizzare il micro come un nodo virtuale o disconnettere il PC dall'interfaccia per compiere un'analisi off-line dei dati precedentemente acquisiti.

Il secondo gruppo è costituito dai byte che MICRO*software* utilizza per configurare i 4 registri CANIDM (CAN IDentifier Mask register) ed i 4 CANIDT (CAN IDentifier Tag register);CAN*software* formatta questi dati in modo tale che MICRO*software* debba preoccuparsi solo ed unicamente di caricare i byte ricevuti nei registri corrispondenti senza eseguire ulteriori elaborazioni.

- I registri CANIDT, per ogni singolo canale, specificano l'identificatore associato al messaggio CAN (da inviare se il canale è di trasmissione o da usare per filtrare i messaggi ricevuti se il canale è programmato in ricezione). CAN*software* nella generazione dei byte da inviare tiene conto che: CANIDT3 è significativo solo se il canale è programmato

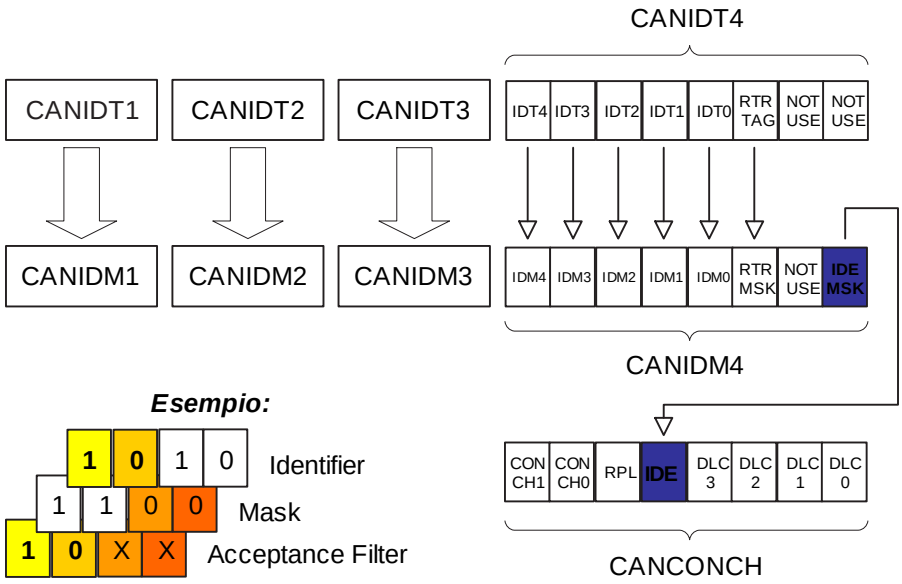


Figura 5.2: ATMEL T89C51CC01: meccanismo di filtraggio degli identificatori o "Acceptance Filtering".

per gestire identificatori estesi[1], il bit RTRTAG (Remote Transmission Request Tag value) di CANIDT4 specifica se l'identificatore deve essere associato ad un canale destinato a contenere/ricevere un DATA Frame o a un REMOTE Frame.

- I 4 registri CANIDM definiscono una maschera che il CAN controller utilizza durante l'operazione di "acceptance filter" per selezionare i bit dei corrispondenti 4 registri CANIDT su cui effettuare la verifica in fase di ricezione (figura 5.2). Oltre al bit RTRMSK associato all'RTRTAG, il registro CANIDM4 include un bit IDEMSK (IDentifier Extension Mask value) che abilita il controllo sul tipo di identificatore associato al messaggio (standard o esteso) e specificato nel bit IDE (IDentifier Extension) del registro CANCONCH (CAN message object CONtrol and DLC register).

Con gli ultimi cinque comandi (IV° gruppo) CAN*software* fornisce al microcontrollore i dati da utilizzare per configurare i registri che contengono

[1]Gli identificatori estesi previsti dalla versione 2.0B del protocollo CAN sono composti da 29 bit, il T89C51CC01 appartiene alla categoria dei "moduli CAN attivi": il CAN controller integrato gestisce indifferentemente il protocollo standard (2.0A) e quello esteso (2.0B).

Bit Position	Mnemonic	Description
0	**cmdSMPEnable**	Se 1 abilita la funzione di triplice campionamento del bus all'interno dello stesso bit time (funzione "SMP").
1	**cmdMessageEnable**	Se 1 abilita MICRO*software* all'invio del contenuto del campo DATA dei messaggi ricevuti.
2	**cmdTimeEnable**	Se 1 abilita l'invio delle informazioni di stamping per la gestione della funzione di "time stamp".
3	**cmdAutobaud**	Se 1 abilita la funzione di "autobaud".
4	**cmdIdeTag**	Imposta il valore del campo IDE di CANCONCH: 0 - "standard" 2.0A; 1 - "extended" 2.0B
5		**Not used**
6		
7		

Tabella 5.2: CAN*analyser*: dettaglio della struttura del comando "iniarray[8]".

i parametri di bit-timing del CAN controller dettagliatamente descritti in appendice C:

- CANBT1: "CAN Bit Timing register 1";
- CANBT2: "CAN Bit Timing register 2";
- CANBT3: "CAN Bit Timing register 3".

I dati che compongono il vettore "ini_array[x]" sono i 14 byte della "sequenza di inizializzazione" per mezzo della quale CAN*software* invia i parametri da utilizzare per configurare il CAN controller (ID e maschera per il filtro dei messaggi ed i parametri di bit-timing); di questi 14 byte ci rimane solo da esaminare il nono, l'"ini_array[8]" (il III° gruppo).

"ini_array[8]" ha una struttura composta da 5 campi organizzati come descritto in tabella 5.2. Per mezzo dei bit "cmdAutobaud", "cmdSMPEnable" e "cmdIdeTag" si completa la configurazione del CAN controller; rispettivamente viene abilitata/disabilitata la procedura di autoconfigurazione dei parametri di bit-timing, del campionamento ripetitivo del bus e il tipo di messaggi da filtrare. Per mezzo dei campi rimanenti si attivano/disattivano

Bit Position	Mnemonic	Value	Description
0		1	**Reserved**
1	**cmdAckToDISflag**	x	Comando di ACK alla richiesta di disconnessione.
2	**cmdAckToAMflag**	x	Comando di ACK alla richiesta di inizio acquisizione dati (conferma anche la corretta connessione dell'interfaccia alla porta PC).
3		1	**Reserved**
4	**cmdAckToSIMflag**	x	Comando di ACK alla richiesta di inizio simulazione (non implementata).
5	**cmdOvrRunFlag**	x	Notifica dell'overflow del registro CANSTMP per la gestione della funzione di "time stamp".
6	**cmdOvrBufFlag**	x	Notifica della condizione di overbuffer ed utilizzato per la gestione della funzione di "time stamp".
7			**Not Used**

Tabella 5.3: MICRO*software*: dettaglio della struttura dei comandi inviati.

alcune funzionalità e cioè quella di "time stamp" che consente di produrre un riferimento *assoluto* di tempo per la rete CAN e l'invio dei dati contenuti nei DATA Frame acquisiti.

5.1.1.2 I comandi inviati da MICRO*software*

I comandi che CAN*software* riceve hanno tutti a comune la struttura riportata in tabella 5.3; tutti i comandi descritti hanno a comune i bit 0 e 3 (posti ad 1), in questo modo non è possibile per CAN*analyser* confondere un "comando" con i "CAN message" cioè con i byte di dato (la struttura di questi "CAN message" è descritta nella sezione 5.1.2.2).

Con i primi tre comandi di tabella 5.3 MICRO*software* fornisce l'ack, la risposta alle richieste di *acquisizione dati*, *simulazione* o di *disconnessione dal bus*; inviata la richiesta di attivazione della modalità operativa CAN*analyser* rimane in attesa di ricevere questi ack per verificare che l'interfaccia sia propriamente connessa al PC ("cmdAckToAM" e "cmdAckToSIM") e per avere conferma del fatto che CAN*interface* ha trasferito tutti i messaggi che ave-

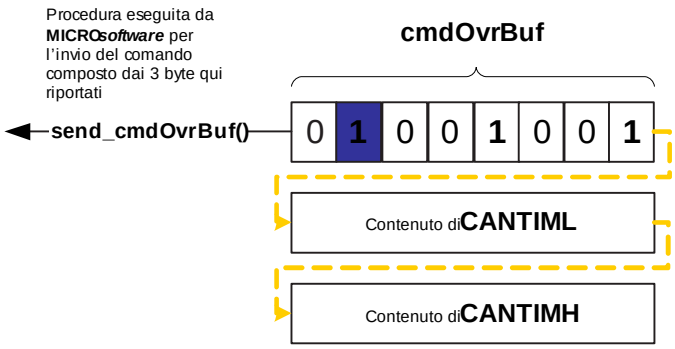


Figura 5.3: MICRO*software*: dettaglio della struttura del comando di overbuffer.

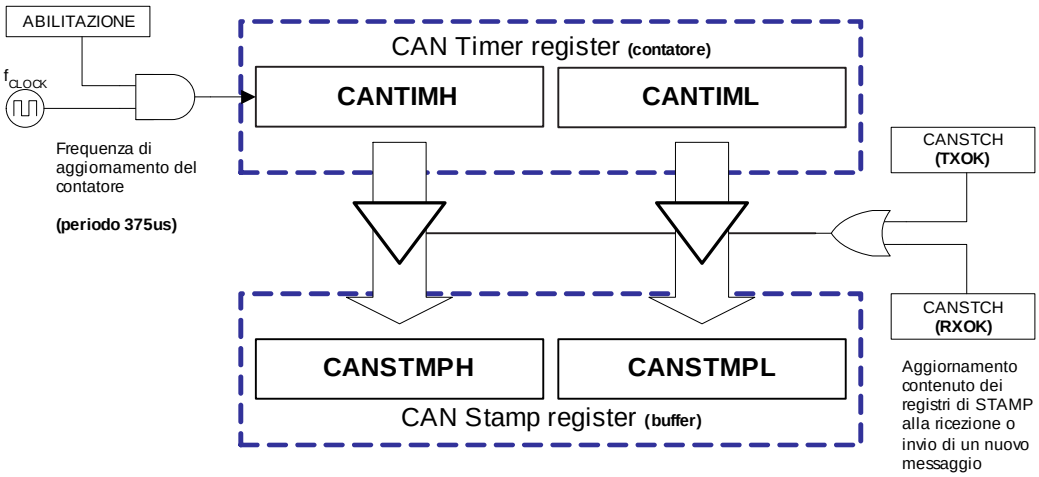


Figura 5.4: ATMEL T89C51CC01: diagramma semplificato a blocchi dell'utilizzo dei registri CANTIM e CANSTMP.

va acquisito e mantenuto in memoria prima della ricezione del comando di disconnessione (cmdAckToDIS).

"cmdOvrRunFlag" viene usato per comunicare l'overflow del registro di time stamp mentre il bit "cmdOvrBufFlag" viene impiegato per riconoscere il trasferimento dei due byte che seguono l'invio di questo comando e che contengono il valore attuale dei registri utilizzati per lo stamping dei messaggi, CANTIML (CAN TIMer Low register) e CANTIMH (CAN TIMer High register). In figura 5.3 si è riportata la struttura di questo comando mentre in figura 5.4 si vuole mostrare in che maniera i registri di stamp vengono utilizzati nel microcontrollore T89C51CC01 ed evitare che in futuro si possa fare confusione tra l'utilizzo dei registri CANSTMP (CAN STaMP timer register) e CANTIM.

Il protocollo RS232 definisce due tecniche per negoziare l'inizio e la fine della trasmissione: via software per mezzo dei caratteri "XON" e "XOFF" oppure via hardware per mezzo di segnali e linee dedicate. Il PC e l'intefaccia CAN*interface* sfruttano l'RS232 come protocollo di codifica dei byte ma implementano per mezzo dei comandi descritti una personale tecnica di controllo del flusso di comunicazione via software.

5.1.2 Dati inviati al PC durante l'acquisizione: "CAN messages"

L'acquisizione dei dati da parte dell'analizzatore richiede che MICRO*software* abiliti la funzionalità di "Listening Mode" del microcontrollore[2] e che tutti i 15 canali disponibili vengano programmati in modalità "Buffer Receive", così facendo si riduce al minimo il rischio di perdere messaggi che transitano sul bus per mancanza di mailbox libere[3]. L'arrivo di un nuovo messaggio e quindi il caricamento di un nuovo canale viene rilevato ad interruzione di programma (interrupt 7), la procedura "can_drv_it_rxok()" richiamata dall'interrupt aggiorna la struttura dati "CodaRX" con l'indice della mailbox che viene occupata dal frame acquisito. Appena il sistema è disponibile e cioè non devono essere gestite nuove richieste di interruzione e non devono essere inviati i comandi con cui viene gestito l'overflow del "time stamp" e notificato l'overbuffer delle mailbox, MICRO*software* si occupa solo di inviare il contenuto delle mailbox ad CAN*software* prelevando l'indice del canale contenente il messaggio da inviare da "CodaRX". La sequenza di dati che vengono inviati durante l'acquisizione comandi esclusi, sono chiamati "CAN message" (vedi figura 5.1).

5.1.2.1 "CodaRX": struttura ed utilizzo

Si tratta di una coda circolare gestita con tecnica FIFO ed implementata per mezzo di un array di 15 elementi[4] (byte); due indici puntano rispettivamente al *primo elemento libero* dove verrà eseguito il prossimo "inserimento" ed al *primo elemento occupato* da dove verrà eseguita la prossima "estrazione".

[2]MICRO*software* deve settare il bit "AUTOBAUD" nel registro CANGCON (CAN General CONtrol register).

[3]In mancanza di canali disponibili a memorizzare il nuovo messaggio ricevuto il microcontrollore viene programmato per generare un'interruzione di programma al verificarsi della quale i 15 canali occupati vengono svuotati inviando il loro contenuto a CAN*software* e successivamente il μC viene riabilitato in acquisizione.

[4]La dimensione rispetta il numero massimo di mailbox disponibili nel T89C51CC01 e configurate in modalità "Buffer Receive".

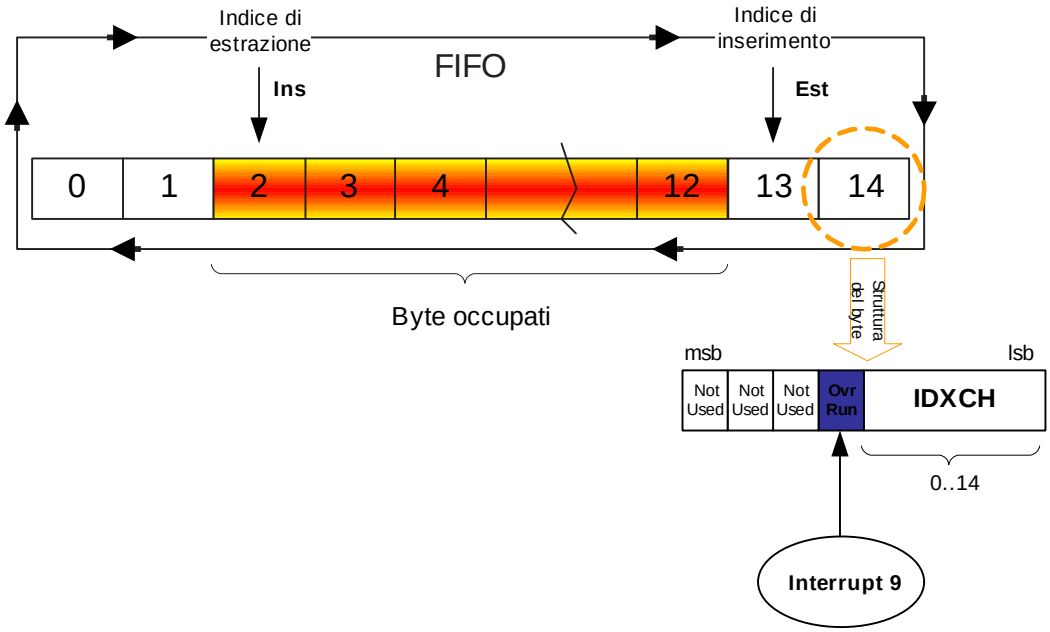


Figura 5.5: MICRO*software*: struttura ed organizzazione di "CodaRX".

Lo scopo di questa struttura dati (figura 5.5) è principalmente quello di memorizzare l'ordine con cui vengono caricate le mailbox, a tale scopo il campo "IDXCH" (IDentifier X CHannel) puntato dall'indice "Ins" viene caricato con l'identificativo del canale che ha ricevuto l'ultimo frame; per mezzo di questo campo MICRO*software* è in grado di inviare i "CAN message"a CAN*software* esattamente nell'ordine in cui i frame sono transitati sul bus.

- La routine di gestione dell'interruzione "can_drv_it_rxok()" recupera l'indice del canale per mezzo della funzione "FindFirstChIt()", lo carica nel campo "IDXCH" del byte puntato da "Ins", incrementa l'indice della coda e termina fornendo l'ack all'interrupt[5].

- La funzione "FindFirstChIt" restituisce l'indice del canale a più alta priorità che ha generato un'interruzione di tipo "7" e questa condizione viene riconosciuta scansionanto il contento dei registri "CANSIT[6]": la

[5]Con l'ack viene resettata via software la condizione che aveva generato l'interruzione, una delle conseguenze è che viene "azzerato" il bit associato al corrispondente canale nei registri "CANSIT".

[6]Assieme al verificarsi di altri eventi che però MICRO*software* non abilita, il caricamento in un canale libero di un nuovo frame ha l'effetto di settare il bit corrispondente al canale occupato nei registri "CANSIT 1/2" (CAN Status InTerrupt register 1 e 2) e come successiva conseguenza la generazione di una interruzione di programma, la numero 7 che

procedura termina al primo bit che trova settato (partendo da quello meno significativo). Il problema nell'utilizzare questo metodo è che si corre il rischio che al momento in cui viene invocata questa procedura esistano più bit ad 1 nei registri "CANSIT"; se "FindFirstChIt" ritorna l'indice del primo canale che trova aver generato un'interruzione (quello a più alta priorità) non è detto che chiamate successive forniscano una sequenza di indici cronologicamente sequenziali. Il caso peggiore è quello in cui i messaggi sul bus CAN si susseguono uno dopo l'altro (bus impegnato al 100%) ed abbiano durata minima (47μs corrispondente a 47 bit[7] ad un bitrate di 1 Mbit/s). Il problema viene risolto programmando l'interrupt alla massima priorità (si impostano ad 1 i bit "IPH" ed "IPL" associati all'interruzione) ed ottimizzando il codice di gestione dell'interruzione stessa in questo modo la routine di servizio "can_drv_it_rxok()" non ha mai una durata superiore a 45μs[8] e quindi l'indice restituito è sempre quello corretto.

Il campo indicato con "OvrRun" viene sfruttato per memorizzare l'eventuale condizione di overflow del registro "CANTIMH+CANTIML"; quest'informazione verrà inviata a CAN*software* durante la trasmissione del "CAN message" associato all'indice contenuto in "IDXCH". Anche la condizione di overrun come quella di ricezione di un nuovo messaggio viene rilevata ad interruzione di programma per mezzo dell'interrupt 9.

5.1.2.2 Struttura dei "CAN message"

Il modo nel quale i dati relativi ai frame acquisiti vengono formattati ed organizzati per essere inviati al PC è mostrato in figura 5.6. Alcuni dei byte indicati sono presenti solo in certe condizioni:

- "CANIDT3" solo quando il messaggio da inviare ha identificatore esteso;
- i byte di dato "CANMSG" del frame vengono inviati solo se l'operatore ha attivato la relativa funzionalità (in tale caso MICRO*software* deve aver ricevuto il comando "cmdMessageEnable" descritto in 5.1.1.1) e solo se si tratta di un DATA Frame. In queste ipotesi il numero di byte

viene intercettata da "can_drv_it_rxok()".

[7] Data frame standard (11bit di identificatore), senza campo dati e interframe minimo (3bit).

[8] "FindFirstChIt()" che è richiamata dalla routine di servizio, ha una durata che dipende dal numero di canali da scansionare, il caso peggiore è quello in cui il frame carica la 14-ma mailbox e quindi la funzione è costretta a scansionare tutti i bit di "CANSIT1" e "CANSIT2" prima di restituire un valore.

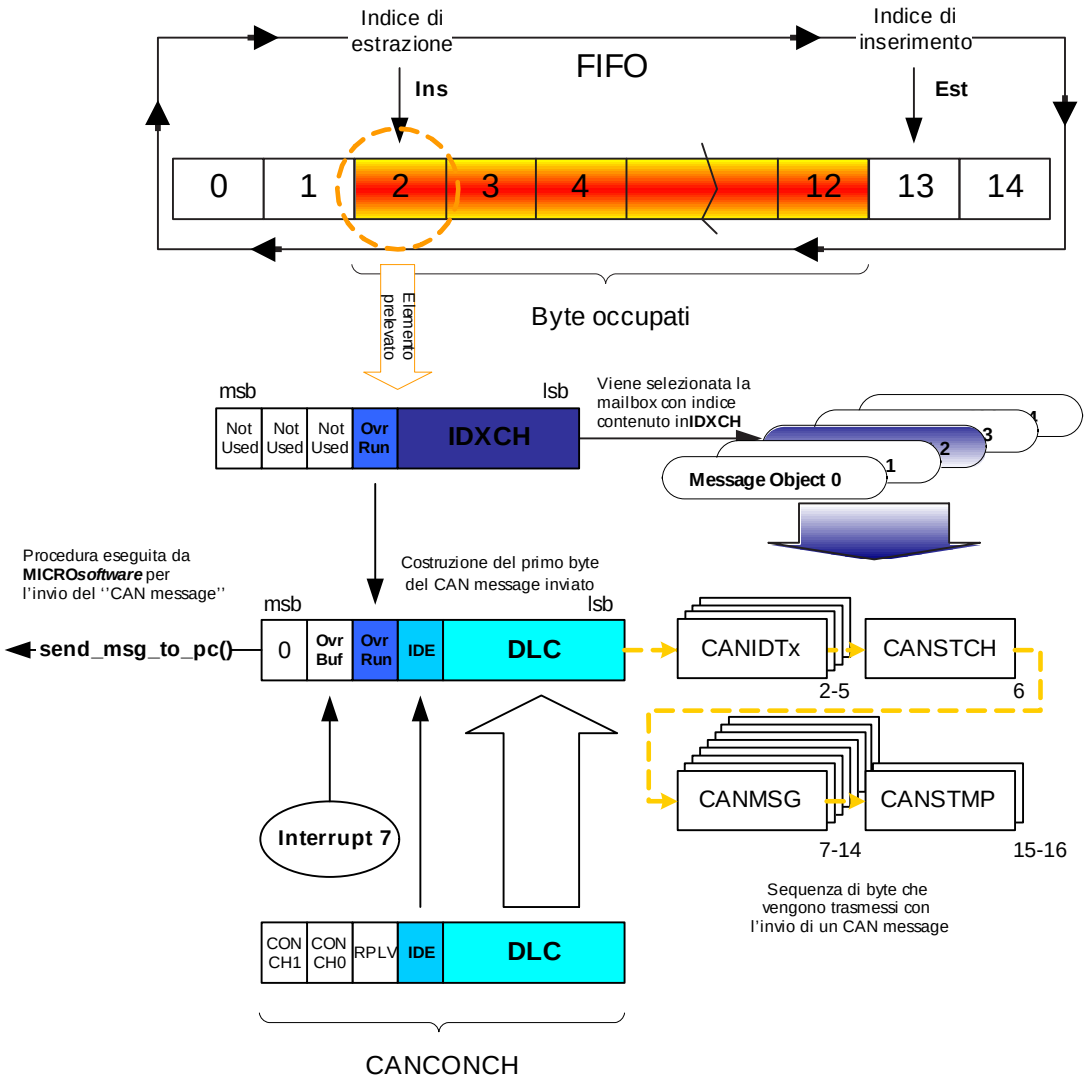


Figura 5.6: MICRO*software*: formattazione ed organizzazione dei "CAN message".

di dati che compongono il "CAN message" sono pari al contenuto del campo "DLC" del registro "CANCONCH";

- "CANSTMPL" e "CANSTMPH" contengono il valore che avevano rispettivamente i registri "CANTIML" e "CANTIMH" (figura 5.4) al momento in cui il frame è stato caricato nella mailbox e vengono inviati solo se l'operatore ha attivato la relativa funzionalità di "time stamp" (in tale caso MICRO*software* deve aver ricevuto il comando "cmdTimeEnable" descritto in 5.1.1.1).

Oltre ai dati caratterizzanti il frame e cioè: identificatore (CANIDT), il campo dati (CANMSG), il "time stamp" (CANSTMP) e le informazioni di stato contenute nel registro "CANSTCH[9]" (CAN message object STatus CHannel register), il "CAN message" che viene trasferito include anche altre informazioni necessarie affinché CAN*software* interpreti correttamente i byte che gli arrivano. Il primo byte del "CAN message" infatti, include un campo "DLC" che contiene il numero di byte di tipo "CANMSG" trasferiti, un bit "IDE" che specifica il tipo di identificatore e quindi se viene o meno inviato il registro "CANIDT3" ed altri due bit, "OVRRUN" e "OVRBUF" il cui ruolo sarà descritto nei paragrafi successivi.

> Attraverso il primo byte che viene inviato CAN*software* ha tutte le informazioni necessarie per interpretare i byte che seguiranno.

In conclusione ogni messaggio "CAN message" può essere composto da un numero di byte compreso tra un valore minimo di 5 (identificatore standard, frame di tipo REMOTE e funzione di "time stamp" non attiva) ad un massimo di 16 (identificatore esteso, frame di tipo DATA con 8 byte di informazione, funzione di "time stamp" attiva).

> Riorganizzando la struttura dei dati da trasferire in base al tipo di messaggio l'interfaccia intercetta sul bus si è cercato di sfruttare al meglio la limitata banda passante del canale di comunicazione a nostra disposizione.

5.1.2.3 Gestione dell'overflow

Nonostante tutti i canali vengano configurati in ricezione la limitata banda passante della porta RS232 del PC[10] (baudrate massimo di 115.2Kbit/s)

[9] Gli ultimi 5 bit di questo registro contengono lo stato di errore rilevato sul frame acquisito e cioè: "Bit Error" (BERR bit), "Stuff Error" (SERR), "CRC Error" (CERR), "Form Error" (FERR) e "Ack Error" (AERR).

[10] L'UART16550 è il circuito di gestione della porta seriale utilizzato nella quasi totalità dei Personal Computer oggi disponibili in commercio. La versione 16550 in grado di

Descr.	**Msg. byte**	**Serial**	**115.2Kbit/s**	**Frame**	**1Mbit/s**	**BusLoad**
ID std. no DATA no STAMP	5	50bit	425μs	47bit	47μs	11%
ID std. no DATA si STAMP	7	70bit	608μs	47bit	47μs	7,7%
ID std. 8By DATA si STAMP	15	150bit	1.302ms	111bit	111μs	8,5%
ID ext. no DATA no STAMP	6	60bit	521μs	67bit	67μs	12,8%
ID ext. no DATA si STAMP	8	80bit	695μs	67bit	77μs	11%
ID ext. 8By DATA si STAMP	16	160bit	1.389ms	131bit	131μs	9,4%

Tabella 5.4: MICRO*software*: stima dei valori massimi di "bus load" che garantiscono l'acquisizione di dati in real-time senza perdita di informazioni.

non consente l'acquisizione real-time dei dati senza perdita di informazioni. In tabella 5.4 sono riportate stime dei valori massimi di "bus load" che il sistema è in grado di gestire, tra l'altro quelle riportate in tabella sono stime migliorative di quelle reali calcolate supponendo che il tempo richiesto per l'invio dei un messaggio completo al PC sia pari al tempo richiesto "dalla sola interfaccia seriale" per trasmettere i byte di cui il "CAN message" è composto[11].

> Il superamento del valore di "bus-load" riportato in tabella rende non più nulla la probabilità che ad un certo punto dell'acquisizione tutti i canali risultino occupati e quindi di alcuni frame in transito sul bus non verranno rilevati.

Al verificarsi dell'*overflow* (tutte le mailbox sono occupate nell'istante in cui viene ricevuto un nuovo frame) il sistema interrompe l'acquisizione dei

lavorare ad un bitrate di 1.5Mbit/s con errori minimi solo se pilotata a frequenza di 24Mhz. Per motivi di compatibilità software la frequenza con cui viene gestita questa UART nei PC è di 1.8432Mhz che consente di sfruttare la UART fino ad un bitrate massimo di 115.2Kbit/s.

[11] I tempi reali sono più lunghi perché si deve tenere conto dei tempi di esecuzione del codice sul micro e del fatto che l'esecuzione delle routine di servizio (interrupt) possono interrompere la procedura "send_msg_to_pc()" di invio dei dati ed in conseguenza allungarne la durata.

messaggi (i canali non vengono più riabilitati) mentre si procedere ad inviare il contenuto delle 15 mailbox "piene" al PC. Questa deficienza operativa deve essere comunicata all'operatore per una corretta interpretazione dei dati, a tale scopo l'overflow viene rilevato ad interruzione di programma (come la ricezione anch'esso genera un interrupt 7) e come è riportato anche in figura 5.6 il bit "OvrBuf" , in questo caso del 15-mo messaggio (l'ultimo) che il micro invia a CAN*software*, viene settato.

5.1.2.4 Gestione del "time stamp"

Il T89C51CC01 dispone di un timer a 16bit realizzato dai due registri "CANTIMH" e "CANTIML" per la gestione dello stamp dei messaggi (figura 5.4).

- Il contenuto di questi due registri viene caricato rispettivamente in "CANSTMPH" e "CANSTMPL" al ricevimento o all'invio di un frame.
- In ricezione la cattura avviene quando il flag "RxOK" di "CANSTCH" viene settato.
- In trasmissione quando il flag "TxOK" di "CANSTCH" viene settato.
- Con un clock a 16Mhz come quello impiegato nella CAN*interface* la condizione di overrun[12] si verifica circa ogni 24.58ms ed il micro, se configurato opportunamente, genera un interrupt 9.

$$t_{OVERRUN} = 0.375\mu\text{s x } 65536 = 24.58 \quad [ms]$$

La finestra temporale gestita dal timer è piuttosto limitata, associare ad ogni singolo messaggio un riferimento temporale di transito sul bus richiede qualche cosa in più rispetto a tenere sotto controllo solo i registri CANSTMP; dopo 24.58ms di acquisizione cosa si fà?

CAN*software* deve essere informato dell'overrun e questa comunicazione deve essere sincronizzata con l'invio dei messaggi in modo tale che i valori di time stamp trasmessi vengano pesati in modo corretto come si vede in figura 5.7 (si deve rispettare la sequenza cronologica degli eventi). MICRO*software* implementa questa funzionalità per mezzo dell'interrupt 9 che richiama la routine di servizio "can_drv_it_overrun" riportata nel listato 5.1 a pagina 146. Questa procedura:

[12] Il termine "overrun" viene utilizzato per indicare la condizione di overflow del registro timer.

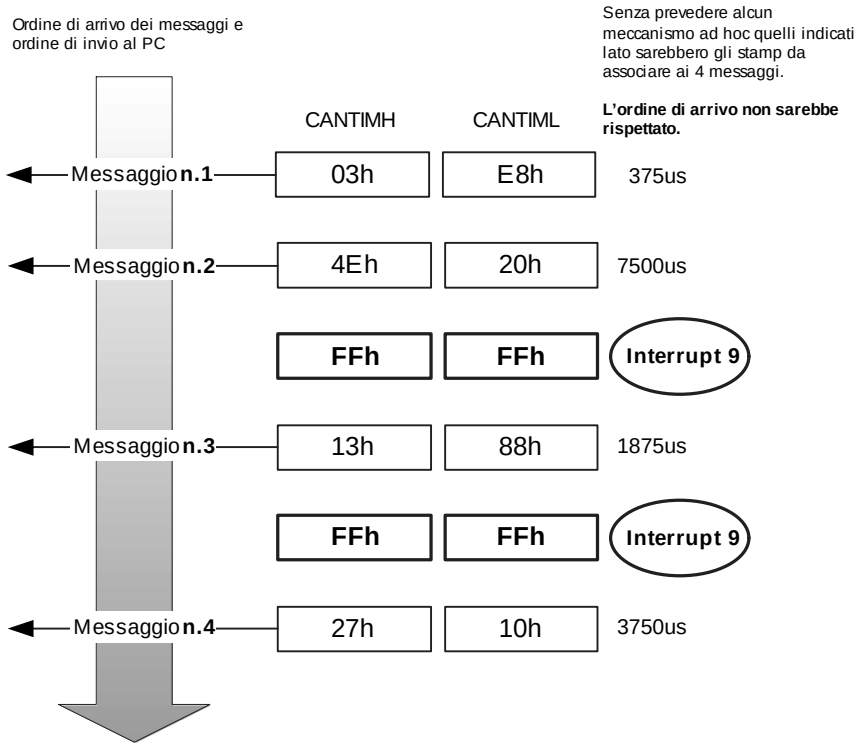


Figura 5.7: MICRO*software*: sincronizzazione dei "CAN messagge" l'invio del comando di overrun.

- setta il bit "OvrRun" dell'ultimo elemento di "CodaRX" (Quello puntato dall'indice di inserimento "Ins");

- se si è verificato l'overflow dei canali o se il bit "OvrRun" precedente è già impostato si predispone (la variabile globale "OvrRunFlag" viene settata) per l'invio del comando di overrun per mezzo della funzione "send_cmdOvrRun()".

Quindi la condizione di "overrun" viene notificata a CAN*software* in due maniere: durante l'invio dei "CAN message" con il primo byte che contiene il bit "OvrRun" mostrato in figura 5.6 (CAN*software* controlla questo bit e se lo trova settato incrementa opportunamente la variabile che utilizza per gestire il "time stamp" *assoluto* dei messaggi), oppure attraverso l'invio di un "comando" cioè di una sequenza di byte che CAN*software* riconosce ed utilizza per gestire di conseguenza la funzionalità. In figura 5.8 e 5.9 sono schematizzate le tre casistiche nelle quali è necessario inviare il comando di overrun attraverso la funzione "send_cmdOvrRun()" o il comando di "send_cmdOvrBuf" precedentemente descritto; di seguito diamo la spiegazione dei tre casi a cui si stà facendo riferimento.

- E' necessario inviare il comando "send_cmdOvrRun()" se l'overrun si è verificano almeno due volte mentre "CodaRX" era vuota, sono cioè passati più di 49ms dalla ricezione dell'ultimo frame.

- Se "CodaRX" è completamente occupata ed in questa condizione si verifica l'overflow del CAN timer deve essere inviato il comando di overrun "send_cmdOvrRun()" non prima però che siano stati inviati al PC tutti e 15 i canali, affinché anche in questo caso venga rispettata la sequenza cronologica degli eventi. E' altresì necessario che lo svuotamento di tutte le mailbox avvenga prima che si verifichi un nuovo overflow del CAN timer[13].

- Se si è verificato l'overflow dei canali le mailbox non vengono più riabilitate e la coda viene mano a mano svuotata. La situazione è simile alla precedente in cui però le mailbox erano tutte occupate ma non si era verificato l'overflow: se si verifica un overrun, al termine dello svuotamento dovrebbe essere inviato un "send_cmdOvrRun()" in questo caso invece viene inviato un comando il comando di overflow tramite

[13]Nel caso peggiore l'invio di 15 canali richiede un tempo stimato in circa 23.18ms ((1.5ms + 45μs) x 15) dove si sono considerati i tempi di esecuzione di 15 interrupt 7 e i tempi di invio di un messaggio composto da 16byte (1.5ms è una stima in forte eccesso). Questo tempo è inferiore a quello con cui si ripete ciclicamente l'overflow del timer

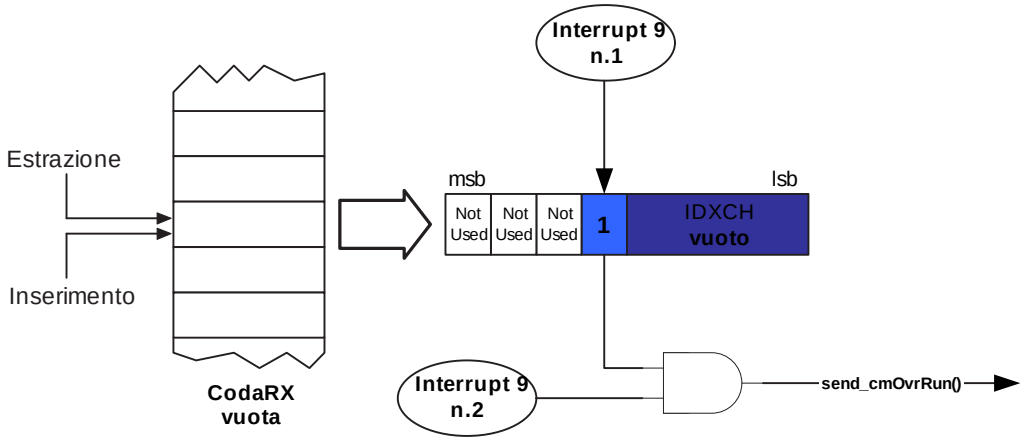


Figura 5.8: MICRO*software*: overrun in condizione di "CodaRX" vuota.

la funzione "send_cmdOvrBuf" con cui si trasmette il valore attuale dei registri di time stamp. CAN*software* con questo dato e conoscendo i tempi di riabilitazione dell'interfaccia è in grado di superare anche la condizione di overflow e fornire un riferimento di tempo assoluto.

```
1  void can_drv_it_overrun (void)
2  {
3      Uchar ovr;
4
5      if ((ElemCodaRX == DIM_CODARX) || OvrBufFlag)
6          OvrRunFlag = 1;
7      else
8      {
9          ovr = CodaRX[Ins] & 0x10;
10         if (!ovr)
11             CodaRX[Ins] |= 0x10;
12         else
13             OvrRunFlag = 1;
14     }
15
16     CANGIT &= ~MSK_CANGIT_OVRTIM;
17 }
```

Listato 5.1: can_drv_it_overrun()

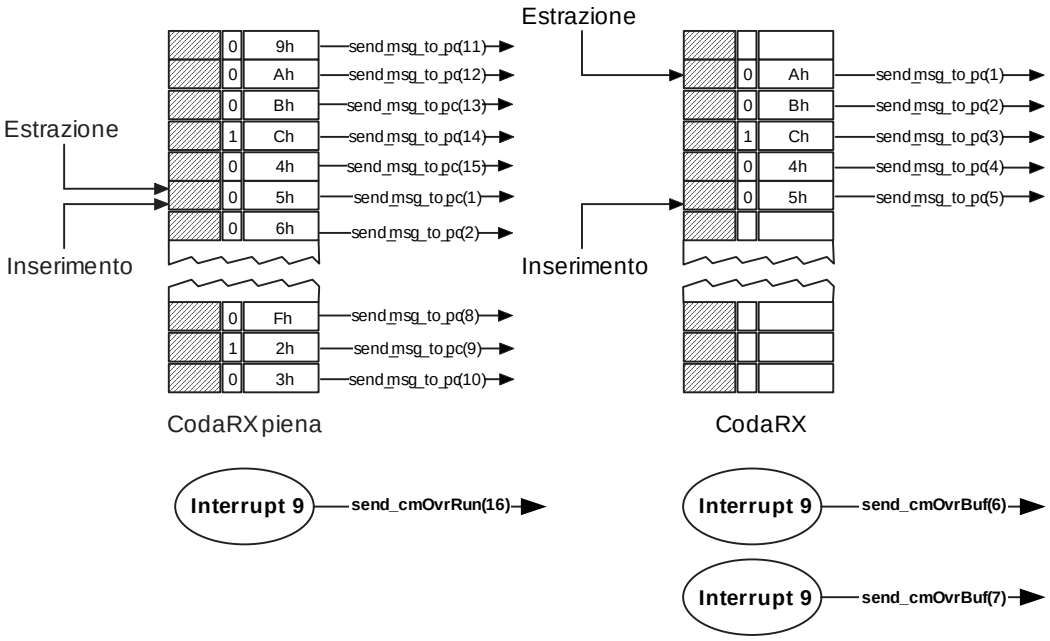


Figura 5.9: MICRO*software*: overrun in condizione di "CodaRX" piena.

5.1.3 Le altre procedure

Apparte le funzioni di invio dei comandi e quelle che implementano i servizi di interruzione che abbiamo cercato di descrivere in questa prima parte del capitolo, MICRO*software* è composto da altre interessanti procedure, per lo più procedure di inizializzazione, che di seguito descriviamo.

init_global_var() : inizializza la struttura dati "CodaRX" ed altre variabili utilizzate nel codice.

init_serial2() : imposta i parametri di configurazione per la la comunicazione via RS232.

autobaud() : ricava per approssimazioni successive una configurazione di bit timing adeguata al bitrate incognito della rete, i parametri calcolati vengono inviati a CAN*software* che in questo modo è in grado di fornire all'utente indicazioni sul bitrate stimato.

decode_init_sequence() : dai dati ricevuti da CAN*software* e da quelli eventualmente generati dalla funzione "autobaud()": inizializza le variabili locali, i parametri del CAN controller (imposta i canali in "Buffer Receive mode", i parametri di bit-timing, abilita le interruzioni) ed abilita quest'ultimo.

Queste procedure si occupano per lo più della fase di configurazione del microcontrollore ed in modo particolare del CAN controller; terminata questa prima fase inizia un ciclo interrotto solo dalla ricezione di una richiesta di disconnessione e cioè di arresto dell'acquisizione, in questa fase i messaggi prelevati dalla rete, caricati nelle relative mailbox dal CAN controller e riorganizzati opportunamente dal software vengono inviati al PC assieme ai comandi di gestione dell'overbuffer e dell'overrun.

5.1.3.1 La porta di comunicazione seriale

Lo scambio in modo seriale di informazioni tra due dispositivi richiede una qualche procedura di sincronizzazione attraverso cui il ricevitore è in grado di identificare quando un dato viene trasmesso. La comunicazione con la porta COM del PC è una comunicazione di tipo asincrono[14] dove quindi nessun segnale di clock viene inviato con i dati[15]. Ogni dato (tipicamente 8 bit) è sincronizzato usando un "bit di start" che precede la trasmissione dell'informazione (si parte sempre dal bit meno significativo), un clock interno al ricevitore scandisce gli istanti di campionamento (in base al bitrate impostato) a partire dall'istante di ricezione del bit di start. La comunicazione si chiude con l'invio dello "stop bit" e la linea si porta in stato di idle corrispondente al "mark state" della seriale o livello logico 1 in cui rimane in attesa di un un nuovo dato.

5.1.3.2 Programmazione della UART

La UART del T89C51CC01 programmata in modalità[16] 1 è configurata per trasmettere e ricevere trame di 10 bit composte da un bit di start, uno di stop e 8 bit di dato con un bitrate che può essere impostato ad un valore "arbitrario" sfruttando come "baudrate generator" il Timer 1 o il Timer 2. Quando uno dei due timer (altro non sono che contatori comandati da un segnale di clock) viene sfruttato come generatore del bitrate quest'ultimo viene determinato dalla frequenza con cui il contatore raggiunge la condizione di

[14] I termini "comunicazione sincrona" e "asincrona" creano un pò di confusione, è bene ricordare che entrambi richiedono la "sincronizzazione" tra ricevitore e trasmettitore

[15] Comunicazione sincrona: i dati sono inviati a gruppi con cadenza regolare e sono preceduti da caratteri speciali chiamati "syn" o "synchronous idle characters" che il ricevitore usa per sincronizzarsi con il trasmettitore. Non ci sono gap tra i caratteri trasmessi, la banda viene sfruttata quasi totalmente, si tratta di un meccanismo di comunicazione utile se si devono trasferire grosse quantità d'informazioni

[16] La modalità di funzionamento della porta seriale del microcontrollore viene selezionata per mezzo dei bit "SM0" e "SM1" nel registro di configurazione "SCON" (Serial CONtrol register).

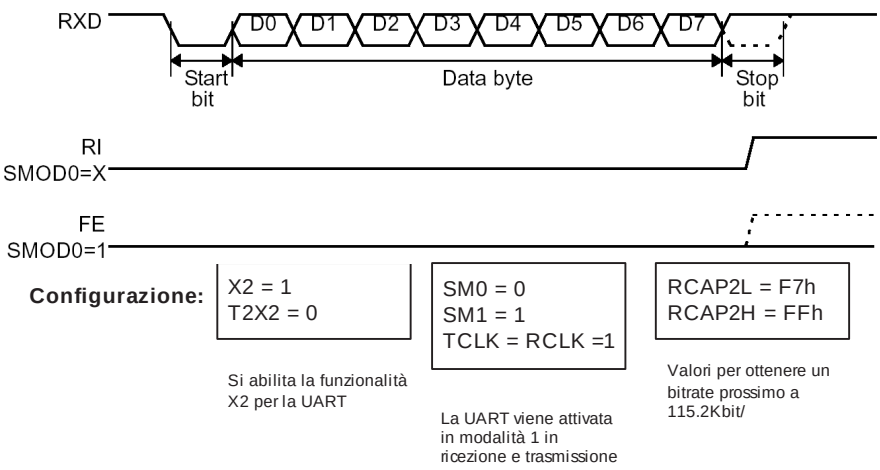


Figura 5.10: ATMEL T89C51CC01: timing dei segnali della UART configurata in modalità 1.

overflow. Per fare in modo di ottenere una frequenza costante il timer viene programmato in modalità "Auto-Reload" e cioè raggiunto l'overflow viene automaticamente ricaricato al valore preimpostato contenuto nei registri "RCAP1" (Reload/CAPture timer register 1) o "RCAP2" (Reload/CAPture timer register 2).

Il Timer 2 che è stato sfruttato per questo scopo è un contatore a 16 bit implementato attraverso due registri da 8 bit, "TH2" (Timer 2 High byte register) e "TL2" (Timer 2 Low byte register). Il Timer 1 lavora in modalità auto-reload solo come registro ad 8 bit e ciò non consente di sfruttarlo come generatore di bitrate[17] per CAN*interface* e come conseguenza viene impiegato il Timer 2; programmando i due registri "RCAP2L" e "RCAP2H" come in figura 5.10 si riesce a generare un bitrate di 111.1Kbit/s[18] che si discosta di meno del 4% dal valore richiesto di 115.2Kbit/s garantendo un errore minimo che interfaccia e PC si è verificato, sono in grado di sopportare.

La UART viene configurata per essere sfruttata in modalità full-duplex con identico baudrate in ricezione e trasmissione attraverso i bit "TCLK" e "RCLK" di "T2CON" (Timer 2 CONntrol register).

5.1.3.3 Rilevamento del CAN bit-rate in automatico

In una rete CAN ogni nodo partecipa alla gestione del traffico per effetto del fatto che i messaggi vengono inviati sulla rete in modalità "broadcast"

[17]Il valore massimo generabile è di 83.3Kbit/s.

[18]A tale scopo si sfrutta il micro in modalità X2 (6 cicli di clock per istruzione) impostando opportunamente i bit del registro "CKCON": X2 = 1, T2X2 = 0.

e ogni nodo in ricezione deve essere in grado di gestire l'acknowledge che deve essere fornito al modulo trasmittente. *La comunicazione non viene disturbata se un nodo non fornisce l'ack di risposta ma ce n'è almeno un'altro che decodifica il messaggio in modo corretto* (è questo secondo modulo che fornisce l'ack). Questo comportamento può essere utilizzato da un nodo non ancora inizializzato per rilevare automaticamente il bitrate della rete a cui viene connesso, quindi è possibile implementare funzione di "Hot-plugging"; ovviamente in questa fase il nodo non deve in alcun modo interagire con le altre stazioni della rete. La procedura di MICRO*software* che si occupa di gestire questa funzionalità è detta "autobaud()".

Il micro configurato in "Listening Mode" è in grado di provare differenti configurazioni di bit time senza che venga trasmesso un Error Frame in caso di bitrate errato. Quando viene trovata la configurazione corretta il sistema non rileva più condizioni di errore alla ricezione di un frame, in questo modo riconosce la configurazione di bit timing corretta ed il nodo è a questo punto pronto per partecipare in modo attivo alla gestione del bus. Lo schema di figura 5.11 descrive la tecnica di rilevamento del CAN baudrate impiegata in MICRO*software*.

Una volta che la procedura di "autobaud()" è terminata i parametri di bit timing così ricavati vengono formattati su 5 byte ed inviati a CAN*software* rispettando la sequenza riportata in tabella 5.5.

Mnemonic	**Description**	**Bin**	**Hex**
ini_array[9]	Bit timing Prescaler BRP	00xx xxxx	00-3Fh
ini_array[10]	Bit timing Re-sync. SJW	0000 00xx	00-03h
ini_array[11]	Bit timing Prop. time PRS	0000 0xxx	00-07h
ini_array[12]	Bit timing Phase seg.1 PHS1	0000 0xxx	00-07h
ini_array[13]	Bit timing Phase seg.2 PHS2	0000 0xxx	00-07h

Tabella 5.5: MICRO*software*: comandi di bit timing inviati a CAN*software*.

5.2 CAN*software*

CAN*software* è l'interfaccia *Microsoft Windows* © che consente all'utente di gestire l'analizzatore di protocollo. Sviluppato in Microsoft *Visual C++ v.6* © la realizzazione dell'interfaccia grafica è stata semplificata dall'impiego delle classi "MFC" (Microsoft Foundation Class library) a cui Visual C++ si appoggia senza alcuna difficoltà.

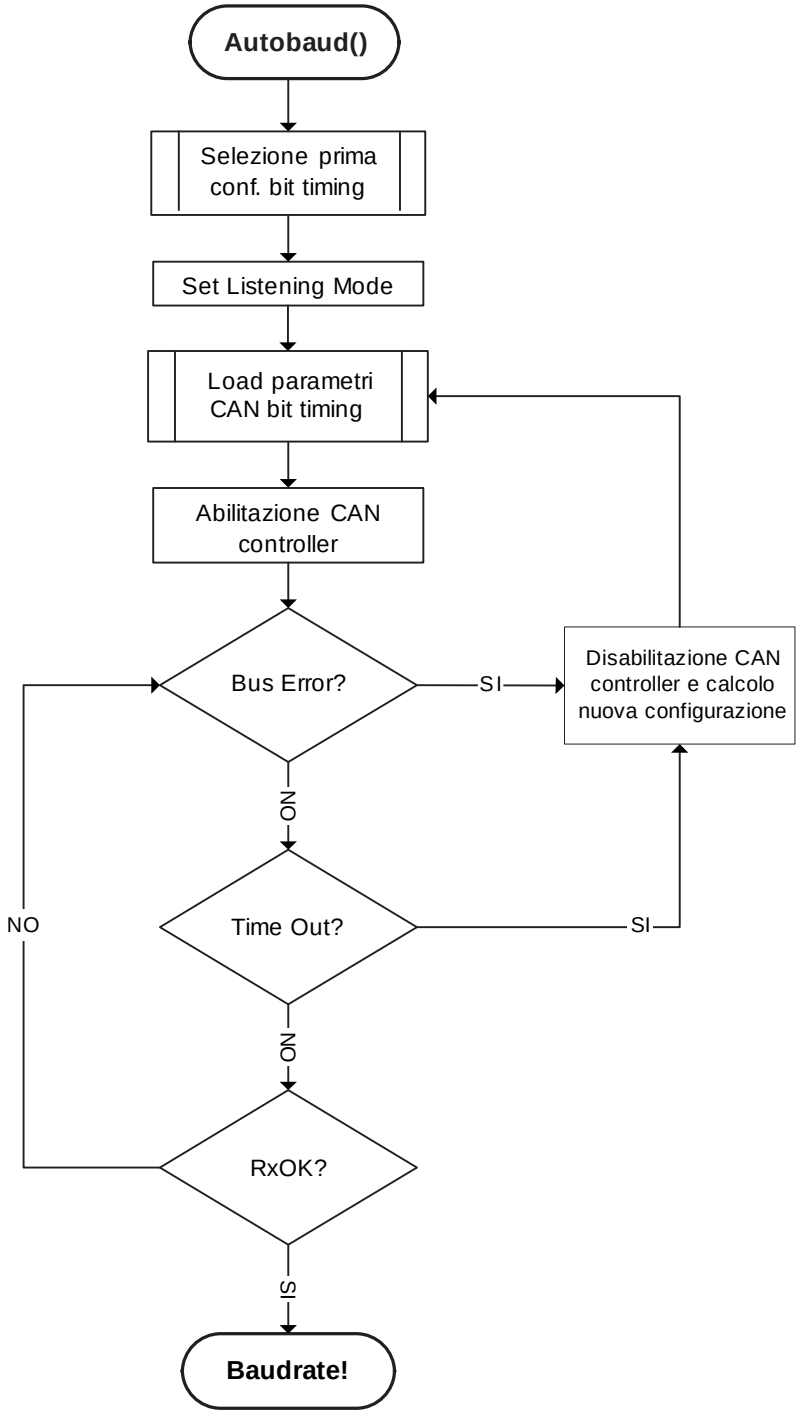


Figura 5.11: MICRO*software*: diagramma di flusso a blocchi della procedura di "autobaud()".

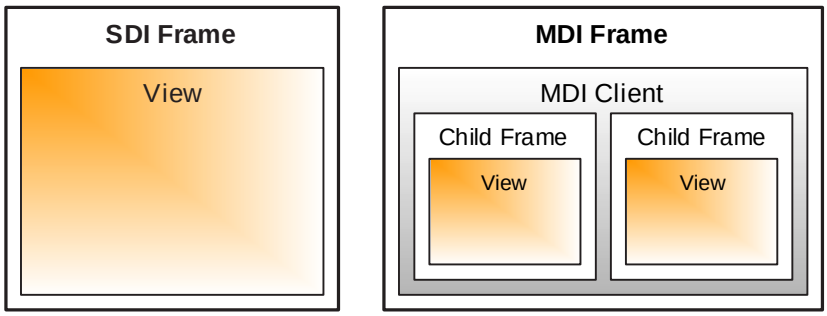


Figura 5.12: Applicazioni SDI e MDI: gerarchia delle finestre

5.2.1 L'architettura documento/vista

CAN*software* appartiene alla categoria delle applicazioni "SDI" (Single Document Interface) nelle quali all'utente viene permesso di lavorare su un solo documento alla volta, MFC offre un valido aiuto per generare applicazioni di questo genere in grado di contenere gli elementi tipici di una interfaccia Windows: un menù, una barra degli strumenti, una di stato ed altri.

Il cuore delle applicazioni SDI basate su MFC è una struttura interconnessa di classi chiamata comunemente "Architettura documento/vista" o "Document/View Framework".

Il concetto fondamentale su cui si basa questa architettura è la separazione tra i dati effettivi e la presentazione all'utente di ciò che i dati rappresentano. Questa divisione viene ottenuta conservando i dati in una classe (classe documento) e presentando le informazioni all'utente attraverso un'altra classe (classe vista). La libreria MFC mette a disposizione classi documento come "CDocument" che forniscono molte funzioni e metodi di interfaccia utente per il caricamento e la memorizzazione delle informazioni nei file; mette a disposizione più classi vista derivate dalla classe "CView" che forniscono le funzioni di accesso ai dati del documento e rispecchiano la gamma di tipi di applicazioni che è possibile basare sull'architettura Document/View come ad esempio [6]:

- "CScrollView" consente lo scorrimento della vista in situazioni in cui l'area logica è più grande di quella fisica visualizzabile;
- "CTreeView" per visualizzare ed organizzare informazioni in ordine gerarchico;
- "CFormView" consente di far apparire l'applicazione come se fosse il modulo di un database;

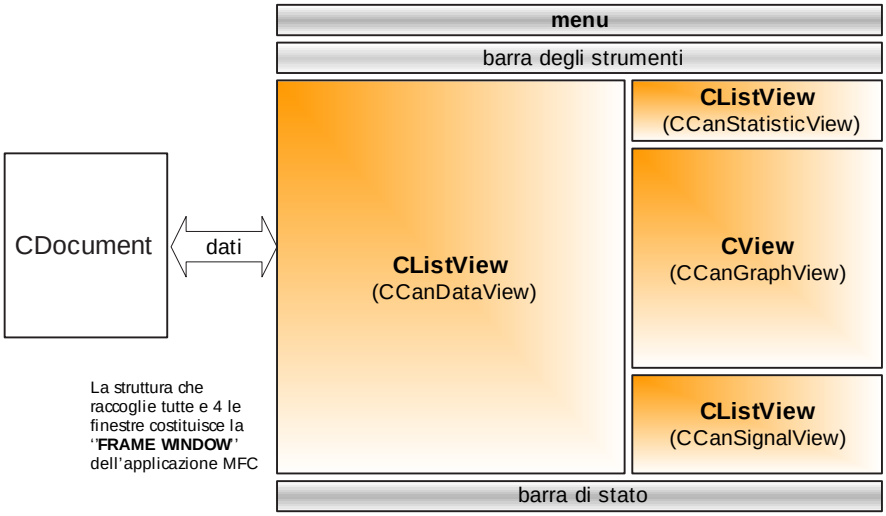


Figura 5.13: CAN*software*: architettura dell'interfaccia.

- "CHtmlView" che utilizza lo scheletro di "Internet Explorer" per consentire lo sviluppo di browser web.

L'interfaccia di CAN*software* è in grado di fornire all'operatore quattro differenti rappresentazioni dei dati ciò avviene per mezzo delle quattro classi vista rappresentate e disposte come in figura 5.13[19]; a tale scopo si impiegano due tipi di classi:

CView : è la classe predefinita che implementa tutte le funzionalità fondamentali relative a una vista compresa l'interazione con il documento.

CListView : utilizza un controllo elenco come vista consentendo all'applicazione di visualizzare colonne di testo.

Le quattro classi interagiscono contemporaneamente con la classe documento e condividono l'unica finestra (Frame Window object) presente in una applicazione SDI[20], e vengono sfruttate per fornire le informazioni seguenti:

[19]Sono indicate le classi derivate da "CView" messe a disposizione da MFC ed utilizzate per l'implementazione delle 4 viste, è riportato anche nome utilizzato come identificatore all'interno del codice di CAN*software*.

[20]Per ottenere questo risultato si sono agganciate alla "Frame Window" due "Splitter Window" derivate dalla classe MFC "SplitterWnd". La prima finestra di splitting divide l'area client in due sottofinestre dette "pane", la seconda di queste viene ulteriormente suddivisa in tre finestre. Ad ogni "pane" viene associata la classe vista opportuna; tutte e 4 le sottofinestre sono ridimensionabili dall'utente trascinandone il contorno.

Mnemonic	**Description**	**Note**	**Default Value**
Index	Numero d'ordine		X
TimeStamps	Istante di arrivo	secondi	X
D/R	Tipo di frame	DATA	D
		REMOTE	R
S/R	Tipo di identificatore	STANDARD	S
		EXTENDED	E
CAN_Id	Identificatore	esadecimale	X
DLC	Numero byte campo CAN data	esadecimale	X
DATA	Byte del campo CAN data	esadecimale	X
ERROR	Tipo di errore/i	ACK Error	A
		FRAME Err.	F
		CRC Error	C
		STUFF Err.	S
FSiZ	Dimensione frame	n.bit	X
SFF	Numero di stuff bit	n.bit	X

Tabella 5.6: CAN*software*: elenco ed interpretazione dei campi della vista "CCanDataView".

CCanDataView : fornisce una rappresentazione in ordine cronologico dei frame che transitano sul bus CAN; il significato ed i valori che possono assumere i vari campi sono riportati in tabella 5.22. Ogni riga di cui si compone la vista rappresenta un "CAN message", le colonne sono i campi di questi record e sono ovviamente comuni a tutti i messaggi che transitano sul bus e che vengono intercettati da CAN*analyser*.

CCanSignalView : riporta l'elenco dei "segnali" creati dall'operatore e fornisce per ognuno di essi i dati descrittivi principali: nome e colore abbinato (Name), unità di misura (Unit), descrizione (Description), idenfificatore associato (ID) e tipo (Type). Il campo "Value" è il risultato delle elaborazioni fatte sui byte del campo DATA associato al messaggio con identificatore ID, impiegando come parametri di calcolo le informazioni passate al sistema in fase di creazione del segnale.

CCanGraphView : fornisce una rappresentazione grafica del dato associato al campo "Value" precedente. Vengono rappresentati solo i segnali abilitati per mezzo delle Check-Box che appaiono nelle righe della vista precedente. Il grafico è ottenuto per mezzo di una semplice interpolazione lineare dei dati ed i segnali analogici e digitali sono gestiti in modo

tale da occupare due distinte sezioni dello stesso "pane"allo scopo di migliorare la risoluzione della rappresentazione grafica.

CCanStatisticView : in questa classe vista derivata da "CListView" vengono visualizzate le statistiche del bus: valore di "WorkLoad" e cioè di occupazione del bus espresso in termini percentuali (campi "WRKL_IST" istantaneo e "WRKL_MED" ottenuto mediando su un numero di campioni specificato dall'operatore) e il numero attuale dei frame visualizzati/acquisiti.

5.2.2 La classe "CCanDATA"

L'applicazione in fase di acquisizione, decodifica ed elabora i dati trasmessi dall'interfaccia hardware come mostrato nel paragrafo 5.1.2.2. La classe "CCanDATA" di CAN*software* si occupa delle fasi di interpretazione e memorizzazione di questi dati e cioè delle operazioni più importanti nella gestione delle informazioni gestite da CAN*analyser*.

Dopo aver creato un oggetto "CCanDATA" per ogni "CAN message" ricevuto questo viene visualizzato attraverso la vista "CCanDataView" e sfruttato per le elaborazioni necessarie a ricavare i valori istantanei dei segnali creati dall'utente che l'applicazione mostra nella vista "CCanSignalView" e "CCanGraphView", mentre le informazioni statistiche sulla rete ricavate da questi dati e vengono mostrate in "CanStatisticView". Nel listato 5.2 sono riportate le variabili e funzioni membro più significative della classe[21] ed in figura 5.15 un esempio di come avviene lo scambio di dati tra questa classe e le viste.

```
1  CObject {
2      DECLARE_SERIAL(CCanDATA);
3
4  public:
5      CCanDATA(bool bCanMsgEn, bool bCanStmpEn);
6      virtual void Serialize(CArchive& ar);
7      unsigned long GetId();
8
9  private:
10     void GenerateCRCsequence();
11     unsigned char GenStuffBitsNum();
12     bool GenNewTimeStamp(double& dTime);
```

[21]Non sono indicate le variabili di appoggio e le funzioni di interfaccia con le altre classi (a titolo esemplificativo si è riportata la sola "GetId()" che viene impiegata per ricavare l'identificatore del messaggio associato all'oggetto "CCanDATA").

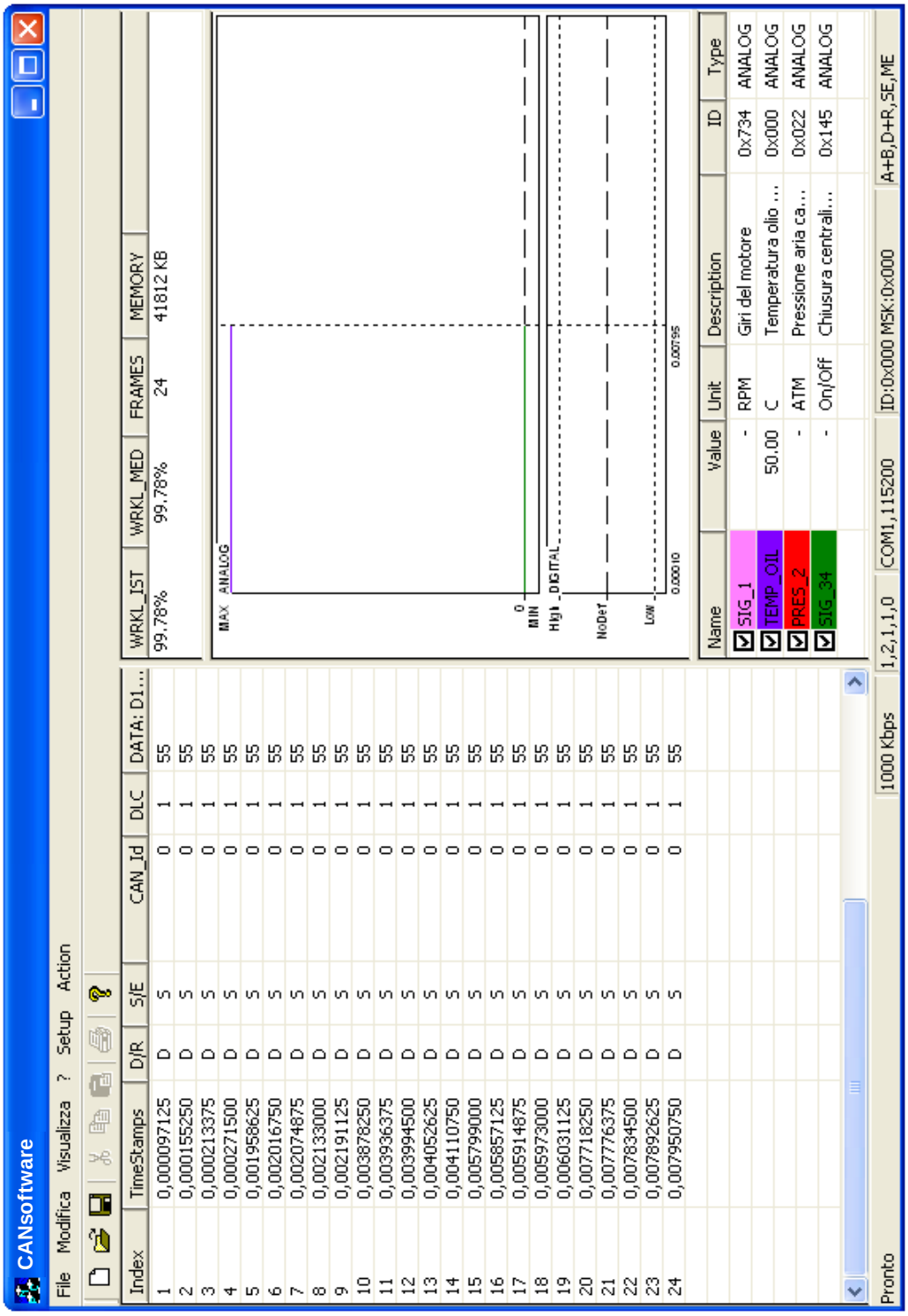


Figura 5.14: CAN*software*: viste dell'applicazione.

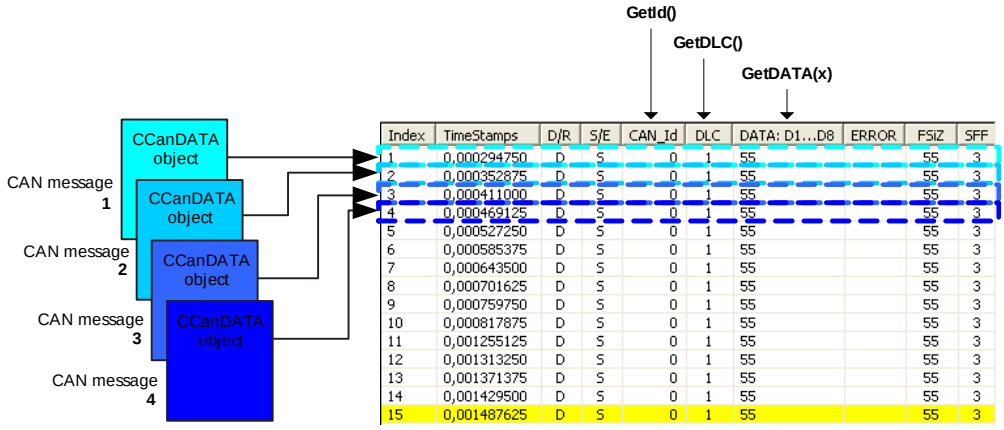


Figura 5.15: MICRO*software*: esemplificazione dell'interazione tra gli oggetti "CCanDATA" ed una delle viste (CCanDataView).

```
13
14      bitset<118> m_DestuffedFrame;
15      bitset<15> m_CRC;
16      tCanIdTagRegs CanId;
17
18      unsigned char m_Status;
19      unsigned char m_byMsg[8];
20      unsigned long m_lngId;
21      unsigned char m_Control;
22      double m_FrameTimeArrive;
23      unsigned char m_StuffedFrameSize;
24      unsigned char m_DestuffedFrameSize;
25  }
```

Listato 5.2: MICRO*software*: parte dichiarativa della classe "CCanDATA".

5.2.2.1 Le variabili membro

Le variabili membro della classe "CCanDATA" sono utilizzate per memorizzare i dati relativi al messaggio CAN intercettato da CAN*interface* sul bus ed inviato da MICRO*software* e quindi conterrano informazioni come: l'identificatore, il "time stamp", il campo DLC, i byte di dato ecc...

Come mostrato nel listato 5.2 al costruttore della classe vengono passati due parametri booleani: "bCanMsgEn" e "bCnaStampEn" attraverso i quali viene comunicato alla funzione se sono o meno attive rispettivamente le funzionalità di invio dei byte del campo DATA e quella di "time stamp"

che l'operatore può decidere di abilitare o meno; da queste informazioni ricava la sequenza attesa dei byte in arrivo da CAN*interface* ("CANMSG" e "CANSTMP").

Il primo controllo che esegue questa funzione consiste nel verificare che il primo byte ricevuto codifichi un "comando" oppure sia il primo byte di un "CAN message" (5.1.2.2); se si tratta di un messaggio esegue le seguenti operazioni:

- il primo byte viene caricato nella variabile "m_Control";
- ricava dai byte "CANIDT1..4" e dal bit "IDE" in "m_Control" l'identificatore del messaggio e lo carica nella variabile "m_lngId[22]";
- dal byte "CANSTCH" ricava i bit che descrivono l'eventuale verificarsi di una delle condizioni di errore previste dal protocollo CAN; queste informazioni, il bit "IDE", il bit "RTRTAG" trasmesso con "CANIDT4" ed il valore attuale dei due booleani passati al costruttore vengono organizzati e memorizzati nella variabile "m_Status". Inglobando tutte queste informazioni codificate attraverso semplici bit (sono tutte variabili booleane) in un unico byte si riduce la quantità di dati che devono essere salvati e gestiti dall'applicazione.
- I byte "CANMSG", se trasmessi, vengono caricati così come sono nell'array "m_byMsg".
- La funzione "GenNewTimeStamp()" se è abilitata la gestione del "time stamp", calcola e salva nella variabile float "m_FrameTimeArrive" l'istante temporale *assoluto* di arrivo del messaggio espresso in secondi. Il valore di questa variabile viene aggiornato ad ogni chiamata della funzione "GenNewTimeStamp()" elaborando opportunamente il contenuto dei byte "CANSTMP" ed dei vari "comandi" che contengono informazioni necessarie alla gestione dello stamping come il "comando di overrun" con cui MICRO*software* comunica l'overflow del registro "CAN timer" e il "comando di overflow" con cui viene trasmesso il valore attuale dello stesso "CAN timer".

[22] La funzione membro "GetId()" precedentemente menzionata non fà altro che ritornare il valore della variabile membro "m_lndId"; nella stessa maniera lavorano molte altre funzioni membro di "CCanDATA" che non sono state riportate nel listato e che sono utili solo per agevolare l'interfacciamento di "CCanDATA" con le altre classi; un'altro esempio è la funzione "GetDATA(x)" che restituisce il byte in posizione "x" nell'array "m_byMsg" che contiene il campo DATA del messaggio CAN.

- "m_StuffedFrameSize" e "m_DestuffedFrameSize" sono due variabili della dimensione di un byte che contengono rispettivamente le dimensioni in numero di bit del frame fisicamente trasmesso sul bus CAN (il frame "stuffato") ed il numero dei bit che compongono il frame in assenza di bit di stuff.

Se il primo byte viene riconosciuto come un "comando" viene anch'esso caricato nella variabile "m_Control"; i byte che CAN*interface* invia al PC devono essere decodificati dalla classe "CCanDATA" che a tale scopo si aiuta con strutture dati più o meno complesse, visto poi che molti dei byte che essa riceve devono essere differentemente interpretati a seconda di certe condizioni le strutture utilizzate sono per lo più delle "union" come ad esempio quella riportata nel listato 5.3 che definisce il tipo "tCtrlCmd" utilizzato per la dichiarazione della variabile "m_Control" che come detto assume significati differenti a seconda che si riceva un "CAN message" o un comando.

```
1  union tCtrlCmd {
2      struct
3      {
4      unsigned char
5                  no_def1:1,       //00000001
6                  EndCmd:1,        //000000?0
7                  CmdBit3:1,       //00000?00
8                  no_def4:1,       //00001000
9                  CmdBit5:1,       //000?0000
10                 OvrRun:1,        //00?00000
11                 OvrBuf:1,        //0?000000
12                 CmdBit8:1;       //?0000000
13     };
14
15     struct
16     {
17     unsigned char
18                 Dlc:4;           //0000????
19     };
20
21     unsigned char
22                 Init;            //????????
23 };
```

Listato 5.3: MICRO*software*: definizione del tipo "tCtrlCmd" utilizzato per definire la variabile membro "m_Control".

In CAN*software* sono molte le strutture di questo tipo, nel caso specifico si può notare che:

- una volta caricato il byte in "m_Control" se il primo e quarto bit valgono "1" significa che si tratta di un "comando"; la verifica viene effettuata in modo elementare esaminando i campi "m_Control.no_def1 e 2".
- I primi 4 bit del byte sono interpretati come "DLC" del frame se "m_Control" contiene il primo byte di un "CAN message";
- se si tratta di un comando diventano significativi i bit indicati con "OvrRun", "OvrBuf", "EndCmd" e tutti gli altri che non vengono attualmente sfruttati;
- infine il campo "Init" è stato inserito per l'inizializzazione della variabile, operazione questa che viene normalmente effettuata con un byte e non bit a bit.

Tornando al lavoro svolto da CAN*software* alla ricezione di un comando, se viene riconosciuto un comando di overrun o di overbuffer (inviati da MICRO*software* solo se è abilitata la funzione di "time stamp") viene richiamata al funzione di aggiornamento "GenNewTimeStamp()".

5.2.2.2 Le funzioni membro

Serialize() implementa il salvataggio su file delle variabili membro appena descritte sfruttando il meccanismo della serializzazione [6] che il framework MFC mette a disposizione per le classi derivate da "CObject" come è stata implementata appositamente per sfruttare questa metodologia la classe "CCanDATA".

GenerateCRCsequence() Durante la fase di decodifica dei byte del "CAN message" il costruttore "CCanDATA()" sfrutta la variabile "m_DestuffedFrame[23]" per ricostruire bit dopo bit la struttura originale del frame trasmesso sul bus in assenza dei bit di stuff e limitatamente alla parte del messaggio che precede il campo "CRC" e che quindi rientra nella "Stuffed Area" (la parte del frame sottoposta a stuffing dei bit). Successivamente, la funzione "GenerateCRCsequence()" riportata nel listato 5.4 a pagina 161 ricostruisce la sequenza CRC e la accoda alla variabile "m_DestuffedFrame" (la funzione "GenerateCRCsequence()" implementa in C++ quanto descritto in 3.1.2.4).

[23] Questa variabile è di tipo "bitset", la classe "bitset" appartiene alle "Standard Template Library" o "STL" e fornisce tutta una serie di funzioni che consentono di costruire una variabile composta da un numero arbitrario di bit e che cosentono di accedere ai singoli bit dell'oggetto "bitset" (è possibile settare e resettare singolarmente i bit della variabile.

GenStuffBitsNum() Sulla sequenza di bit "m_DestuffedFrame" generata dalla precedente funzione va a lavorare la "GenStuffBitsNum()" che si occupa di calcolare il numero di bit di stuff che il CAN controller del nodo trasmettitore ha inserito nel messaggio ed aggiorna il valore delle due variabili "m_StuffedFrameSize" e "m_DestuffedFrameSize"; queste informazioni sono necessarie per poter calcolare il parametro di bus load della rete.

Esistono poi tutta una serie di funzioni di interfaccia attraverso cui la classe interagisce con il resto del sistema; in precedenza abbiamo menzionato due di queste funzione: "GetId()" che restituisce il valore della variabile privata contenente l'identificatore del messaggio "m_lngId" (unsigned long) o la funzione "GetDATA(x)" che ritorna l'x-mo byte del campo DATA del frame.

```
1  CCanDATA::GenerateCRCsequence() {
2      bool CRCNXT;
3      int Valore;
4
5      // m_CRC                 -> CRC_RG
6      // m_DestuffedFrame[i]   -> NXTBIT
7
8      m_CRC.reset();
9      for (int i=0; i<m_SizeOfSequenceToStaff; i++)
10     {
11         CRCNXT = m_DestuffedFrame[i] ∧ m_CRC[14];
12         m_CRC <<= 1;
13         if (CRCNXT)
14             m_CRC ∧= 0x4599;
15     }
16
17     for (int j=14; j>=0; j--)
18     {
19         if (m_CRC[j])
20             Valore = 1;
21         else
22             Valore = 0;
23     }
24 }
```

Listato 5.4: MICRO*software*: implementazione della funzione di generazione del campo CRC del frame per mezzo della funzione "GenerateCRCsequence".

5.2.3 CAN*software*: applicazione multithread

Durante la fase acquisizione dati il lavoro svolto da CAN*software* può essere suddiviso nelle seguenti attività:

Task 1 rimane in attesa di ricevere dati dall'interfaccia CAN*interface* per mezzo del collegamento seriale con il PC; i byte ricevuti vengono caricati in una struttura dati di tipo "CCoda";

Task 2 attraverso la procedura "Thread_ListenMode" i dati caricati in "CCoda" vengono decodificati e gli oggetti di tipo "CCanDATA" che vengono costruiti durante questa fase di elaborazione vengono organizzati in un array di oggetti chiamato "m_canArray" di tipo "CObArray": ogni oggetto caricato in questo vettore è un "CAN message" o un comando;

Task 3 gestisce l'interazione con l'operatore e cioè: visualizza i dati nelle quattro viste disponibili, rileva le richieste dell'utente e le inoltra.

Queste operazioni costituiscono tre task che il processo porta avanti in modo "simultaneo". I tre task sono implementati per mezzo di altrettanti "thread" e la sincronizzazione per l'utilizzo delle risorse condivise e cioè delle strutture dati di tipo "CCoda" e "CObArray" avviene per mezzo di meccanismi di protezione come sezioni critiche e semafori messi a disposizione del sistema operativo[24].

Il vantaggio nel realizzare CAN*software* come un'applicazione multithread è soprattutto quello di sfruttare a pieno la limitata banda passante della porta seriale: il task di gestione di questa risorsa di comunicazione si deve occupare solo di prelevare il dato letto dalla porta COM[25] e memorizzarlo nella struttura dati di tipo "CCoda", a questo punto può predisporsi per una nuova lettura; è compito di un'altro task che lavora in "parallelo" a questo occuparsi di prelevare i dati da "CCoda" ed elaborarli. *I tempi di latenza che intercorrono tra una lettura e la successiva sono minimi.*

5.2.3.1 La classe "CCoda"

"CCoda" implementa e fornisce le funzioni di interfaccia necessarie alla gestione di una coda circolare di byte in modalità FIFO; la parte dichiarativa è riportata nel listato 5.5.

[24]Si ricorda che è l'OS che deve fornire e supportare la gestione di risorse condivise tra task e tra processi per mezzo di opportune primitive utilizzate per rendere atomiche le funzioni di elaborazione svolte su queste risorse.

[25]In Windows l'interazione con le risorse di sistema ed in questo particolare caso con la porta seriale è filtrata attraverso il driver di gestione della stessa.

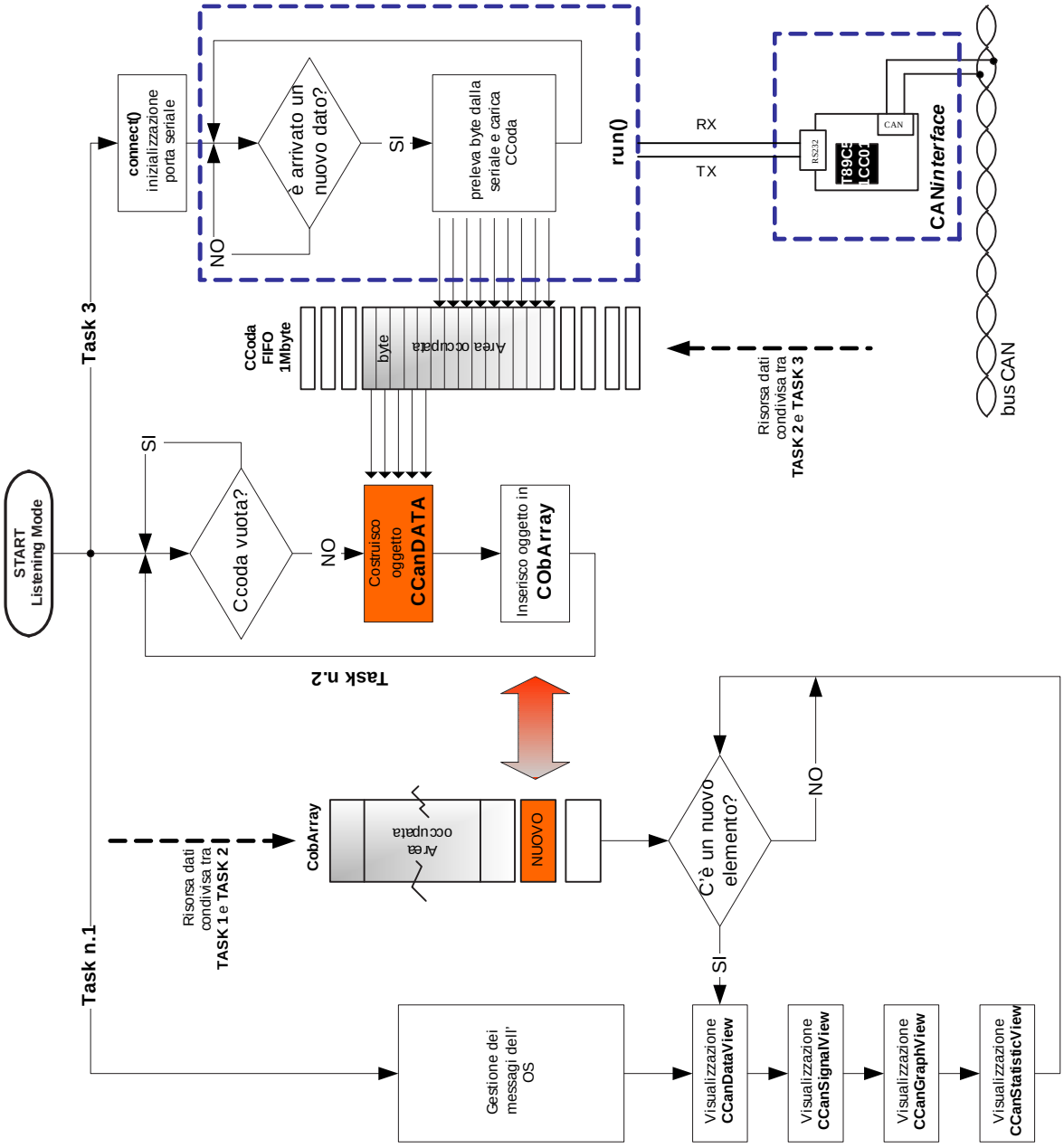


Figura 5.16: CAN*software*: schematizzazione delle strutture dati condivise tra i task e delle operazioni che interagiscono con esse.

```
1  class CCoda { private:
2      // gestione
3      unsigned long m_FirstOUT;
4      unsigned long m_LastIN;
5      unsigned long m_SizeCoda;
6
7      // handle semafori
8      HANDLE m_hsem_IsFull;
9      HANDLE m_hsem_IsEmpty;
10
11     // implementazione coda
12     BYTE m_BufferCoda[SizeCoda];
13
14 public:
15     CCoda();
16
17     // estrazione
18     void Pop (InfoType&);
19     bool PopWithTimeOut(InfoType&, DWORD);
20
21     // inserimento
22     void Push (InfoType);
23     InfoType* GetBufferPointer(int);
24     void ReleaseIsEmpty(int);
25 }
```

Listato 5.5: CAN*software*: parte dichiarativa della classe "CCoda".

La coda è realizzata per mezzo dell'array di byte "m_BufferCoda", le variabili "m_FirstOUT" e "m_LastIN" sono rispettivamente sfruttate come indici del primo elemento inserito (il primo da estrarre) e come indice del primo elemento libero (quello in cui inserire il nuovo byte prelevato dalla seriale), la funzione "Pop()" ritorna l'elemento in posizione "m_FirstOUT" mentre la "Push()" inserisce il dato (passato come paramentro) nell'array in posizione "m_LastIN".

Come mostrato in figura 5.16 un oggetto di classe "CCoda" viene impiegato come risorsa condivisa tra il task di gestione della seriale (Task 3) e quello di costruzione degli oggetti "CCanDATA" (Task 2); il Task 2 estrae dati dall'array mentre il Task 3 lo carica con i byte letti dal buffer della seriale.

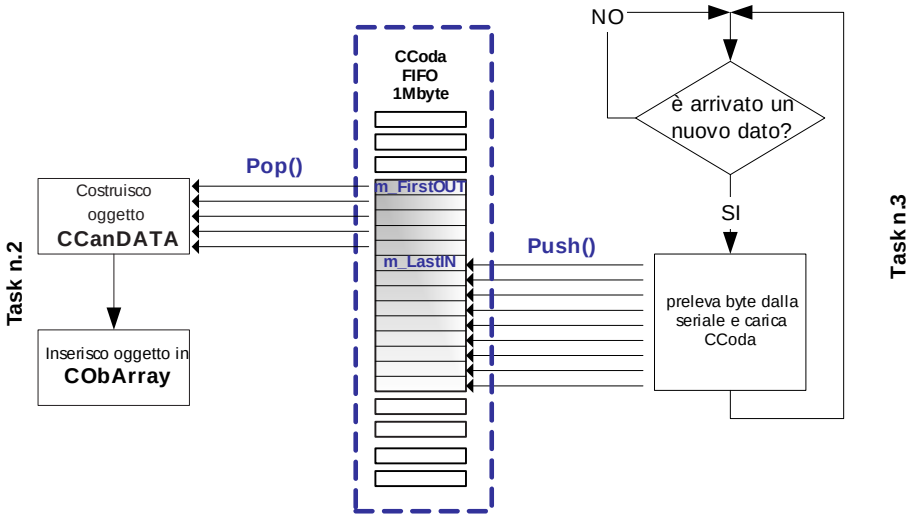


Figura 5.17: CAN*software*: implementazione di "CCoda"

La risorsa viene condivisa tra i due thread per mezzo di due semafori creati e gestiti sfruttando direttamente le API di Windows[26]: "CreateSemaphore", "ReleaseSemaphore" e "WaitForSingleObject" per mezzo delle quali le operazioni di "Wait[27]" e "Signal[28]" vengono eseguite dal sistema operativo in modo atomico; *i due semafori assicurano che i task non possano avere contemporaneamente accesso allo stesso elemento dell'array.*

In una prima versione la coda era stata implementata per mezzo di una lista che se da un lato consentiva di avere una struttura la cui dimensione variava dinamicamente e si adattava alle esigenze del momento[29], dall'altro rendeva molto più complessa e lenta l'implementazione di un efficiente meccanismo di protezione. Utilizzando un semplice array la dimensione della struttura è fissa costituisce cioè un parametro di progetto[30] però con soli due semafori i due task possono accedere contemporaneamente alla struttura in modo protetto (non accedono contemporaneamente solo se stanno lavorando

[26]In questo caso l'utilizzo delle MFC è risultato limitante per le particolari funzionalità che si volevano implementare.

[27]Blocca l'accesso alla risorsa (nel nostro caso al singolo elemento della struttura dati) se c'è un'altro task che ci stà lavorando sopra, in caso contrario fornisce l'accesso e fà in modo di bloccare qualunque altro task ne faccia richiesta.

[28]Attraverso di essa il task che stava utilizzando la risorsa condivisa la libera consentendo ad eventuali altri task che erano in attesa di avere accesso alla stessa.

[29]Veniva impegnata solo la memoria necessaria a contenere i dati inseriti dal Task 3 e non ancora prelevati dal Task 2.

[30]La dimensione dell'array può essere impostata in fase di compilazione accedendo alla costante "SizeCoda" nel file "Parametri.h; nella versione attuale la coda è ampia 1Mbyte.

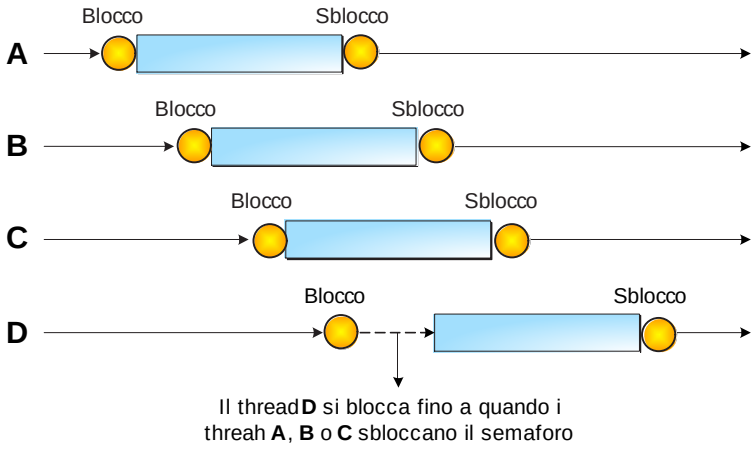


Figura 5.18: Esempio di funzionamento delle primitive semaforiche: i task A, B, C e D condividono la stessa risorsa per mezzo di un semaforo che consente l'accesso contemporaneo alla risorsa ad un massimo di 3 task (semaforo inizializzato a 3).

sullo stesso elemento dell'array).

L'accesso "contemporaneo" all'array (Task 2 che esegue una "Pop()" ed il Task 3 che esegue una "Push()") crea problemi solo se l'indice "m_LastIN" coincide con "m_FirstOUT"; solo in questo caso "Push()" e "Pop()" lavorebbero sul solito elemento dell'array. In realtà i due indici sono uguali solo se la coda è piena o è vuota ma in questi due casi uno dei due semafori blocca l'esecuzione della funzione infatti:

- se la coda è piena il semaforo "IsFull" blocca la "Push()" non essendo disponibile un elemento vuoto in cui inserire un nuovo byte;
- se la coda è vuota "IsEmpty" blocca la "Pop()" perché non ci sono elementi da estrarre.

5.2.3.2 Task 3: la gestione della seriale

Il Task 3 gestisce la porta seriale del PC interfacciandosi con il driver di Windows per mezzo del quale il programmatore vede la risorsa di comunicazione. Quando l'operatore attiva l'acquisizione apre una connessione con l'interfaccia CAN*interface*, ciò avviene tramite la creazione di un oggetto di classe "Tserial_event" per mezzo del quale la porta seriale viene gestita in modalità *event-driven*. Una volta che la risorsa di comunicazione è stata inizializzata l'oggetto di tipo "Tserial_event" crea e manda in esecuzione un thread (Task 3) a cui viene passato come parametro l'handle della funzione

"run()[31]" di cui riportiamo in figura 5.19 il relativo diagramma di flusso. In modo particolare si fà notare che [7]:

- la porta di comunicazione viene "aperta" in modalità "overlapped[32] per mezzo della API di Windows "CreateFile()"; il driver di supporto alla comunicazione consente di gestire la risorsa come se si trattasse di un file.
- Il Task 3 viene mandato in esecuzione chiamando la API "_beginthreadex".
- La funzione "run()" rimane in attesa di un evento per mezzo della API "WaitForMultipleObject()" che ritorna l'indice identificativo dell'evento a più alta priorità tra quelli che si sono verificati e per mezzo dell'"Event Dispatcher" realizzato con un costrutto "Case of" la richiesta viene inoltrata ed eseguita.
- La lettura di un dato,viene effettuata attraverso la API "ReadFile()" a cui bisogna passare l'indirizzo di un'area di memoria disponibile in cui caricare i dati; alla generazione di questo indirizzo ci pensa la funzione "GetBufferPointer()" di "CCoda" che ritorna un puntatore al primo elemento libero della struttura "CCoda[33]". Avendo abilitato la porta in modalità "overlapped" l'esecuzione del codice non si arresta alla "ReadFile" anche se i dati non sono attualmente disponibili.

5.2.3.3 Task 2: costruzione degli oggetti CAN

La funzione globale "Thread_ListenMode()" implementa il secondo Task. Gli oggetti di classe "CCanDATA" che produce vengono salvati in una struttura dati MFC di tipo "CObArray" chiamata "m_CanArray"; si tratta di una struttura dati con le funzionalità di un vettore ma di dimensione che cambia dinamicamente come una lista. L'impiego di un oggetto di tipo "CObArray"

[31]La creazione di un thread richiede di identificare tramite il suo handle la funzione che dovrà essere mandata in esecuzione all'attivazione del thread stesso; possiamo dire che la funzione passata costituisce il task associato al thread. Nel caso di CAN*software* il Task 3 è implementato attraverso la funzione "run()".

[32]In una operazione di I/O di tipo "nonoverlapped" il thread chiamante rimane bloccato fino al termine dell'esecuzione della operazione e per tutto questo tempo blocca l'acceso alla risorsa. Nel caso di "overlapped I/O" più thread possono fare operazioni di I/O allo stesso tempo e mentre queste sono in attesa di essere completate possono essere svolti altri compiti.

[33] La "ReadFile" in questo modo praticamente svolge anche la funzione di una push infatti i dati prelevati da seriale vengono caricati dalla stessa "ReadFile" nella coda, le prestazioni sono ulteriormente migliorate non essendoci la necessità di invocare la "Push()".

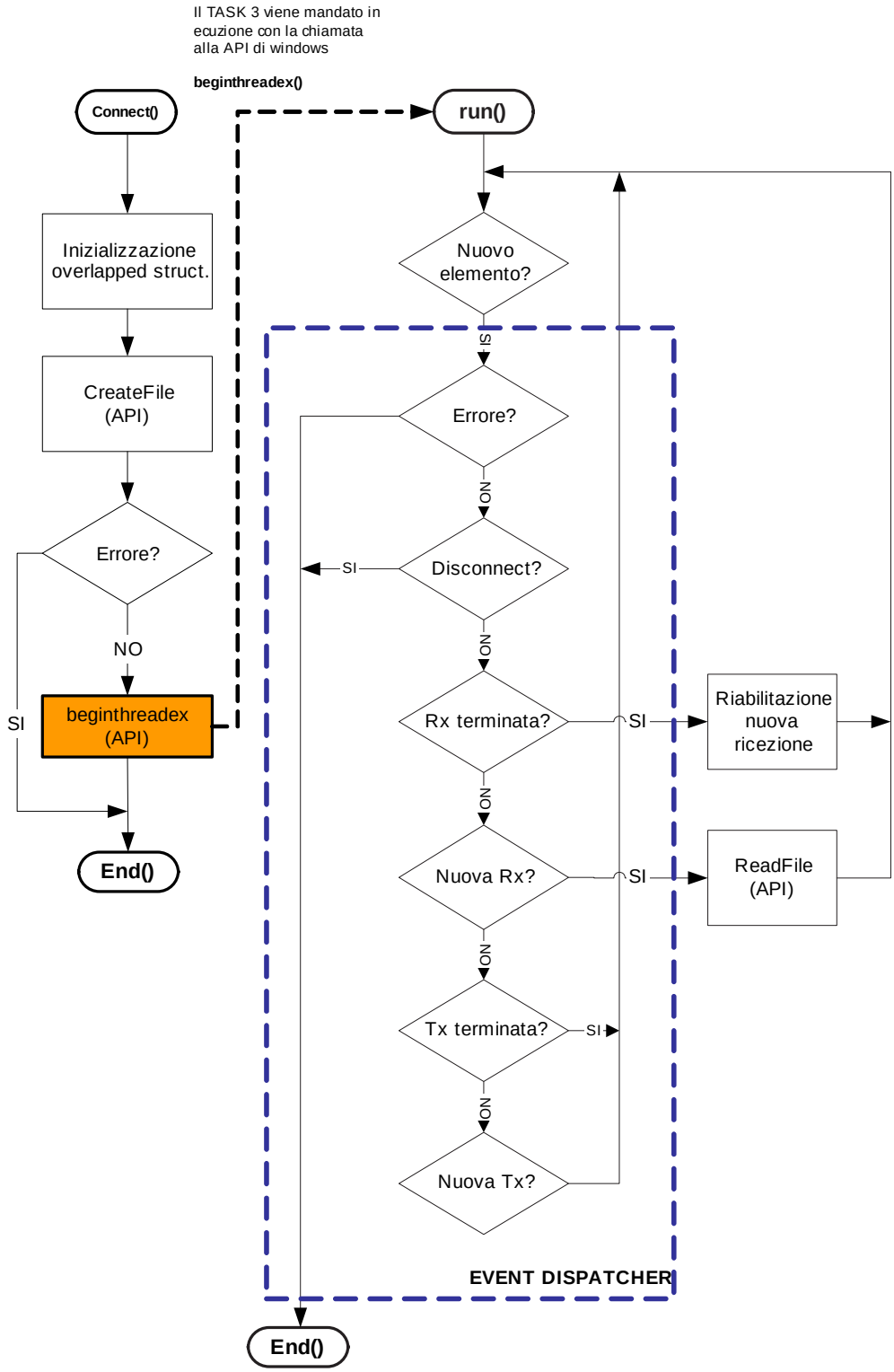


Figura 5.19: CAN*software*: diagramma di flusso a blocchi della funzione “run()”.

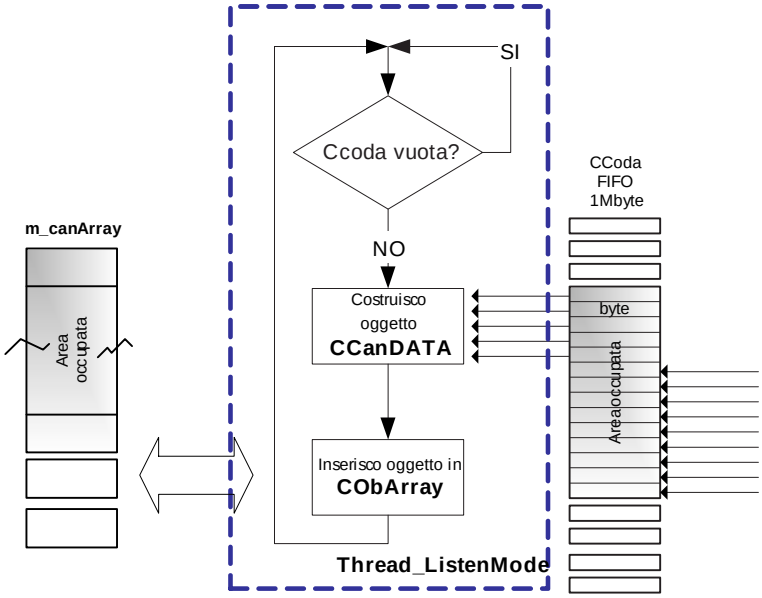


Figura 5.20: CAN*software*: task di costruzione degli oggetti "CCanDATA".

come struttura dati in cui contenere gli oggetti CAN ci consente di tenere in memoria i messaggi prima di salvarli su disco rigido e senza la preoccupazione di dover gestire liste di dati molto lunghe (ci pensano le funzioni MFC membro della classe "CObArray").

Il semplice codice della funzione "Thread_ListenMode" è riportato nel listato 5.6 ripulito della parte dichiarativa delle variabili locali e di alcune istruzioni di inizializzazione.

```
1  UINT Thread_ListenMode (LPVOID pParam) {
2
3      // si esce dal ciclo WHILE in due condizioni:
4      //
5      // - MEMORIA ESAURITA
6      // - DISCONNESSIONE
7
8      while ((pCanDATAElem = new CCanDATA(
          bMessageEnablePar,bStampEnablePar)) != 0)
9      {
10         // SI BLOCCA LA RISORSA
11         canArray_cs.Lock();
12
13         if (!pCanDATAElem->IsCmd())
14            pListenMode->m_ListenModeStat.NumFrameArrived
                ++;
```

```
15
16          try
17          {
18              pListenMode->m_canArray.Add(pCanDATAElem);
19          }
20              catch (CMemoryException cme)
21              {
22                  AfxMessageBox ("ERRORE␣-->>␣RISORSE␣DI␣
                       SISTEMA␣ESAURITE");
23                  break;
24              }
25
26          // TIME STAMP
27          pListenMode->AggFrameTimeArrive(
               IndexDataPacketArrived);
28
29          // CONTATORE FRAME
30          IndexDataPacketArrived++;
31
32          // RICEVUTO COMANDO DI DISCONNESSIONE?
33          if (pCanDATAElem->IsCmd() && pCanDATAElem->
               GetEndCmd())
34          {
35              iCurrentSize = pListenMode->m_canArray.
                   GetSize() - 1;
36              delete pListenMode->m_canArray.GetAt(
                   iCurrentSize);
37              pListenMode->m_canArray.RemoveAt(
                   iCurrentSize,1);
38              canArray_cs.Unlock();
39              break;
40          }
41
42          // VISUALIZZAZIONE DATO IN ARRIVO
43          ::PostMessage(h_CanDataView,WM_TIMER,
               ID_UpdateViewTimer,NULL);
44
45          // LIBERO RISORSA CONDIVISA
46          canArray_cs.Unlock();
47      }
48  }
```

Listato 5.6: CAN*software*: codice della procedura "Thread_ListenMode" tramite cui si implementa il Task2.

Attraverso il ciclo "while" che termina solo alla ricezione dell'ack di CAN*interface* al comando di disconnessione inviato (riga 33-40), il thread passa al costruttore di "CCanDATA" i byte prelevati dalla coda (riga 8) e carica l'array con l'oggetto così costruito (riga 18), vengono successivamente aggiornate alcune variabili come "time stamp" (riga 27) e il contatore del numero di frame (riga 30).

Mentre la protezione della struttura "CCoda" avviene tramite semafori, la protezione dell'array "m_CanArray" è gestita attraverso le "sezioni critiche"; alla riga 11 la risorsa viene bloccata e riservata al task mentre viene liberata solo dopo aver terminato il ciclo di elaborazione.

5.2.3.4 Task 1: visualizzazione dei dati

Mentre Task 2 carica il vettore il Task 1 ne preleva gli elementi attualmente disponibili per visualizzarne il contenuto; come Task 2 anche Task 1 implementa la protezione della risorsa tramite sezioni critiche (i due task non potranno mai lavorare contemporaneamente sull'array).

Tutte le classi derivate da "CView" dispongono di una funzione membro "OnUpdate" dove preferibilmente inserire il codice che consente l'aggiornamento della vista. Ognuna delle nostre quattro classi implementa una propria funzione di aggiornamento dei dati in modo del tutto personale, queste funzioni vengono chiamate in successione per consentire il refresh continuo dei dati in tutte e quattro le aree client in cui si è suddivisa la finestra principale.

In modo particolare:

CCanDataView i messaggi CAN vengono organizzati nella lista visualizzata in figura **??** i cui campi sono stati dettagliatamente descritti in tabella 5.22. Se si verifica una condizione di overflow (quindi dei messaggi vengono persi) questa viene notificata all'operatore con una riga gialla in corrispondenza dell'ultimo messaggio valido ricevuto prima della reinizializzazione del sistema, mentre se viene rilevato un errore su uno dei messaggi il frame "corrotto" viene indicato da una riga rossa mentre la condizione di errore è codificata nel campo "ERROR" della vista.

CCanSignalView alla ricezione dei messaggi vengono calcolati i valori dei segnali e visualizzati in questa lista aggiornando il campo "VALUE". Nel caso in cui il segnale sia analogico e l'operatore abbia impostato uno o più allarmi, al verificarsi di una condizione anomala viene modificato il colore di sfondo del campo "VALUE": se il valore calcolato supera la

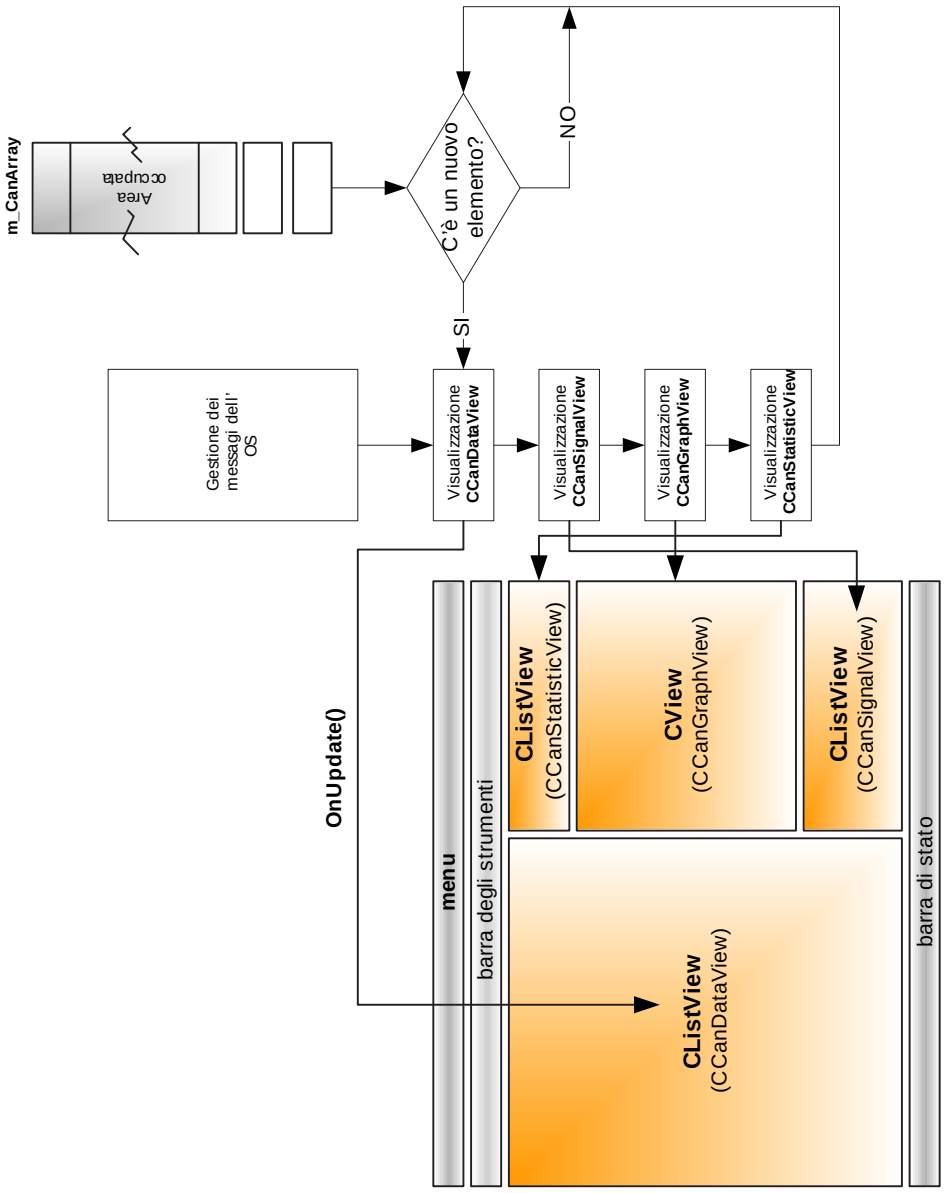


Figura 5.21: CAN*software*: meccanismo di visualizzazione dei dati attraverso le 4 viste disponibili.

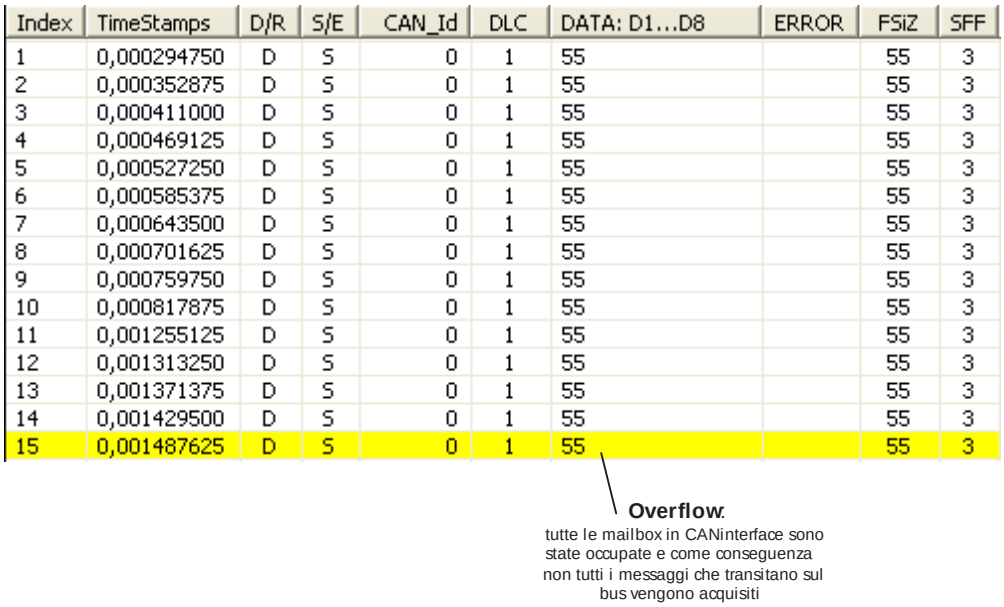


Figura 5.22: MICRO*software*: visualizzazione dei messaggi.

soglia alta impostata il campo diventa di colore rosso, se viene superata la soglia limite inferiore il campo diventa blu (figura 5.23).

Nelle altre due viste di figura 5.24 e 5.25 viene fornita la rappresentazione grafica dei segnali ed alcune informazioni statistiche sul bus.

5.2.4 Menu: "setup"

La prima volta che CAN*software* viene mandato in esecuzione i parametri di configurazione dell'interfaccia CAN*interface* vengono impostati automaticamente a dei valori di default, successivamente l'operatore può modificarli accedendo al menu "setup" per poterli adattare alla rete CAN a cui l'analizzatore viene collegato; questi parametri vengono dall'applicazione salvati all'interno del file di log "log.txt" che CAN*software* genera dopo la prima esecuzione del programma[34]. CAN*software* utilizza queste informazioni per agevolare l'operatore che in questo modo se ad esempio usa sempre una certa configurazione non deve reimpostarla ogni volta; i dati nel file di log vengono aggiornati ad ogni variazione degli stessi.

[34]Quando CAN*software* viene mandato in esecuzione verifica che nella directory in cui si trova l'eseguibile sia presente anche il file "log.txt" da cui recupera i parametri di inizializzazione delle varie dialog. Se il file di log non viene trovato l'utente viene avvertito di questa mancanza ed un nuovo file "log.txt" viene generato.

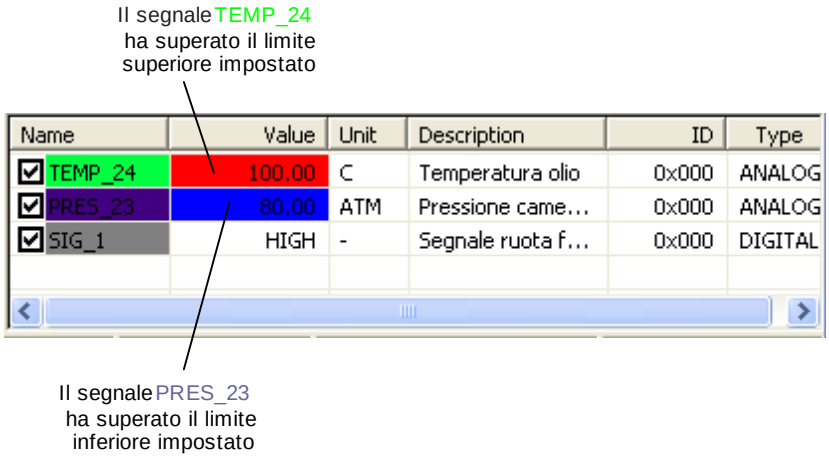


Figura 5.23: MICRO*software*: visualizzazione dei "segnali".

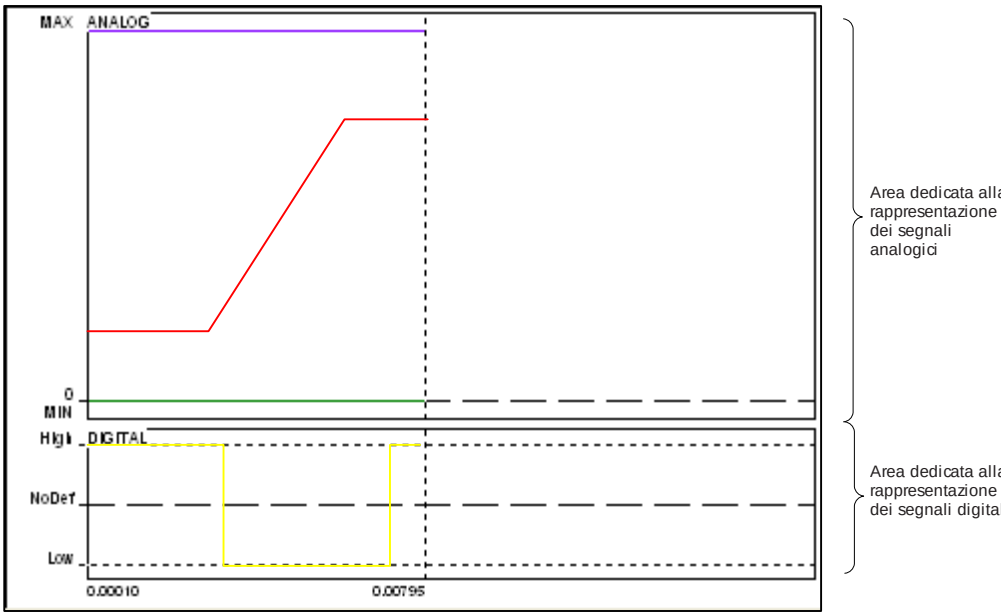


Figura 5.24: MICRO*software*: visualizzazione grafica dei "segnali".

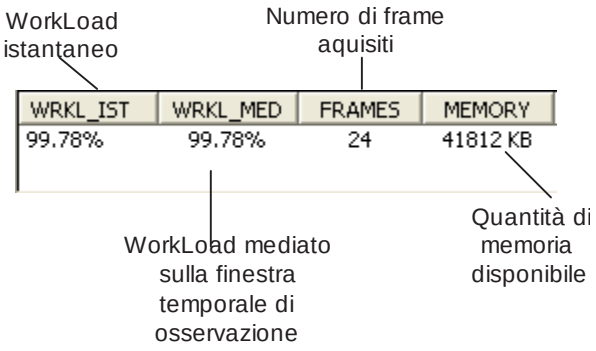


Figura 5.25: MICRO*software*: visualizzazione delle statistiche.

Attraverso le voci presenti nel drop-down menu "setup" l'operatore accede alle dialog tramite cui configurare i parametri di funzionamento dell'analizzatore di traffico.

5.2.4.1 Parametri di bit timing

L'opzione "CAN Bit Timing" del menu "setup" consente di impostare i parametri omonimi del CAN controller di CAN*interface* ed a tale scopo l'applicazione aiuta l'operatore nella scelta della configurazione adeguata alla rete sotto test. Selezionata questa voce si apre una finestra o dialog mostrata in figura 5.26 che consente all'operatore di impostare i parametri desiderati.

Una volta selezionato il baudrate e premuto il pulsante "Calcola" CAN*software* genera tutte le possibili configurazioni di bit timing[35] corrispondenti al bit rate specificato e le visualizza nella tabella "Bit Timing Parameters" ricavata nella dialog e mostrata in figura 5.27 dove sono riportate le seguenti informazioni (si veda a tale proposito anche l'appendice C):

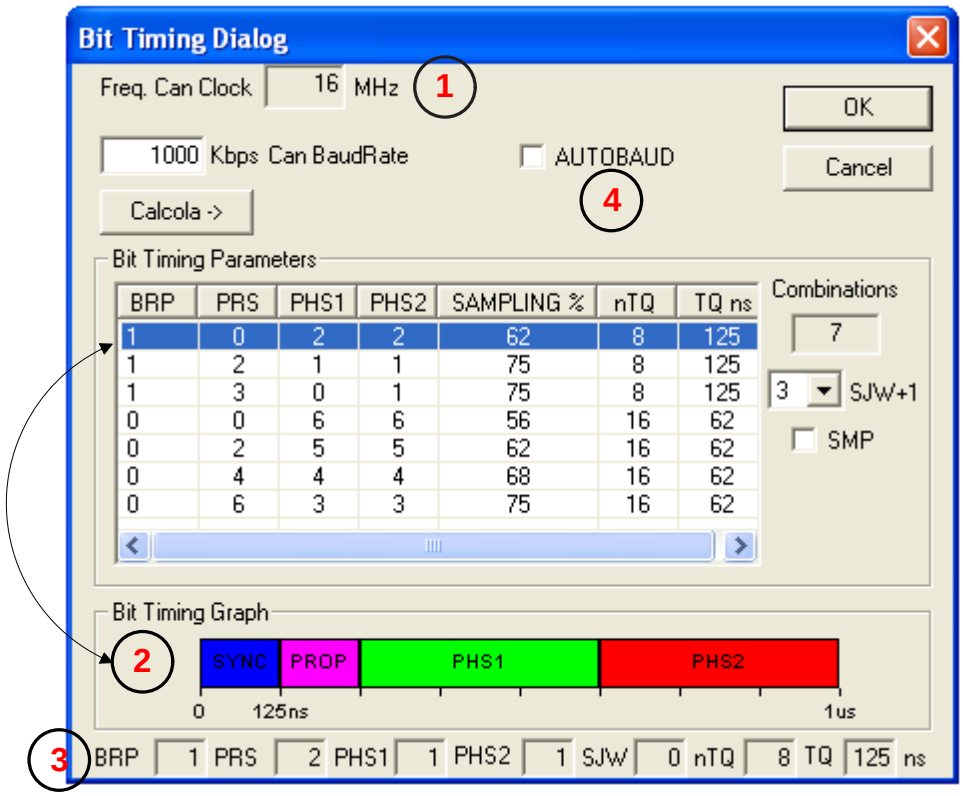
BRP riporta il valore a cui deve essere impostato il registro di "Baudrate Prescaler" del microcontrollore;

PRS riporta il "Propagation Delay Segment";

PHS1 il valore del "Phase Segment 1";

PHS2 il valore del "Phase Segment 2";

[35]Il software ricava queste configurazioni nel rispetto dei criteri specifici del protocollo CAN 2.0B e che sono stati descritti in 3.1.7.



Barra di stato della finestra. Sono riepilogati i parametri di bit timing con cui è attualmente configurata CAN*interface*.

Figura 5.26: CAN*software*: finestra per l'impostazione dei parametri di bit timing della rete CAN.

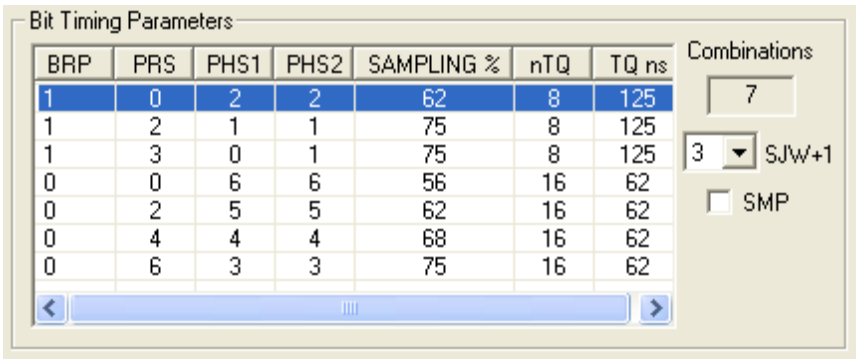


Figura 5.27: CAN*software*: tabella riepilogativa dei parametri di configurazione.

SAMPLING % corrisponde alla valutazione della collocazione del "Nominal Sample Point" espressa in termini percentuali rispetto alla durata del "bit time";

nTQ riporta il numero di Time Quanta che compongono il bit time nella configurazione associata alla riga;

TQ (ns) esprime in nanosecondi la durata del Time Quanta;

Combinations riporta il numero totale di combinazione disponibili per il baudrate selezionato;

SJW contiene la lista dei valori possibili per il parametro di "Resynchronization Segment";

SMP è una check-box che consente di selezionare la mondalità a campionamento singolo o multiplo.

L'operatore effettuerà la scelta tra le configurazioni proposte valutando l'istante di campionamento e in base al valore di massimo ammissibile per "SJW". Individuata la configurazione adatta alla rete è sufficiente selezionare nella tabella la riga corrispondente e premere il pulsante "OK" per dare conferma della scelta; la configurazione evidenziata diviene quella corrente. La dialog riporta anche altre utili informazioni:

1. riporta la frequenza di lavoro dell'interfaccia CAN*interface*;
2. visualizza graficamente il "bit time" relativo alla configurazione selezionata nella lista, distinguendo i diversi segmenti che lo compongono;
3. nella barra di stato posta sotto il grafico sono riepilogati i parametri con cui CAN*interface* è attualmente impostata;

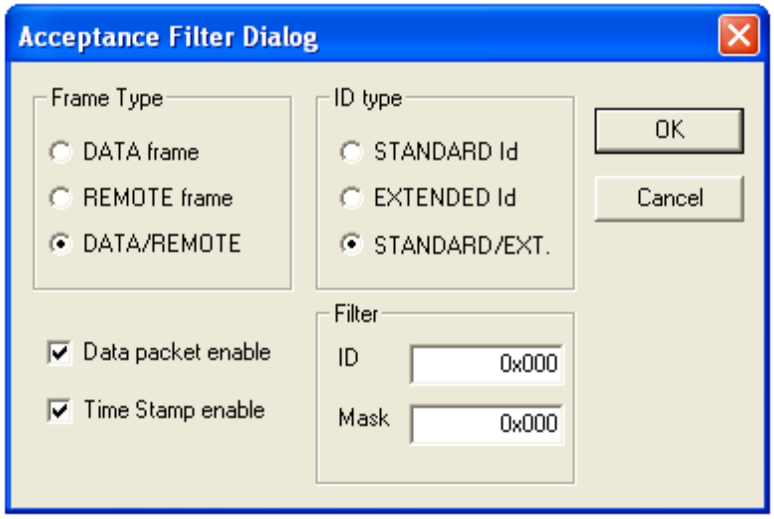


Figura 5.28: CAN*software*: finestra di impostazione dei parametri di filtro dei messaggi.

4. per mezzo di questa check-box l'operatore può attivare la funzionalità di hot-plugging. Una volta che la check-box viene selezionata CAN*software* non da più la possibilità di accedere ai parametri di configurazione poiché questi verranno automaticamente calcolati da CAN*interface*.

5.2.4.2 Filtro dei messaggi

I messaggi che transitano sulla rete possono in alcuni casi possono risultare solo di parziale interesse per chi deve analizzare il traffico; per mezzo della finestra "Acceptance Filter Dialog" alla quale si accede selezionano la voce "CAN Filter" dal menu "setup" si ha la possibilità di impostare i parametri con cui filtrare i frame in arrivo selezionando solo quelli che si ritengono effettivamente *utili*.

CAN*software* non esegue a posteriori alcuna operazione di selezione dei messaggi; il filtro viene eseguito direttamente dal microcontrollore di CAN*interface* in parte via hardware (programmando i canali in ricezione con ID e maschere opportune ed impostando il tipo DATA o REMOTE che si vuole ricevere, il meccanismo di "Acceptance Filter" del T89C51CC01 fà il resto) ed in parte via software (è lo stesso MICRO*software* che decide se inviare il campo DATA del frame e le informazioni per la gestione del "time stamp"). Il vantaggio di questa soluzione è quello di limitare il numero di byte che CAN*software* e CAN*interface* devono scambiarsi attraverso il canale

di comunicazione che purtroppo ha nel nostro caso una banda estremamente limitata (tabella 5.4). In figura 5.28 si è riportato lo screenshot della finestra di impostazione del filtro nella quale sono presenti i seguenti campi:

Frame Type : si può decidere se acquisire solo frame di tipo DATA, REMOTE o entrambi (allo scopo vengono impostati adeguatamente i bit "RTRTAG" e "RTRMSK" rispettivamente in "CANIDT4" e "CANMSK4" di tabella 5.1).

ID Type : l'analizzatore è in grado di gestire indifferentemente il protocollo CAN nelle versioni 2.0A e 2.0B e con questa opzione si seleziona se CAN*interface* deve o meno accettare messaggi con identificatori con ID standard (11 bit) e/o ID estesi (29 bit) (vedi "cmdIdeTag" in tabella 5.2).

Filter : In certe applicazioni l'utente può essere interessato ad uno specifico gruppo di ID oppure non può rischiare di perdere dati che transitano su un bus estremamente affollato; attraverso i campi "ID" e "Mask" si è in grado di specificare il gruppo di identificatori dei messaggi che interessano (registri "CANIDT" e "CANMSK" di tabella 5.1).

Data packet enable : Abilita CAN*interface* ad inviare il campo DATA dei frame acquisiti (vedi "cmdMessageEnable" di tabella 5.2).

Time Stamp enable : Abilita CAN*interface* ad inviare i byte di "time stamp" associati ai messaggi e gli eventuali comandi di timing (vedi "cmdTimeStampEnable" di tabella 5.2).

5.2.4.3 Gestione dei "segnali"

CAN*software* consente all'utente di convertire i dati ricevuti (campo DATA) attraverso i CAN DATA Frame in quantità fisiche significative detti "segnali" e consente all'operatore di creare e gestire un database di questi segnali contenente le informazioni necessarie per effettuare la conversione. L'accesso al database per la definizione dei segnali e/o la loro modifica avviene selezionando dal menu "setup" la voce "Edit Signals".

La figura 5.29 mostra la finestra che si apre quando si accede al database. Creati i segnali di cui si ha bisogno fornendo i parametri di identificazione (nome, descrizione, unità di misura e commenti vari) ed i parametri di calcolo utilizzati da CAN*software* alla ricezione dei messaggi CAN per elaborarne il valore, i segnali vengono riportati anche nella finestra gestita dalla vista "CCanSignalView" nella quale sono visualizzati solo i parametri descrittivi

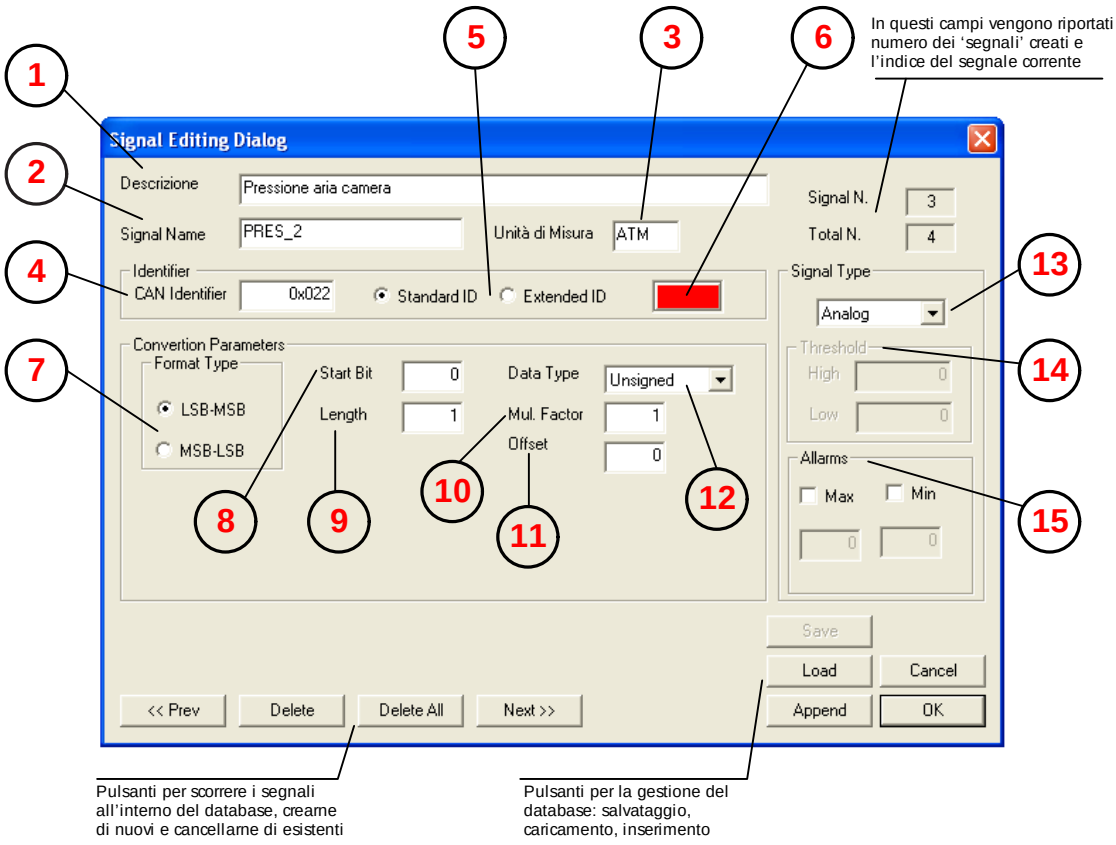


Figura 5.29: CAN*software*: finestra di editing dei segnali.

Figura 5.30: CAN*software*: visualizzazione dei segnali.

(vedi figura 5.30) ed il campo "Value" che conterrà l'ultimo valore elaborato.

Per capire che cosa sia un segnale possiamo fare un esempio: in fase di test della centralina di comando del motore di un veicolo può essere necessario avere a disposizione informazioni come la velocità di rotazione del motore e la velocità del veicolo; usando un semplice analizzatore di bus CAN le informazioni visualizzate sarebbero una poco significativa sequenza di data byte:

```
CAN ID  DLC  D0  D1  D2  D3  D4  D5  D6  D7
0x618    8   00  40  23  FF  00  00  00  00
```

Da questi dati grezzi l'operatore non è immediatamente in grado di ricavare le informazioni di cui ha bisogno, in questo caso supponiamo che il byte in D1 rappresenti la velocità del veicolo ed i byte D2 e D3 la velocità di rotazione del motore. Per ricavare le due quantità fisicamente significative il byte D1 deve essere prima convertito in decimale (= 64), moltiplicato per 2 (= 128) e gli va sommato −40(= 88); il valore ottenuto costituisce la velocità espressa nell'unità di riferimento scelta e cioè:

```
Velocità del veicolo = 88Km/h
```

Nello stesso modo la velocità del motore è ottenuta dai byte D2 e D3 interpretati come un intero a 16 bit. Impostando D2 come il byte MSB e D3 come quello meno significativo (LSB) il dato da elaborare sarà fatto così:

```
MSB       LSB
00100011  11111111  =  0010001111111111
```

e corrisponde al valore decimale 9215. In questo particolare caso l'operatore imposta i dati del segnale in modo tale da ricavare la velocità dividendo il valore decimale per 2 (moltiplicandolo per 0.5) e sommando un offset di +512; il valore espresso in RPM (Revolutions Per Minute) ottenuto è il seguente:

```
Velocità motore = 5120 RPM
```

La finestra contenente la lista dei segnali riporta questi valori e può essere ridimensionata così come le sue colonne. Un doppio-click del mouse su una delle righe permette all'utente di editare il segnale selezionato ed accedere al database per apportare qualunque variazione ai dati necessari per la conversione e visualizzazione. I parametri che devono essere specificati sono i seguenti:

1. **Description**: il campo è utilizzato per contenere un commento al segnale descrittivo dello stesso ad uso e consumo dell'utente, ad esempio: "Messaggio riservato" o "Segnale derivante da....

2. **Signal Name**: il nome assegnato al segnale, come "Engine Speed" o "Temperatura Ambiente".

3. **Unità di Misura**: l'unità di misura della quantità fisica ricavata a seguito della conversione, ad esempio "Km/h"o "RPM".

4. **CAN Identifier**: il messaggio da utilizzare per la conversione è identificato da un ID, ogni segnale deve avere associato un messaggio e quindi un identificatore a 11 o 29 bit che deve essere specificato in questa Edit-box.

5. **Standard ID / Extended ID**: specifica se l'identificatore è standard o esteso.

6. **Color**: ad ogni segnale può essere associato un colore che viene utilizzato per tracciarne il grafico nella "CCanGraphView".

7. **Format Type**: Questo campo definisce in quale modo i bit del campo DATA vengono organizzati per estrarre il segnale. Selezionando una delle due alternative proposte si imposta il bit MSB o LSB e quindi l'ordine con cui i bit vengono organizzati (vedi appendice F).

8. **Start Bit**: specifica la posizione nel campo DATA (composto da 64 bit) da cui CAN*software* parte ad acquisire i "Lenght" bit con cui costruire il segnale.

9. **Lenght**: specifica il numero di bit da prelevare.

10. **Data Type**: specifica come deve essere interpretata la sequenza dei "Lenght" bit. E' possibile effettuare una scelta tra: interi senza segno

(Unsigned), interi segnati (Signed) o in virgola mobile (Float[36]). Per maggiori dettagli si veda appendice F.

11. **Mul. Factor**: specifica il fattore per il quale viene moltiplicato il dato estratto dal messaggio per la conversione nella grandezza fisica desiderata.

12. **Offset**: definisce una costante che viene sommata al dato grezzo per ottenere il dato reale.

13. **Signal Type**: il valore ottenuto applicando la formula di conversione impostata attraverso i precedenti parametri può essere già la grandezza fisica (Analog) oppure essere trasformata in un segnale digitale (Digital).

14. **Threshold**: consente di fissare la soglia (High) al di sopra della quale il valore derivante dalla conversione viene interpretato come valore alto e la soglia (Low) al di sotto della quale il segnale è a livello logico basso.

15. **Allarms**: se il segnale è analogico l'operatore può impostare degli allarmi su un valore massimo e/o un valore minimo al superamento dei quali viene avvertito visivamente (cambia il calore in sfondo al campo "Value").

Infine sono presenti i pulsanti che consentono la gestione del database:

- con il pulsanti "Next" e "Prev" si scorre il database passando dalla visualizzazione di un segnale ad un'altro. Se si stà definendo un nuovo segnale la pressione del taso "Next" determina il suo inserimento nel database corrente.

- Il pulsante "Delete" cancella dal database il segnale attualmente visualizzato;

- "Delete All" cancella tutti i segnali;

- "Save" salva in un file di estensione ".SIG" l'elenco dei segnali appartenenti al database attuale, in questa maniera l'utente non deve ridefinire i segnali ad ogni nuovo utilizzo dell'analizzatore;

- "Load" consente di caricare un vecchio database di segnali precedentemente salvati in un file ".SIG". L'operatore può crearsi tutti i database che vuole per utilizzarli di volta in volta in reti differenti.

[36]La codifica *floating point* è stata implementata nel rispetto dello standard IEEE su 4 byte [8].

- Per la costruzione di un nuovo insieme di segnali partendo da database già costruiti e magari di grosse dimensioni tramite il pulsante "Append" è possibile agevolare questa operazione.

- "OK" chiude la finestra e nel caso in cui il file corrente sia stato modificato viene aggiornato, se il file non esiste ne viene creato uno nuovo e l'applicazione chiede il nome da assegnare al file e la directory nel quale memorizzarlo.

5.2.4.4 Parametri di funzionamento dell'applicazione

Per adattare alcune particolari funzionalità di CAN*software* alle specifiche esigenze di ogni singolo utente nella finestra "Parameters Dialog" che viene visualizzata selezionando la voce "Parameters" dal menu "setup", è possibile configurare alcuni parametri che sono di seguito descritti.

1. **Number of Samples**: specifica il numero di messaggi che CAN*software* utilizza per calcolare il parametro che esprime in termini percentuali l'occupazione del bus e che viene denominato "WorkLoad". Il metodo utilizzato per il calcolo di questo fattore è riportato in appendice G.

2. **Factor Delay**: quando il sistema lavora in modalità off-line e viene abilitata la funzione di "Playback" questo parametro viene utilizzato dall'applicazione come fattore di ritardo nella visualizzazione dei messaggi. Impostando questo parametro ad un valore superiore ad 1 l'utente ottiene uno "slow-motion" dei dati.

3. **Point Resolution**: questo coefficiente specifica il numero di valori del segnale che vengono rappresentati nella finestra grafica gestita dalla vista "CCanGraphView"; costituisce una specie di risoluzione orizzontale del grafico.

5.2.5 Menu: "action"

Nel drop-down menu visualizzato selezionando la voce "Action" si concentra l'operatività dell'analizzatore. Il sottomenu è suddiviso in due sezioni: la prima che include i comandi "Start Listen" e "Stop Listen" consentono di avviare o fermare l'acquisizione dati, la seconda fornisce le funzioni per la gestione della modalità di "Playback".

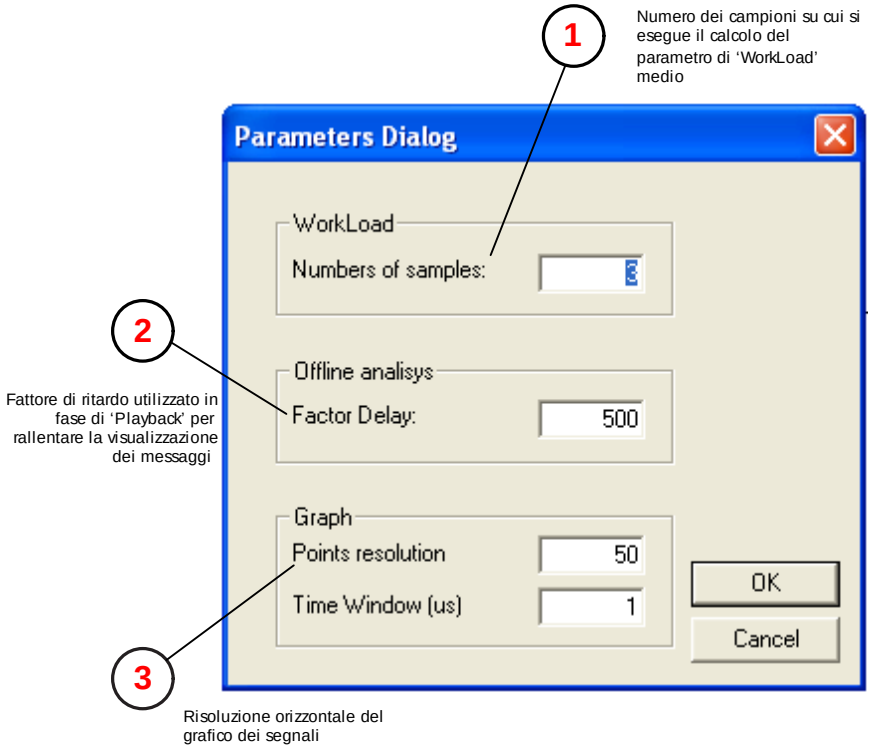


Figura 5.31: CAN*software*: finestra di impostazione dei parametri.

5.2.5.1 Modalità di "Listening"

Selezionando "Start Listen" CAN*software* come prima cosa verifica che l'interfaccia dell'analizzatore sia connessa al PC, superato il controllo e programmato l'hardware i dati ricevuti vengono organizzati e visualizzati nelle quattro finestre disponibili. L'acquisizione termina selezionando la voce "Stop Listen"; avendo implementato l'applicazione come una "Single Document Interface" una nuova connessione richiede la generazione di un nuovo documento. Premuto il pulsante di stop i dati possono essere salvati in un file e/o esportati in un formato compatibile con programmi di elaborazione dedicati e cioè fogli di calcolo come *Microsoft Windows EXCEL* ©.

5.2.5.2 Modalità di "Playback"

Questa modalità consente all'operatore di selezionare un file con estensione ".cdt" ottenuto da una precedente acquisizione e ne riproduce il contenuto simulando il transito dei messaggi sul bus in slow-motion.

In partenza la velocità di riproduzione è fissata dal parametro "Factor Delay" fissato durante il setup dei parametri generali; la riprodutzione può essere terminata selezionando il comando di "Stop", sospesa con "Pause", accelerata o rallentata ulteriormente con "Fast" e "Slow".

Dal lato implementativo si è avuta l'esigenza di generare un riferimento di tempo assoluto ed anziché impiegare le API: "SetTimer" e "KillTimer" con cui si crea, imposta e chiude una risorsa timer si è sfruttato un *high-resolution performance counter*. La funzione "SetTimer" consente di generare un messaggio di tipo "WM_TIMER" ad intervalli di tempo minimi di 55ms che può essere intercettato ed utilizzato per gestire il playback dei dati. Il problema di utilizzare questa tecnica è che il messaggio come qualunque altro finisce nella coda dei messaggi del sistema operativo e quindi non è affidabile come riferimento temporale. La scelta del contatore ad elevate performace è da un lato limitativa perché l'hardware deve essere predisposto, dall'altro è molto più efficiente.

5.2.6 Menu: "file"

Da questo menu è possibile accedere a tutte le tipiche funzioni di gestione dei file e di stampa presenti nelle applicazioni Windows di tipo SDI e MDI.

Ad eccezione della stampa non implementata si ritrovano le classiche funzioni *Apri, Salva* e *Salva con nome* a cui si aggiungono i comandi di esportazione.

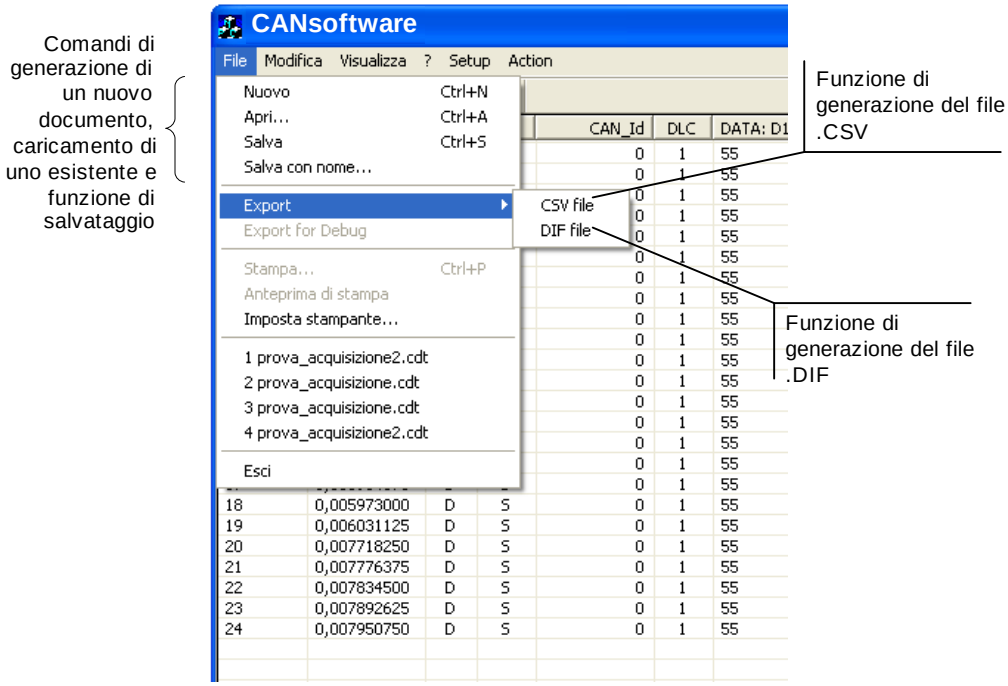


Figura 5.32: CAN*software*: menu "file".

5.2.6.1 Salvataggio dei dati

Per default il dati vengono salvati in file con estensione ".cdt" (CAN DaTa); l'uso di questa estensione non è ovviamente obbligatoria semplifica però l'operazione di apertura del file che può essere ottenuta con un double-click sulla sua icona ed in fase di apertura è l'estensione utilizzata di default per fare una scrematura dei file che vengono visualizzati nella finestra *Apri*.

5.2.6.2 Esportazione dei dati

Per consentire la manipolazione e l'analisi dei dati acquisiti sfruttando applicativi più potenti nati per questi scopi, CAN*software* permette l'esportazione in due diversi formati[37].

CSV file : CSV significa "Comma Separated Values" (valori separati da virgola); si tratta di un formato solo testo che serve a memorizzare dati destinati ad essere impiegati nei fogli di calcolo e nei database software. Questo file può essere aperto direttamente da applicazioni come *Microsoft Windows EXCEL* ©.

[37]Per approfondimenti sulla struttura dei file generati dalle procedure di esportazione si rimanda all'appendice H.

DIF file : Il "Data Interchange Format" è un formato che ha caratteristiche duali al CSV e cioè: la sua struttura è certamente più complessa della precedente, rimane un file di testo ma contiene informazioni sulla formattazione dei campi; per queste ragioni si tratta del formato più adatto in cui esportare i dati se si ha l'intenzione di rielaborarli per mezzo di Excel o altro foglio di calcolo. In figura 5.33 è riportato un esempio di un file ".dif" generato da CAN*software* ed aperto con *Microsoft Windows EXCEL* ©.

5.2.7 La barra di stato

CAN*software* riepiloga nella barra di stato i parametri di setting dell'interfaccia attualmente impostati ed altre utili informazioni.

1. Riporta lo stato dell'applicazione: *Pronto* (disponibile per nuovi comandi), *Exporting* (in fase di esportazione dati) ecc...

2. E' una barra di progressione per mezzo della quale l'operatore può avere un'idea della durata dell'operazione richiesta.

3. Baudrate della rete. Viene visualizzato il valore impostato dall'operatore nella finestra "Bit timing Dialog". Se l'utente sfrutta la funzionalità di autobaud CAN*software* calcola il baudrate della rete dai parametri di bit timing trasmessi dall'interfaccia e visualizza il dato così calcolato nella barra di stato; in definitiva l'utente ha la possibilità di rilevare automaticamente il baudrate di una rete CAN con caratteristiche sconosciute.

4. Sono riepilogati i parametri di bit timing espressi in termini dei valori con cui vengono programmati i registri CANBT del micro: *BRP, PRS, PHS1, PHS2, SJW*.

5. In questo campo della barra di stato si riepilogano i parametri con cui viene programmata la porta di comunicazione: *PORTA, BAUDRATE*.

6. Sono riportati identificatore (*ID*) e maschera (*MSK*) impiegati per programmare il filtro hardware dei messaggi sul bus CAN.

7. In questo campo si riepilogano gli altri parametri utilizzati per filtrare i dati: *ID std/ext, Data/Remote frame, Stamp Enable/Disable, Message Enable/Disable*. In tabella 5.7 si specificano i possibili valori per gli elementi che compongono questo campo.

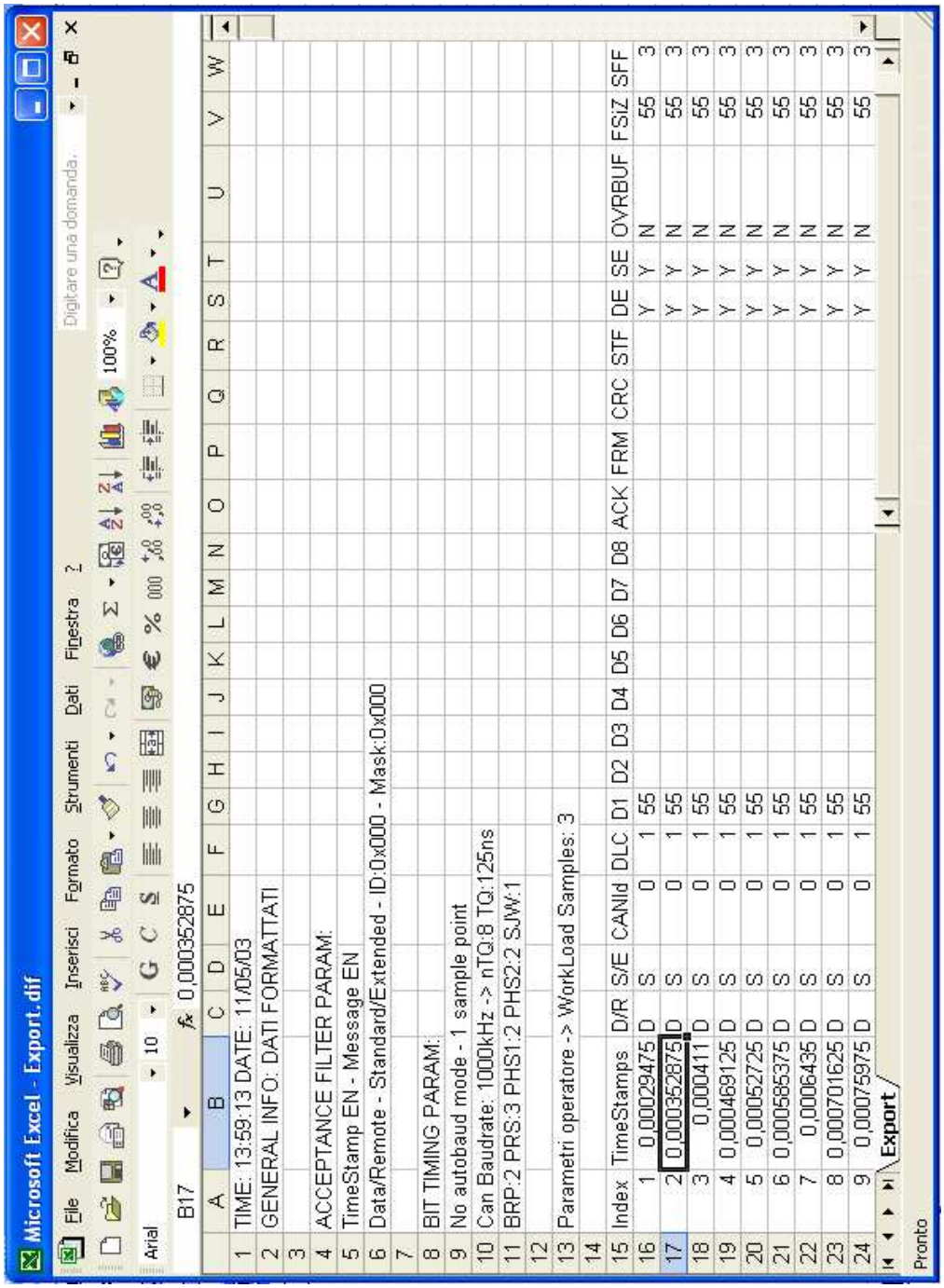


Figura 5.33: CAN*software*: esempio di un file ".dif" generato da CAN*software* ed aperto con *Microsoft Windows EXCEL* ©.

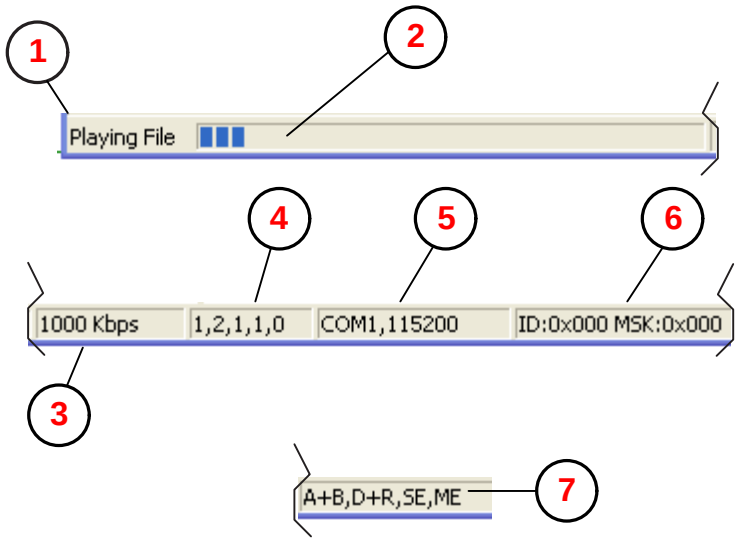


Figura 5.34: CAN*software*: barra di stato dell'applicazione.

Descrizione		Mnemonico
Tipo di ID	solo ID standard	A
	solo ID esteso	B
	tutti	A+B
Tipo di Frame	solo DATA frame	D
	solo REMOTE frame	R
	tutti	D+R
Time stamp	abilitato	SE
	disabilitato	SD
Message	abilitato	ME
	disabilitato	MD

Tabella 5.7: Tabella riepilogativa valori campo parametri.

Bibliografia

[1] TEMIC Semiconductors. *Hardware Description of the C51 Family Products*, 1997.

[2] ATMEL Corporation. *Frequently Asked Questions (FAQ) on T89C51CC01 CAN microcontrollers*, 2000. www.atmel.com.

[3] Robert Bosch GmbH. Fault confinement. In *CAN Specification version 2.0*, pages 24–25, 1991. www.can.bosch.com.

[4] Keil Elektronik Gmbh e Keil Software Inc. *Getting Stanted with uVision2*, 2001. www.keil.com.

[5] ATMEL Corporation. *Atmel T89C51CC01 Datasheet*, 2002. www.atmel.com.

[6] Tim Tompkins Jon Bates. *Visual C++ 6.* McGraw-Hill, first edition, 1998.

[7] Allen Denver. Serial communications in win32. *MSDN*, 1995. Microsoft Windows Developer Support.

[8] Microsoft Coorporation. IEEE floating-point representation and microsoft languages. MSDN.

Capitolo 6

Conclusioni

L'aspetto certamente più rilevante di questo progetto è quello di aver realizzato un oggetto completamente funzionante che può essere impiegato in tutte le situazioni ove vi sia l'esigenza di analizzare il traffico di una rete CAN. Alla "grezza" capacità di acquisire e visualizzare i dati che transitano nella rete, CAN*analyser* implementa alcune funzionalità che aiutano l'operatore nel suo lavoro; prima però di descrivere prestazioni e limiti vediamo in che maniera sono stati realizzati i test per verificare il corretto funzionamento dello strumento.

6.1 Test dell'analizzatore

I test sono stati condotti sulla rete descritta nel paragrafo 3.2.3. Una prima serie di verifiche sono state eseguite collegando i due nodi realizzati per mezzo delle demoboard ATMEL a due interfacce di codifica audio disponibili presso il Dipartimento e sviluppate appositamente per testare la funzionalità del "Voice Over CAN" cioè la trasmissione di un segnale audio intelleggibile attraverso una rete di questo genere. Nel caso descritto le due demoboard funzionano da interfacce di collegamento della scheda di codifica audio e il bus CAN: la scheda invia tramite seriale i byte che sintetizzano il messaggio sonoro alle demoboard su cui è presente un opportuno software che attende l'arrivo di un sufficiente numero di byte e poi invia il burst di frame sulla rete, la stessa demoboard rimane anche in attesa di ricevere i byte in arrivo dall'altra interfaccia che simula il secondo interlocutore del sistema (entrambe le schede audio dispongono di un microfono e di un auricolare). Le due interfacce trasmettono solo DATA Frame con DLC = 8 (sfruttano l'intero campo DATA disponibile), ad ognuna di esse è associato un unico identifica-

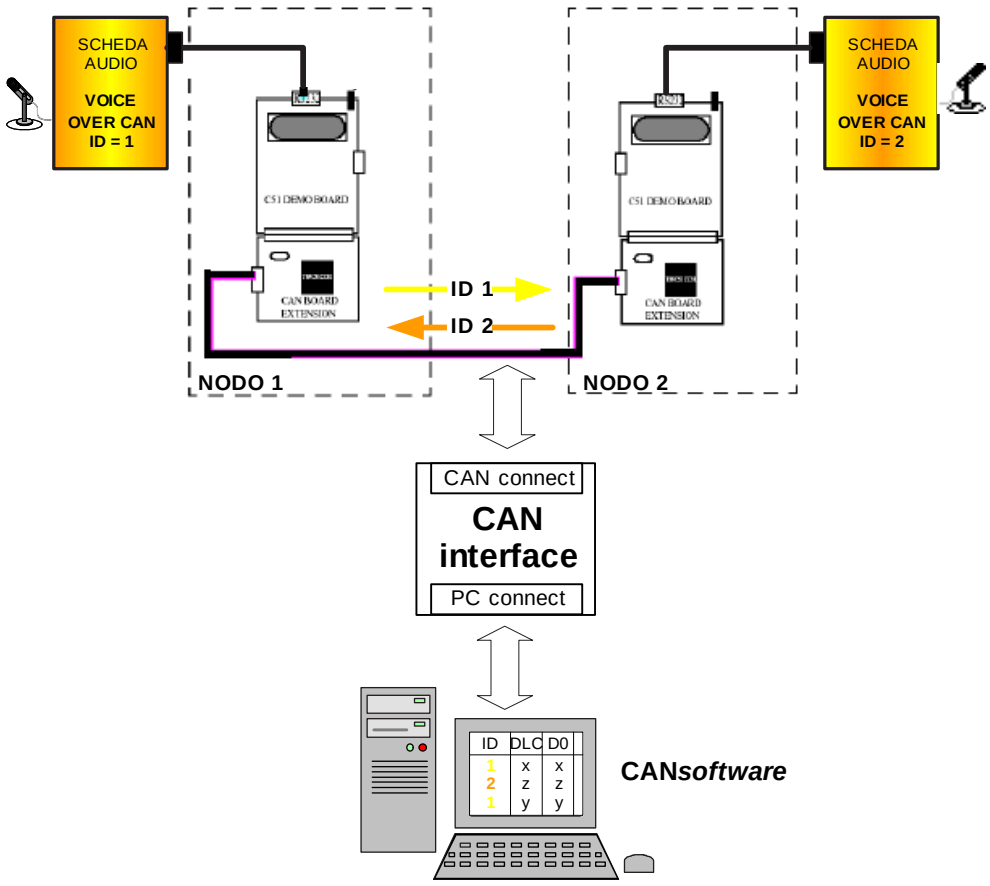


Figura 6.1: Rete CAN realizzata per il test del sistema.

tore con cui vengono trasmessi tutti i messaggi: nel nostro caso il canale di sinistra era stato programmato con ID = 1, quello di destra con ID = 2.

La verifica condotta su questo sistema del tutto funzionante è quindi una verifica completamente attendibile. CAN*analyser* come si evidenzia dal file testo generato dall'esportazione dei dati acquisiti è stato in grado di intercettare questi messaggi e riconoscere l'identificatore, il tipo di frame e la dimensione del campo DATA (contenuto di DLC):

```
Index;TimeStamps;D/R;S/E;CANId;DLC;D1;D2;D3;D4;D5;D6;D7;D8;ACK..
1;0,000194625;D;S;1;8;8b;9;80;0;82;3;84;1;;;;;Y;Y
2;0,001081500;D;S;2;8;0;18;9;92;1a;81;98;0;;;;;Y;Y
3;0,001200375;D;S;1;8;80;1;88;1;89;1;8c;2;;;;;Y;Y
4;0,002080500;D;S;2;8;18;a8;0;98;10;88;80;28;;;;;Y;Y
5;0,002199375;D;S;1;8;88;9;8a;0;88;0;81;1;;;;;Y;Y
6;0,003081375;D;S;2;8;81;8;a8;0;98;88;80;88;;;;;Y;Y
7;0,003201375;D;S;1;8;88;0;8c;81;c;80;9;80;;;;;Y;Y
8;0,004076625;D;S;2;8;10;88;90;10;18;90;89;11;;;;;Y;Y
9;0,004197750;D;S;1;8;8;80;1;80;1;89;1;81;;;;;Y;Y
10;0,005077500;D;S;2;8;88;88;80;a8;1;89;1;9;;;;;Y;Y
11;0,005196750;D;S;1;8;c;82;c;88;18;88;1;8b;;;;;Y;Y
12;0,006075750;D;S;2;8;91;8;88;0;89;11;29;91;;;;;Y;Y
13;0,006196875;D;S;1;8;80;80;0;80;1;80;2;88;;;;;Y;Y
14;0,007085625;D;S;2;8;0;88;10;99;81;89;80;0;;;;;Y;Y
```

Qualche dubbio sull'esattezza dei dati acquisiti può esserci relativamente al contenuto dei campi D0..D7 (campo DATA del frame) e sul valore di "TimeStamps" generato. Il contenuto del campo DATA non è valutabile a priori essendo generato dall'algoritmo di codifica audio sulla base del segnale acquisito attraverso il microfono, non è quindi possibile verificare se i dati sopra riportati sono effettivamente quelli inviati attraverso la rete, in più i frame vengono inviati dal software che gira sulle due demoboard con una cadenza tale che in circa 1ms devono essere trasmessi due messaggi (uno dal nodo destro ed uno da quello sinistro), come possiamo notare lo stamping rispetta questa condizione, i messaggi con identificatore 1 e 2 si alternano sequenzialmente nell'intervallo di tempo richiesto ma anche in questo caso non c'è modo di verificare l'esatta attendibilità dei dati temporali elaborati. Allo scopo di risolvere questi "dubbi' si è approntato un test più selettivo che ci garantisce sul contenuto e sulla cadenza con la quale i frame transitano sul bus.

La seconda serie di test è stata approntata creando per uno dei due nodi della nostra rete un applicativo ad hoc che consente di:

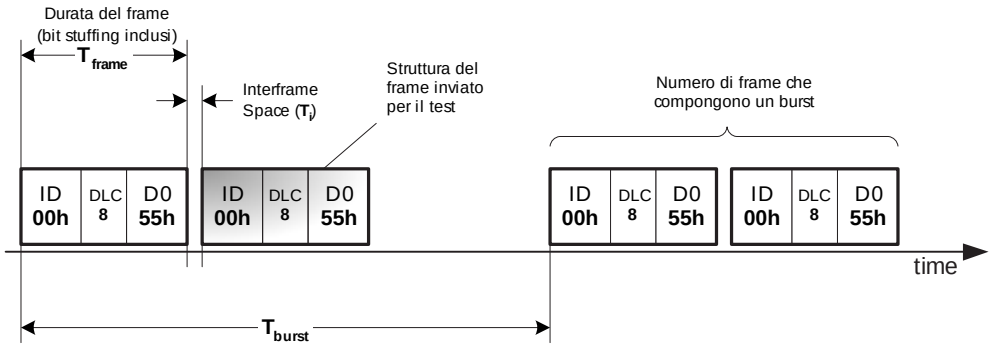


Figura 6.2: Schema di generazione dei burst utilizzati per il test.

- inviare DATA Frame con identificatore fisso: ID = 00h;
- i DATA Frame hanno un campo DATA noto in particolar modo:

$$\text{DLC} = 1,\ \text{D0} = 55\text{h} = 10101010\text{b}$$

- invia questi messaggi in forma di burst cioè di sequenze di frame che hanno una cadenza temporale programmabile per mezzo dei seguenti due parametri:

  1. l'intervallo di tempo con cui si ripete l'invio dei burst;
  2. il numero di frame che compongono la sequenza inviata.

Programmando una delle due demoboard in questa maniera e lasciando all'altra il compito di fornire l'ack di convalida al messaggio (cosa che CAN*interface* non può dare quando è in fase di acquisizione dati) la sequenza di frame che transitano sulla nostra rete di test è rappresentata in figura 6.2. Unico elemento che non è completamente determinato è la durata dell'"Interframe Space" e cioè dell'intervallo di tempo che intercorre tra l'invio di due frame dello stesso burst ma visto che si è implementata una rete con un solo nodo trasmettitore e che immediatamente dopo la fine di un frame ne viene inviato un altro (ammesso che non sia l'ultimo del burst) ci si può aspettare che questo tempo indicato con T_i sia molto prossimo al suo valore minimo pari a 3 *bit time*. La figura 6.3 schematizza la rete implementata per questa seconda serie di test mentre in figura 6.4 viene mostrato lo screenshot del programma durante l'acquisizione dei dati.

Come è possibile osservare anche in questo secondo test: l'ID, il tipo di frame (DATA con ID standard) ed il campo DLC rispettano le caratteristiche dei messaggi CAN che transitano sulla rete, in questo caso possiamo avere

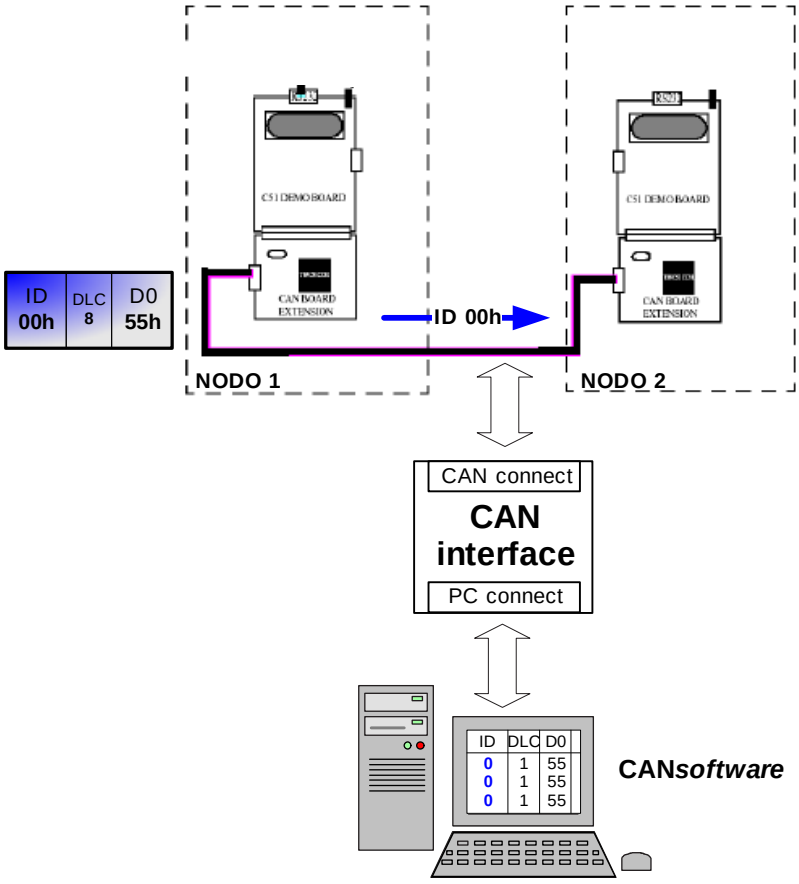
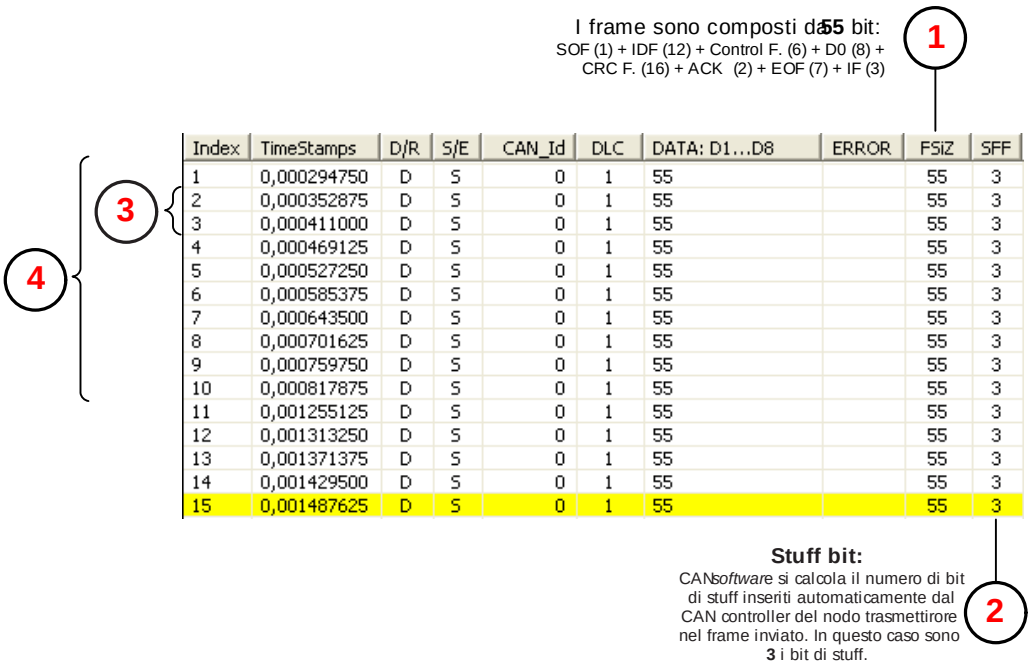


Figura 6.3: Rete CAN realizzata per la seconda serie di test del sistema.



Index	TimeStamps	D/R	S/E	CAN_Id	DLC	DATA: D1...D8	ERROR	FSiZ	SFF
1	0,000294750	D	S	0	1	55		55	3
2	0,000352875	D	S	0	1	55		55	3
3	0,000411000	D	S	0	1	55		55	3
4	0,000469125	D	S	0	1	55		55	3
5	0,000527250	D	S	0	1	55		55	3
6	0,000585375	D	S	0	1	55		55	3
7	0,000643500	D	S	0	1	55		55	3
8	0,000701625	D	S	0	1	55		55	3
9	0,000759750	D	S	0	1	55		55	3
10	0,000817875	D	S	0	1	55		55	3
11	0,001255125	D	S	0	1	55		55	3
12	0,001313250	D	S	0	1	55		55	3
13	0,001371375	D	S	0	1	55		55	3
14	0,001429500	D	S	0	1	55		55	3
15	0,001487625	D	S	0	1	55		55	3

Figura 6.4: Visualizzazione dei dati acquisiti durante una delle prove della seconda serie di test.

anche riscontro dell'esattezza dei campi su cui in precedenza non si era potuta svolgere alcuna verifica. I dati si riferiscono ad una rete CAN impostata con un baudrate di 1Mbit/s, tenendo presente ciò si può notare che:

1. CAN*analyser* ha correttamente rilevato il numero di bit (compresi i 3 *bit time* che costituiscono l'"Intermission Frame") che compongono il messaggio CAN destuffato (campo "FSiZ - Frame SiZe);

2. CAN*analyser* si è calcolato la sequenza CRC, ha ricostruito la sequenza di bit che compone il frame destuffato e su di essa ha calcolato il numero di stuff bit che il nodo trasmittente ha inserito; in questo caso i bit di stuff (campo "SFF") sono 3. *Il valore rilevato è stato calcolato anche manualmente verificando l'esattezza del dato.*

3. In totale i bit che compongono il frame fisicamente trasmesso sul bus sono 58 (55+3) e quindi:

$$T_{frame} = 58bit \text{ x } 1\mu s = 58\mu s$$

incluso i 3 *bit time* dell'"Intermission Frame". Confrontando i tempi di arrivo dei messaggi che si possono ricavare dalla colonna "Time-

Stamps", con riferimento ai frame 2 e 3 di figura si ricava che:

$$T_{frame} = 411.000ms - 352.875ms = 58.125\mu\text{s}$$

quello indicato è la durata del frame stimata da CAN*analyser*. Il valore ricavato è estremamente vicino a quello teorico, la minima imprecisione è dovuta a due fattori:

- come già detto T_i è di difficile valutazione in modo particolare visto che con l'invio del bit di "Start Of Frame" avviene anche la sincronizzazione dei nodi per mezzo della quale vengono recuperati gli errori di fase;
- bisogna anche tenere presente che l'unità temporale minima di cui CAN*interface* dispone per costruire il riferimento temporale utilizzato per lo stamping corrisponde al periodo del segnale di clock che pilota il CAN timer e che vale 0.375μs; al di sotto di questa risoluzione il nostro sistema non può andare.

Si può quindi affermare che il valore di T_{frame} che si ottiene dallo strumento è la migliore approssimazione possibile del valore reale.

4. Il nodo è stato programmato per generare dei burst composti da 10 frame che si ripetono ogni 1.255ms. Confrontando il valore contenuto nei campi "TimeStamps" dei frame 11 e 1 si trova un valore che approssima con il solito errore (descritto sopra) quello teorico impostato.

Questo tipo di test è stato condotto ripetutamente variando non solo i parametri di generazione dei burst (numero di frame e intervallo di generazione) ma anche testando la rete a differenti baudrate e sfruttando la funzionalità di autobaud. Il test di questa funzione di hot-plugging è stato eseguito verificando il valore di baudrate che CAN*analyser* calcola sulla base dei parametri di bit timing che CAN*interface* gli fornisce dopo che è stata in grado di "agganciarsi" correttamente al bus; il baudrate così calcolato viene visualizzato nella barra di stato del programma (vedi anche figura 6.5). La possibilità di confrontare il valore di baudrate calcolato con quello impostato sulla rete riprogrammando ogni volta i nodi 1 e 2 ha consentito di verificare il corretto funzionamento dimostrato anche dal fatto che i frame visualizzati mantengono la struttura che ci si aspetta.

6.2 Performance

CAN*analyser* nella versione attuale implementa le funzionalità di seguito elencate.

- Perfettamente compatibile con lo standard ISO 11898-2.
- Gestisce reti CAN alla massima velocità di 1Mbit/s.
- L'interfaccia si collega al PC attraverso la porta seriale standard RS232, il sistema può dunque essere utilizzato praticamente su qualunque "macchina" visto che tutti i personal dispongono di almeno una porta COM.
- Il software di visualizzazione ed analisi CAN*software* è stato sviluppato per piattaforma *Microsoft Windows* © e funziona su i sistemi operatitivi Windows 95/98/98me/2000/XP.
- L'interfaccia CAN*interface* nasce per essere alimentata da una batteria a 12V come quelle che si trovano nelle autovetture e integra un sistema di protezione per prevenire l'inversione di polarità dei collegamenti.
- Supporta la gestione di un unico canale di tipo "2.0B ATTIVO" è quindi in grado di lavorare con reti CAN nelle quali siano contemporaneamente presenti messaggi con ID *standard* ed ID *estesi*.
- Per ogni messaggio è in grado di fornire informazioni di time stamping con una risoluzione di 0.375ns.
- Tutti i parametri di bit timing possono essere manualmente impostati dall'operatore adattando CAN*analyser* a qualunque situazione reale di impiego: "Prescaler", "Propagation Delay", "Phase segment 1 e 2", "Resynchronization segment". CAN*software* in funzione del baudrate impostato calcola tutte le combinazioni valide dei parametri e limita la scelta che può fare l'utente alla selezione di una combinazione valida tra quelle proposte eliminando ogni rischio di errore.
- Supporta la funzionalità di triplice campionamento del bus che consente di sfruttare l'analizzatore anche in condizioni di segnali fortemente disturbati o in presenza di forte attenuazione.
- Implementa la funzione di "hot-plugging" o di "autobaud": l'analizzatore può essere collegato ad una rete in funzione e senza conoscerne il baudrate, il baudrate verrà determinato automaticamente con una tecnica denominata "Good Configuration Find" GCF.
- Esegue un'operazione di filtro sul traffico dando la possibilità all'operatore di selezionare:
  - il tipo di frame: "DATA", "REMOTE", "ID standard" (2.0A) e "ID extended(2.0B);

  - l'identificatore del messaggio;
  - un gruppo di identificatori validi specificando un ID di base ed una maschera opportuna.

- L'operatore può scegliere di visualizzare o meno il campo DATA dei frame e le informazioni di stamping, ciò consente di sfruttare il canale di comunicazione tra interfaccia CAN e PC nel modo più adeguato alle proprie esigenze: se la rete CAN ha un elevato baudrate ed è fortemente caricata (elevato "bus load") c'è un rischio elevato di perdere informazioni e questo può essere ridotto sfruttando il canale di comunicazione solo per i dati strettamente necessari all'analisi da condurre.

- Fornisce informazioni di vario genere in modo particolare il numero di messaggi acquisiti e due parametri che costituiscono una stima dell'occupazione di banda in termini percentuali detti parametri di "WorkLoad".

- Ad ogni messaggio può essere associato un parametro temporale corrispondente all'istante di transito del frame sul bus; si tratta di un riferimento temporale assoluto.

- Visualizza per ogni messaggio:

  - indice progressivo;
  - istante di arrivo (opzionale);
  - tipo di frame;
  - identificatore;
  - dimensione del campo DATA;
  - contenuto del campo DATA (opzionale);
  - le condizioni di errore descritte nel protocollo;
  - il numero di bit componenti il frame destaffato;
  - il numero di bit di stuff da associare al frame.

- Il verificarsi di una condizione anomala come l'overflow e quindi la perdita di messaggi che ne deriva o gli errori nella struttura del frame trasmesso sulla rete, viene notificato all'operatore colorando opportunamente i frame interessati da uno degli eventi descritti.

- E' possibile costruire un database di "segnali" o più correttamente di grandezze fisiche il cui valore viene calcolato sulla base del contenuto

del campo DATA dei frame arrivati. CAN*analyser* in questa maniera fornisce una funzionalità di alto livello che aiuta l'operatore nella analisi dei dati. Per ogni segnale possono essere specificati parametri descrittivi come:

- il nome;
- una descrizione (opzionale);
- l'unità di misura (opzionale);
- un colore (default);
- tipo di segnale;
- soglia per i segnali digitali (solo per digitali);
- soglie di allarme (opzionale);

parametri identificativi:

- l'identificatore CAN;
- il tipo di identificatore;

ed i parametri necessari alla elaborazione del suo valore e cioè:

- tipo di formato dei bit;
- sequenza dei bit su cui eseguire la conversione;
- metodo di interpretazione dei dati;
- fattore moltiplicativo;
- fattore di offset.

- Il valore dei segnali viene rappresentato graficamente e testualmente, la visualizzazione grafica può essere attivata /disattivata per ogni singolo segnale.
- La condizione di superamento delle soglie di allarme da parte dei segnali analogici viene notificata all'operatore.

6.3 Limiti

Il fattore limitante principale delle prestazioni di CAN*analyser* è certamente rappresentato dalla limitata banda del canale di comunicazione tra PC ed interfaccia. Come evidenziato nel paragrafo 5.1.2.3 l'effetto di questo limite è la perdita parziale di informazioni cioè non tutti i frame che transitano

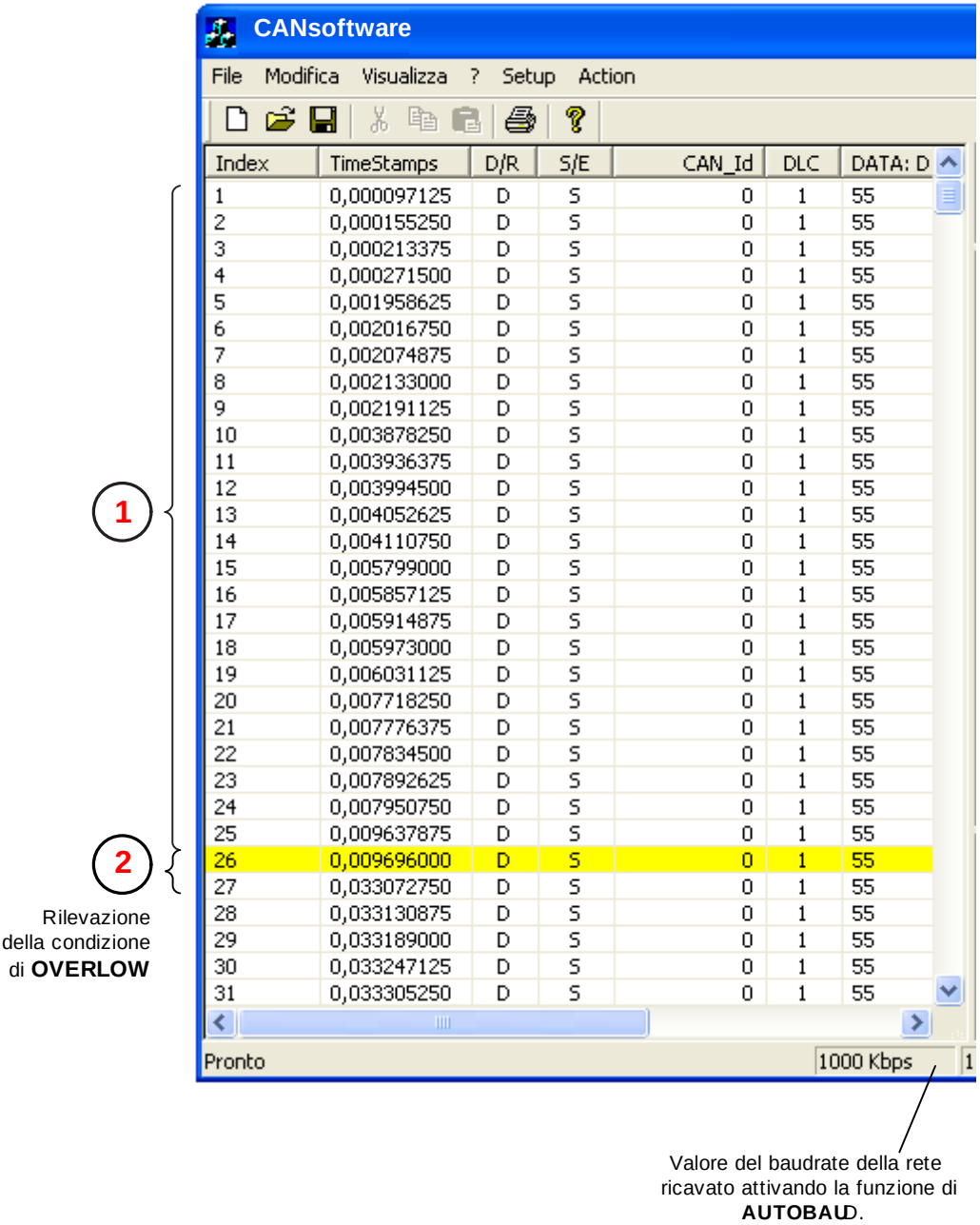


Figura 6.5: Visualizzazione della condizione di "overflow".

sul bus CAN finiscono con l'essere acquisiti dall'analizzatore. Ad esempio i dati visualizzati in figura 6.5 e ricavati da uno dei test eseguiti, mostrano come l'interfaccia CAN*interface* rimanga scollegata dal bus per molto tempo, tempo richiesto per la reinizializzazione completa del sistema avvenuta la quale l'acquisizione riparte. Per tutta la durata della fase di reinizializzazione i frame che transitano sul bus non vengono intercettati e quindi sono disponibili.

Nella seconda parte del capitolo dedicato all'interfaccia CAN*interface* (4.2) si è fornita una soluzione hardware a questo grosso problema di banda che prevede di sostituire oppure utilizzare in parallelo un canale di connessione USB la cui banda passante può, nel caso mostrato, arrivare a 1Mbit/s e cioè circa 10 volte maggiore della massima banda passante della seriale RS232 del Personal Computer.

Altri limiti possono essere identificati nelle differenze riscontrate con altri strumenti dello stesso genere ed in particolar modo con quelli descritti in 2.4. L'implementazione delle funzionalità mancanti può essere realizzata adattando il codice attuale di CAN*software* e di MICRO*software* alla gestione dei nuovi comandi; sotto questo aspetto l'implementazione hardware, apparte le precedenti considerazioni sulla banda passante a disposizione, non pone particolari limitazioni, la loro realizzazione è stata demandata solo per motivi di tempo. Non sono state implementate le seguenti funzionalità:

- simulazione di nodi virtuali;
- non è possibile trasmette alcun messaggio sulla rete;
- come conseguenza anche della precedente limitazione CAN*analyser* non è in grado di gestire alcun protocollo di alto livello; in ricezione questa possibilità è limitata dalla definizione manuale dei "segnali".
- CAN*interface* dispone di un solo canale, alcune interfacce disponibili in commercio ne implementano almeno due utilizzabili in modo indipendente oppure per realizzare un gateway tra due reti CAN che lavorano a differenti baudrate.
- CAN*interface* disponendo di un solo transceiver è compatibile solo con reti CAN ISO 11898-2; non è possibile la connessione dello strumento a reti CAN di altro tipo come quelle "Single-Wire".

6.4 Note dell'autore

L'elevata dimensione del sorgente di CAN*software* non consente di allegare a questo documento il codice o parti particolarmente rilevanti di esso per

questo motivo il progetto dell'interfaccia Windows realizzato in *Microsoft Windows* © *Visual C++ v.6* © può essere scaricato separatamente da questo documento in formato .zip: "CANSOFTWARE.ZIP".

Il codice di MICRO*software* ripulito di commenti e dei comandi utilizzati in fase di debug, non utili per la comprensione del suo funzionamento è riportato in appendice I, l'intero progetto creato con l'ambiento di sviluppo μVision2 di Keil Software è anch'esso disponibile in download: "MICROSOFTWARE.ZIP".

Appendice A

Configurazione hardware dell'ATMEL T89C51CC01

A.1 Configurazione dei pin

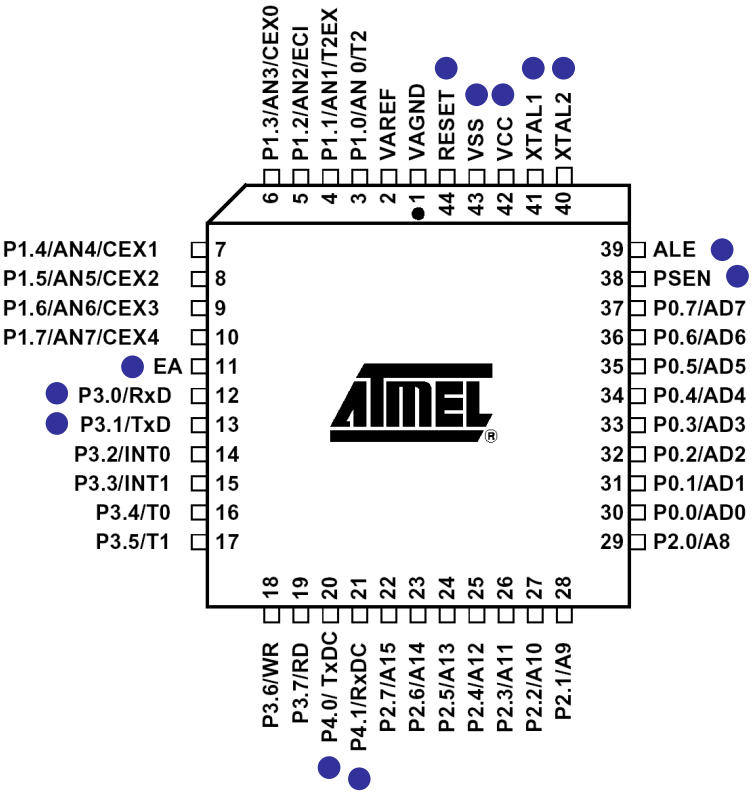


Figura A.1: ATMEL T89C51CC01: configurazione dei pin.

In tabella A.1 sono sinteticamente descritti i segnali associati ai pin del micro limitatamente a quelli utilizzati nella realizzazione dell'interfaccia CAN*interface* e che per comodità abbiamo indicato anche in figura **??**.

A.2 Circuito interno di generazione del clock

In figura A.2 è riportato lo schema del circuito interno di oscillazione sfruttato per la generazione del clock e collocato tra i pin XTAL1(41) e XTAL2(40) del microcontrollore.

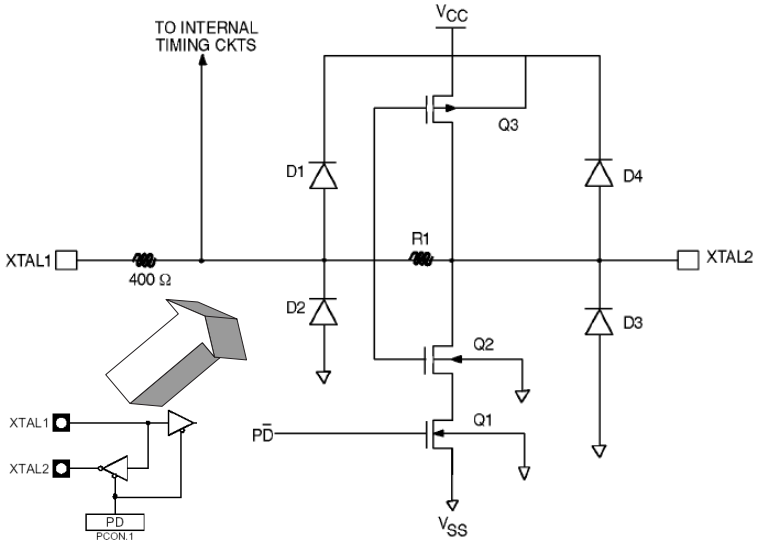


Figura A.2: ATMEL T89C51CC01: parte interna al microcontrollore del circuito di oscillazione per la generazione del clock di sistema.

A.3 Circuito interno di generazione del reset

In figura A.3 è riportato il circuito interno di reset collocato. L'integrazione della resistenza di pull-down consente di limitare ad un semplice condensatore i componenti da collegare esternamente per la corretta gestione del reset del microcontrollore

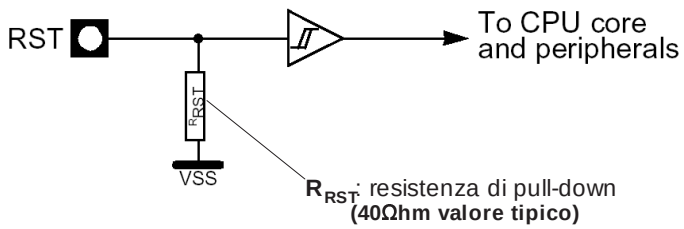


Figura A.3: ATMEL T89C51CC01: parte interna al microcontrollore del circuito di generazione del reset.

Pin Name	Pin Num.	Pin Type	Descrizione
EA	11	I	**$\overline{\text{EA}}$**: Quando il pin External Access è mantenuto a livello alto, le istruzioni vengono prelevate dalla flash memory interna quando il program counter è inferiore a 8000H, viceversa il micro preleva tutte le istruzione dalla memoria di programma esterna.
RxD	12	I/O	**P3.0/RxD**: Ingresso di ricezione dati (asincrono) o ingresso/uscita dati (sincrona) della interfaccia seriale.
TxD	13	O	**P3.1/TxD**: Uscita di trasmissione dati (asincrona) o uscita clock (sincrona) della interfaccia seriale.
TxDC	20	O	**P4.0/TxDC**: Uscita di trasmissione del CAN controller.
RxDC	20	I	**P4.1/RxDC**: Ingresso di ricezione del CAN controller.
PSEN	38	O	**$\overline{\text{PSEN}}$**: Il pin Program Store Enable[1] consente assieme ai pin EA e ALE di far andare in esecuzione il bootloader al reset del sistema e quindi si stabilire uno stato di programmazione del micro o di esecuzione del codice di programma utente.
ALE	39	O	**ALE**: L'Address Latch Enable[2] viene sfruttato per stabilire la "hardware condition" sopra indicata.
XTAL2	40	O	Uscita del circuito di oscillazione.
XTAL1	41	I	Ingresso per il circuito oscillatore esterno ed ingresso del circuito interno di generazione del clock. Se si utilizza una sorgente di clock esterna il pin XTAL2 deve rimanere sconnesso.
VCC	42		Alimentazione.
VSS	43		Massa.
RESET	44	I/O	**Reset**: Un livello alto su questo pin che duri più di cicli macchina resetta il dispositivo. Un pull-down resistivo interno permette il *Power-on reset* usando esternamente una semplice capacità collegata a VCC.

Tabella A.1: ATMEL T89C51CC01: configurazione dei pin.

Bibliografia

[1] ATMEL Corporation. *Atmel T89C51CC01 Datasheet*, 2002. www.atmel.com.

[2] TEMIC Semiconductors. *Hardware Description of the C51 Family Products*, 1997.

Appendice B

ISP e flash memory del T89C51CC01

B.1 FM0 e FM1

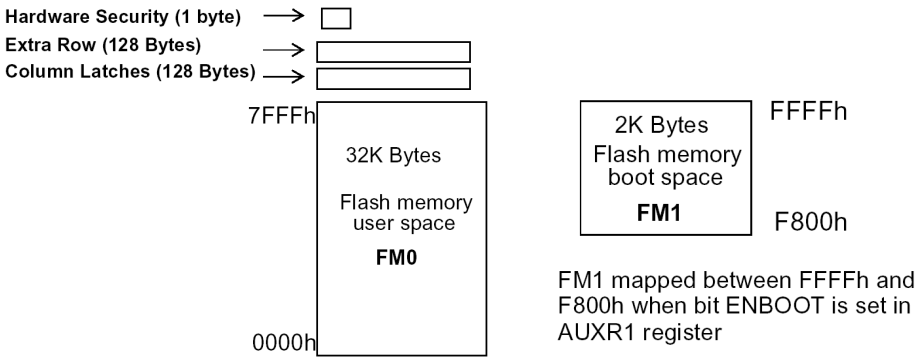


Figura B.1: ATMEL T89C51CC01: organizzazione della memoria flash.

La flash memory presente nel T89C51CC01 è suddivisa in due banchi denominati FM0 e FM1; come si vede dalla figura B.1 l'area di memoria indicata con FM0 è composta da quattro blocchi:

User space : sono 32Kbyte organizzati in 256 pagine di 128 byte ognuna; è destinata a contenere il codice di programma del microcontrollore che viene mandato in esecuzione al reset del sistema quando non viene attivata la funzione di "In-System Programming" o ISP.

Extra Row : ha la dimensione di una pagina; alcuni dei campi in cui è suddivisa sono riservati al produttore altri vengono usati dal bootloader.

Hardware Security Byte : contiene i così detti "Fuse Bit"; solo i 2 bit più significativi sono manipolabili dal software[1] agli altri detti "Lock Bit", si accede solo in modalità di programmazione "parallela[2]" e con le restrizioni del "Security Level" impostate attraverso gli stessi bit nella precedente programmazione.

Column Latches : ha anch'essa la dimensione di una pagina e costituisce il buffer di ingresso per le tre precedenti aree di memoria.

FM1 ha la dimensione di 2Kbyte, contiene il bootloader "standard" e le API necessarie a supportare l'"In-System Programming".

B.2 In-System Programming (ISP)

Con il termine di "bootloader" nel T89C51CC01 si fà riferimento ad una particolare applicazione ed ad una serie di primitive (funzioni API) che hanno come particolarità quella di essere collocate in un'area di memoria dedicata a cui il microcontrollore può accedere al reset del sistema in alternativa alla "User Space area". Il bootloader potendo andare in esecuzione alternativamente all'applicazione utente può essere utilizzato per implementare funzionalità particolari che non hanno utilità quando il μC lavora in modalità operativa "normale".

L'ISP consente per mezzo del bootloader che fornisce tutte le API necessarie di programmare e riprogrammare il microcontrollore senza rimuoverlo dal sistema in cui è inserito e senza la necessità di preprogrammarlo con un'applicazione dedicata allo scopo. ATMEL fornisce i propri integrati in due versioni che si distinguono per il tipo di API che il bootloader mette a disposizione e con cui gestisce l'ISP: una prima versione fornisce il supporto per la programmazione seriale tramite RS232 (UART), una seconda versione la gestisce attraverso il bus CAN.

> In conclusione la "User Space area memory" del microcontrollore T89C51CC01 può essere programmata in modalità "paral-

[1] Il bit X2B consente di abilitare la modalità X2 al reset via hardware, BLJB consente di stabilire in caso di attivazione via software del bootloader se attivare quello "standard" precaricato dalla stessa ATMEL in FM1 o quello "customizzato" dall'utente e caricato in FM0.

[2] Nel relativo datasheet ATMEL si usa il termine di "parallel mode" riferendosi alla modalità di programmazione del chip attraverso un programmatore dedicato e per distinguere questa tecnica da quella di programmazione sfruttando l'ISP.

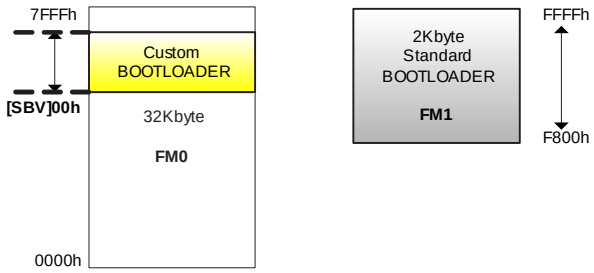


Figura B.2: ATMEL T89C51CC01: area di memoria destinata al bootloader.

lela" attraverso opportuno programmatore oppure sfruttando la funzionalità di ISP gestita attraverso le API del bootloader[3].

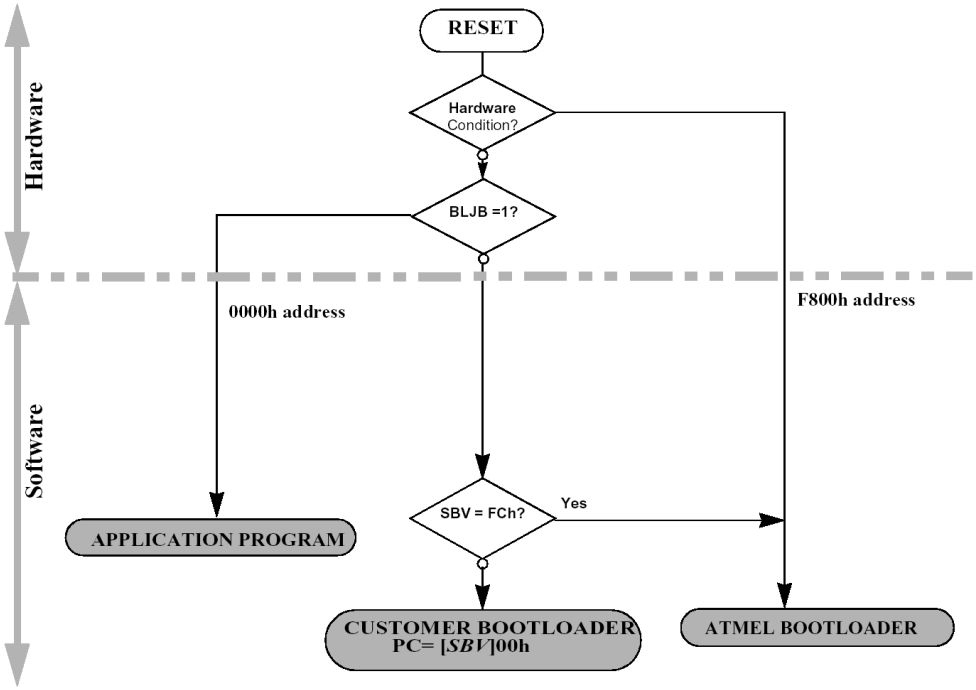


Figura B.3: ATMEL T89C51CC01: schematizzazione del processo di "boot" dopo il reset del sistema.

[3]All'area FM1 si ha accesso in scrittura solo per mezzo della programmazione "parallela". Il progettista può anche realizzare un proprio bootloader in cui integrare le funzionalità che desidera e caricarlo in una porzione di memoria dedicata dal micro a tale scopo (in FM0 a partire dall'indirizzo di memoria [SBV]00h dove SBV è il "Software Boot Vector" byte contenuto nella "Extra Row" descritta nella sezione B.1).

Il bootloader caricato in FM1 può essere attivato direttamente dall'applicazione oppure via hardware immediatamente dopo il reset; è questa seconda alternativa quella di interesse per implementare la funzione di programmazione del micro nella CAN*interface*. L'attivazione via hardware si verifica quando sul fronte di discesa del segnale di RESET viene rilevata la seguente condizione:

- PSEN basso,
- EA alto,
- ALE valore alto o non connesso;

come conseguenza il "Program Counter register" (PC) che contiene l'indirizzo della locazione di memoria contenente il codice da eseguire viene inizializzato a F800h (l'inizio di FM1): il bootloader viene mandato in esecuzione al posto dell'applicativo e rimane in attesa di ricevere il comando di inizializzazione dall'host[4] (trasmissione del carattere "U").

Stabilita la comunicazione le informazioni che il micro invia all'host dipendono solo dai comandi dell'host; i software che girano sull'host e comunicano con il μC attraverso le API del bootloader sono detti "In-Application Programming" (IAP) ed a questa categoria appartiene anche il "FLIP" fornito dalla ATMEL e che si è utilizzato per lo sviluppo di CAN*analyser* (figura B.4).

[4]Questo comando viene inviato ripetutamente dall'host al microcontrollore e viene utilizzato dal bootloader per riconoscere il bitrate impostato dall'host (sapendo cosa deve ricevere testa diverse configurazioni di bitrate fino a trovare quella che gli consente di riceve proprio il carattere "U", se la configurazione è errata leggerà un byte differente).

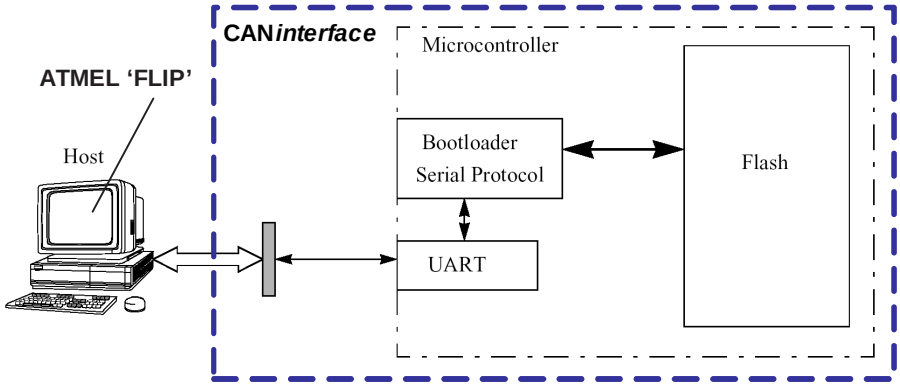


Figura B.4: ATMEL T89C51CC01: schema di programmazione della flash memory.

Bibliografia

[1] ATMEL Corporation. *In-System Programming-Serial Protocol Stack for T89C51CC01 Product*, 2001. www.atmel.com.

[2] ATMEL Corporation. *Atmel T89C51CC01 Datasheet*, 2002. www.atmel.com.

[3] TEMIC Semiconductors. *Hardware Description of the C51 Family Products*, 1997.

Appendice C

Gestione dei parametri di bit timing nel T89C51CC01

I parametri di bit timing che utilizza il CAN controller integrato nel microcontrollore T89C51CC01 vengono impostati per mezzo di tre registri che di seguito descriviamo.

CANBT1 Gli unici bit significativi del "CAN Bit Timing register 1"sono i bit "BRP5:0" che definiscono il "Baud rate prescaler". Il Time Quantum utilizzato come unità temporale di riferimento dal CAN controller viene determinato applicando la seguente relazione:

$$t_q = \frac{BRP[5 :: 0] + 1}{f_{CAN}}$$

BRP[5::0] può assumere un valore compreso tra 1 e 64 e quindi il Time Quantum può essere pari o al più 64 volte più ampio del periodo di clock con cui lavora il CAN controller (tabella C.1).

CANBT2 Il "CAN Bit Timing register 2" (tabellaC.1) codifica nel campo "SJW1:0" la dimensione del "Resynchronization Jump Width" nel rispetto della seguente relazione:

$$t_{SJW} = t_q(SJW[1..0] + 1)$$

Bit Number	Bit Mnemonic	Description
7	-	Reserved
6..1	BRP5:0	**Baud Rate prescaler**
0	-	Reserved

Tabella C.1: ATMEL T89C51CC01: struttura del CAN Bit Timing register 1.

Bit Number	Bit Mnemonic	Description
7	-	Reserved
6..5	SJW1:0	**Resynchronization Jump Width**
4	-	Reserved
3..1	PRS2:0	**Programming Time Segment**
0	-	Reserved

Tabella C.2: ATMEL T89C51CC01: struttura del CAN Bit Timing register 2.

che limita il valore minimo di t_{SJW} ad un Time Quanta come previsto dalle specifiche del protocollo. Il campo "PRS2:0" costituisce il "Propagation delay segment" attraverso cui si imposta un valore di recupero del ritardo pari a:

$$t_{PROP_SEG} = t_q(PRS[2..0] + 1)$$

CANBT3 Attraverso il "CAN Bit Timing register 3" (tabellaC.3) il micro completa l'implementazione dei parametri di bit timing del nodo CAN. "PHS22:0" costituisce la dimensione del "Phase Buffer Segment 2":

$$t_{PHASE_SEG2} = t_q(PHS2[2..0] + 1)$$

mentre con "PHS12:0" si dimensiona il "Phase Buffer Segment 1 ":

$$t_{PHASE_SEG1} = t_q(PHS1[2..0] + 1)$$

Infine, il bit meno significativo e indicato con "SMP" identifica la tecnica di campionamento:

- 0 - il segnale viene campionato un'unica volta in corrispondenza "Nominal Sample Point";
- 1 - il segnale viene campionato 3 volte. Oltre al campionamento in corrispondenza del "Nominal Sample Point" gli altri due campioni vengono prelevati in corrispondenza di un t_q e $\frac{1}{2}t_q$ secondi prima.

In figura C.1 viene rappresentato il meccanismo implementato dal T89C51CC01 per ricavare dalla frequenza dell'oscillatore locale (f_{osc}) i parametri temporali usati dal CAN controller.

Bit Number	Bit Mnemonic	Description
7	-	Reserved
6..4	PHS2 2:0	**Phase Buffer Segment 2**
3..1	PHS1 2:0	**Phase Buffer Segment 1**
0	SMP	**Sample Type**

Tabella C.3: ATMEL T89C51CC01: struttura del CAN Bit Timing register 3.

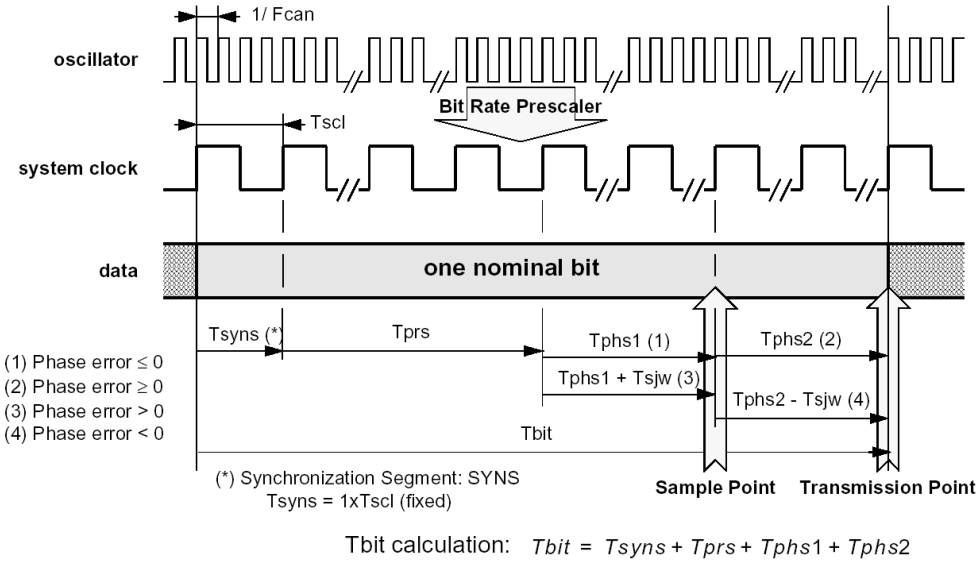


Figura C.1: ATMEL T89C51CC01: struttura generale del bit time ricavato dai parametri di configurazione.

Bibliografia

[1] ATMEL Corporation. *Atmel T89C51CC01 Datasheet*, 2002. www.atmel.com.

Appendice D

Lo standard OBD-II

Lo standard OBD-II o ODB2 (vehicle On-Board Diagnostics II) definisce i protocolli di comunicazione e le caratteristiche fisiche del connettore utilizzato per l'acquisizione dei dati di diagnostica che le centraline installate nelle autovetture devono fornire per legge[1] e che il costruttore stesso sfrutta come sistema di diagnostica del mezzo (test, revisioni, aggiornamenti). Tutte le automobili ed in generale tutti i mezzi di trasporto di piccola stazza venduti negli Stati Uniti dopo il 1° Gennaio 1996 devono possedere on-board un connettore OBD-II ed implementare un sistema di comunicazione in grado di fornire attraverso di questo informazioni sulle emissioni ambientali ed altri dati appartenenti alla lista dei così detti "Diagnostic Trouble Codes" o DTC "generici".

Teoricamente attraverso l'OBD-II è possibile acquisire qualunque tipo di dato che i sensori sparsi nel veicolo sono in grado di produrre, purtroppo benché l'OBD-II sia uno "standard" ogni produttore è lasciato libero di implementarlo come vuole; ogni azienda di autoveicoli definisce i propri codici ed i propri protocolli le cui specifiche non sono normalmente rese pubbliche, fanno eccezione i codici di richiesta ed i protocolli per la gestione dei DTC "generici[2]".

Lo standard supporta diverse interfacce come mostra la figura D.1 dove si è riprodotto lo schema del connettore previsto dalle specifiche. Le interfacce elettriche utilizzate per la gestione delle informazioni di diagnostica sono:

[1] OBD-II nasce come standard negli Stati Uniti, è stato recepito dal Canada mentre in europa ne è stata sviluppata una versione molto simile allo standard originario chiamata "European OBD" (EOBD).

[2] Tra questi si possono ricordare: i dati principali di funzionamento del motore, la temperatura del radiatore, del carburante, dell'olio del motore, i dati di carburazione (pressione dell'aria, del carburante, apertura valvola a farfalla ...), RPM, velocità, dati del sensore di O2 e i dati di funzionamento del sistema di ricircolo dei gas esausti.

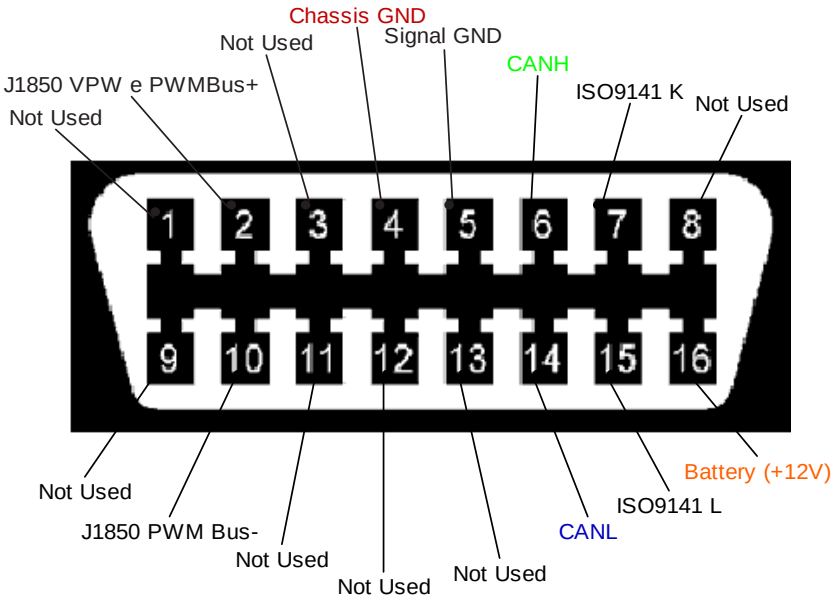


Figura D.1: OBD-II: schema del connettore di diagnostica on-board.

- il "CAN";
- l' "ISO 9141-2" usato da Chrysler e da molti importatori asiatici (Mazda) ed europei;
- il "SAE J1850 PWM" (41.6KHz Pulse With Modulated) utilizzato attualmente nella maggior parte dei veicoli prodotti da Ford;
- e il "SAE J1850 VPW" (10.4KHz Variable Pulse Width) usato nei veicoli General Motors.

mentre il protocollo di alto livello utilizzato sulle interfacce "SAE J1850" è il "SAE J1939", l'interfaccia "ISO 9141" è gestita anche attraverso il "Keyword 2000" (KWP 2000) mentre la rete CAN viene sfruttata per mezzo del protocollo sviluppato ad hoc per applicazioni di diagnostica, l' "ISO 15765".

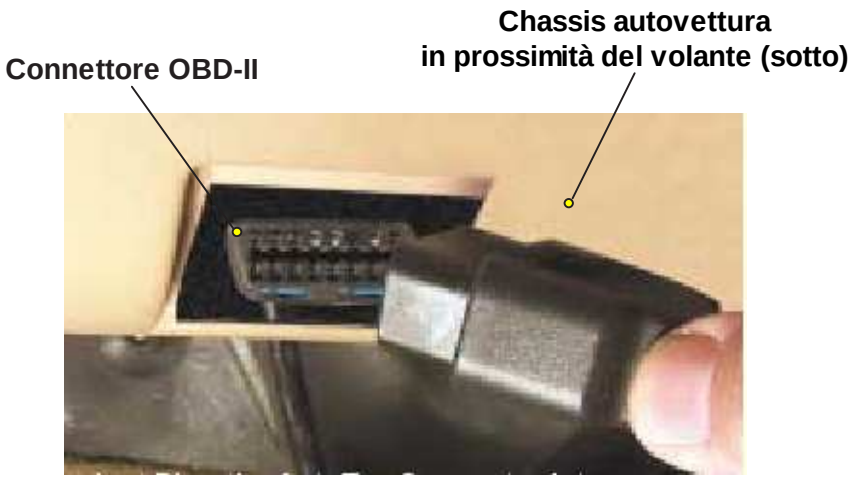


Figura D.2: OBD-II: posizionamento abituale del connettore.

Bibliografia

[1] WEB SITE. www.obd2crazy.com.

[2] WEB SITE. www.obdii.com.

[3] WEB SITE. www.obd-2.de.

Appendice E

Diffusione del CAN nei sistemi di diagnostica on-board

Nel marzo del 2002 l'istituto americano "Equipment and Tool Institute" (ETI) ha eseguito un'indagine che aveva lo scopo di raccogliere informazioni sui protocolli adottati nei sistemi interni di diagnostica dai produttori mondiali di autovetture, il rapporto che ne è derivato è schematizzato nelle tabelle: E.1, E.2 e E.3 [1].

I dati raccolti (forniti direttamente dalle case costruttrici) mostrano l'esistenza di una tendenza comune a quasi tutte le case automobilistiche nell'adottare il protocollo CAN come interfaccia elettrica standard.

[3]Fino al 2002 compreso i veicoli Ford che non gestiranno il protocollo CAN impiegheranno il J1850-41.6 con le seguenti eccezioni: Escort 1996-1.8L protocollo ISO9141, Probe 1996/97-2.5L protocollo ISO9141, Aspire 1996/97 ISO9141.

[4]Le celle colorate indicano che la compagnia non ha fornito risposta in merito ai protocolli adottati.

OBDII Generic Communication Protocols by Manufacturer [Revised Feb. 17, 2003]

Make - Model	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008
Audi/VW	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO9141 KWP2000	ISO9141 KWP2000	ISO9141 KWP2000	ISO9141 KWP2000 CAN	ISO9141 KWP2000 CAN	ISO9141 KWP2000 CAN	CAN
Bentley	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141 (50%) KWP2000 (50%)	ISO 9141 (35%) KWP2000 (65%)	ISO 9141 (20%) KWP2000 (80%)	CAN	CAN
BMW - Mini	N/A	N/A	N/A	N/A	N/A	N/A	KWP2000	KWP2000	KWP2000	KWP2000			
BMW 3-Series, 5-Series, X5, Z3, Z4, Z8, 740i, 740iL, 750iL	ISO 9141/2	ISO 9141/2	ISO 9141/2	ISO 9141/2	ISO 9141/2	ISO 9141/2	ISO 9141/2	ISO 9141/2	ISO 9141/2	ISO 9141/2			
BMW 5-Series; next generation (E60)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	KWP2000	KWP2000			
BMW 745i, 745Li, 765Li	N/A	N/A	N/A	N/A	N/A	N/A	KWP2000	KWP2000	KWP2000	KWP2000			
Chrysler	ISO 9141-2 (100%)	ISO 9141-2 (100%)	ISO 9141-2 (95%) J1850-10.4 (5%)	ISO 9141-2 (85%) J1850-10.4 (15%)	ISO 9141-2 (75%) J1850-10.4 (25%)	ISO 9141-2 (35%) J1850-10.4 (65%)	ISO 9141-2 (15%) J1850-10.4 (85%)	ISO 9141-2 (5%) J1850-10.4 (95%)	CAN (5%) J1850-10.4 (95%)	CAN (15%) J1850-10.4 (85%)	CAN (35%) J1850-10.4 (65%)	CAN (85%) J1850-10.4 (15%)	CAN (100%)
Daewoo-Lanos	N/A	N/A	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	N/A					
Daewoo-Leganza	N/A	N/A	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	KWP2000	KWP2000					
Daewoo-Nubira	N/A	N/A	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	KWP2000	KWP2000					
Ford [3]	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	0% CAN	5% CAN	50% CAN	85% CAN	90% CAN	100% CAN	100% CAN

Figura E.1: Tabella riepilogativa dei protocolli attualmente usati e prospettive di impiego del protocollo CAN in applicazioni automotive: PARTE I.

Make - Model	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008
GM (w/o exceptions)	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	80%: J1850-10.4 15%: CAN 5%: KWP2000	55%: J1850-10.4 40%: CAN 5%: KWP2000	25%: J1850-10.4 70%: CAN 5%: KWP2000	25%: J1850-10.4 70%: CAN 5%: KWP2000	100% CAN
exceptions:									Refer to the row above labeled GM (w/o exceptions)				
Saturn LS 3.0L	N/A	N/A	N/A	N/A	KWP2000	KWP2000	KWP2000	KWP2000					
Saturn VUE 3.0L	N/A	N/A	N/A	N/A	N/A	N/A	KWP2000	KWP2000					
Saturn VUE 3.5L	N/A	N/A	N/A	N/A	N/A	N/A	N/A	KWP2000					
Saturn ION	N/A	N/A	N/A	N/A	N/A	N/A	N/A	CAN					
Cadillac Catera	N/A	KWP2000	KWP2000	KWP2000	KWP2000	KWP2000	N/A	N/A					
Cadillac CTS	N/A	N/A	N/A	N/A	N/A	N/A	N/A	KWP2000					
GEO Metro	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	N/A	N/A					
GEO Prism	J1850-10.4	J1850-10.4	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	N/A					
GEO Tracker	ISO 9141	ISO 9141	ISO 9141	KWP2000	KWP2000	KWP2000	KWP2000	KWP2000					
Pontaic Vibe	N/A	N/A	N/A	N/A	N/A	N/A	N/A	ISO 9141					
Saab 9-3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	CAN					
Kia Sorento	N/A	N/A	N/A	N/A	N/A	N/A	N/A	KWP 2000					
Honda/Acura	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	70%: ISO 9141 30%: CAN	40%: ISO 9141 60%: CAN	CAN
Hyundai	ISO 9141	ISO 9141	ISO 9141	KWP2000	ISO 9141 KWP2000	ISO 9141 KWP2000	ISO 9141 KWP2000	ISO 9141 KWP2000	ISO 9141 KWP2000	ISO 9141 KWP2000	ISO 9141 KWP2000	ISO 9141 KWP2000	CAN
Isuzu	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	J1850-10.4	UNKNOWN	UNKNOWN	UNKNOWN
Jaguar	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141		
Kia - Optima	N/A	N/A	N/A	N/A	N/A	KWP 2000	KWP 2000	KWP 2000					
Kia - Rio	N/A	N/A	N/A	N/A	N/A	ISO 9141	ISO 9141	ISO 9141					
Kia - Sephia	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Kia - Spectra	N/A	N/A	N/A	N/A	ISO 9141	ISO 9141	ISO 9141	ISO 9141					
Kia - Sportage	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141					
Kia Sedona	N/A	N/A	N/A	N/A	N/A	N/A	ISO 9141	ISO 9141					
Kia Sorento	N/A	N/A	N/A	N/A	N/A	N/A	N/A	KWP 2000					
Land Rover	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	0%: CAN	0%: CAN	0%: CAN	62%: CAN	100%: CAN	100%: CAN	100%: CAN

Figura E.2: Tabella riepilogativa dei protocolli attualmente usati e prospettive di impiego del protocollo CAN in applicazioni automotive: PARTE II.

Make - Model	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008
Mazda	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	75%-ISO 9141 25%-CAN	60%-CAN 40%-ISO 9141	60%-CAN 40%-ISO 9141	CAN	CAN	CAN
Mazda-Ford	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	J1850-41.6	50%-J 1850 50%-CAN	50%-J 1850 50%-CAN	Tribute-CAN Truck?	?
Mercedes	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141 KWP 2000	ISO 9141 KWP 2000	ISO 9141 KWP 2000	KWP 2000	KWP 2000 CAN	KWP 2000 CAN	KWP 2000 CAN	KWP 2000 CAN	CAN
Mitsubishi	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141							
Nissan/Infiniti	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	TBD	TBD	CAN	CAN
Porsche	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141					
Rolls Royce[4]	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	KWP 2000	KWP 2000			
Subaru	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141 KWP 2000	ISO 9141 KWP 2000	KWP 2000	CAN	CAN	CAN
Suzuki Aerio 2.0L	N/A	N/A	N/A	N/A	N/A	N/A	KWP 2000	KWP 2000	KWP 2000	KWP 2000	KWP 2000	CAN	CAN
Suzuki Esteem 1.6L	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A
Suzuki Esteem 1.8L	N/A	N/A	N/A	ISO 9141	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A
Suzuki Grand Vitara 2.5L	N/A	N/A	N/A	ISO 9141	ISO 9141	ISO 9141	ISO 9141	KWP 2000	KWP 2000	KWP 2000	KWP 2000	CAN	CAN
Suzuki Grand Vitara 2.7L	N/A	N/A	N/A	N/A	N/A	KWP 2000	KWP 2000	KWP 2000	KWP 2000	KWP 2000	KWP 2000	CAN	CAN
Suzuki Sidekick 1.6L	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Suzuki Sidekick 1.8L	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Suzuki Swift 1.3L	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Suzuki Vitara 1.6L	N/A	N/A	N/A	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Suzuki Vitara 2.0L	N/A	N/A	N/A	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	CAN	CAN
Suzuki X90 1.6L	ISO 9141	ISO 9141	ISO 9141	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Toyota/Lexus	J1850-10.4	J1850-10.4 / ISO 9141	J1850-10.4 / ISO 9141	J1850-10.4 / ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141 / CAN	ISO 9141 / CAN	ISO 9141 / CAN	TBD	TBD
Volvo	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	ISO 9141	90% ISO 9141 10% CAN	5% ISO 9141 95% CAN	CAN	CAN	CAN

Figura E.3: Tabella riepilogativa dei protocolli attualmente usati e prospettive di impiego del protocollo CAN in applicazioni automotive: PARTE III.

Bibliografia

[1] ETI-Equipment and Tool Institute. http://www.etools.org/i4a/pages/-index.cfm?pageid=1866. www.etools.org.

Appendice F

Formati dei bit

CAN*software* gestisce i bit del campo DATA dei frame CAN in modi differenti secondo le scelte fatte dall'operatore in fase di definizione dei segnali. Conoscere in quale maniera vengono elaborati i dati è di fondamentale importanza per definire in modo corretto i segnali e quindi avere risposte che abbiano una certa attendibilità.

F.1 Il campo "Format Type"

I byte del campo DATA arrivano in sequenza da D7 a D0, il primo bit è sempre il più significativo (bit 7 = MSB). Il campo "Start Bit" nella finestra di editazione dei segnali contiene l'indice che individua il bit da cui CAN*software* parte per costruire il segnale. In figura F.1 sono tratteggiati i "Lenght" bit utilizzati, come si può notare vengono selezionati partendo dallo "Start Bit" e procedendo verso destra cioè verso gli ultimi bit arrivati. Le frecce indicano in quale ordine i bit vengono interpretati partendo sempre da quello meno significativo. Come è mostrato in figura l'operatore ha la possibilità di scegliere due metodi indicati con "LSB_MSB" e "MSB_LSB".

F.2 Il campo "Data Type"

La sequenza di bit costruita deve essere "interpretata"; CAN*software* consente di selezionare tre possibili "formattazioni".

Unsigned : la sequenza è interpretata come un numero codificato in base 2. Il numero è sempre positivo, nessun bit è riservato alla codifica del segno:

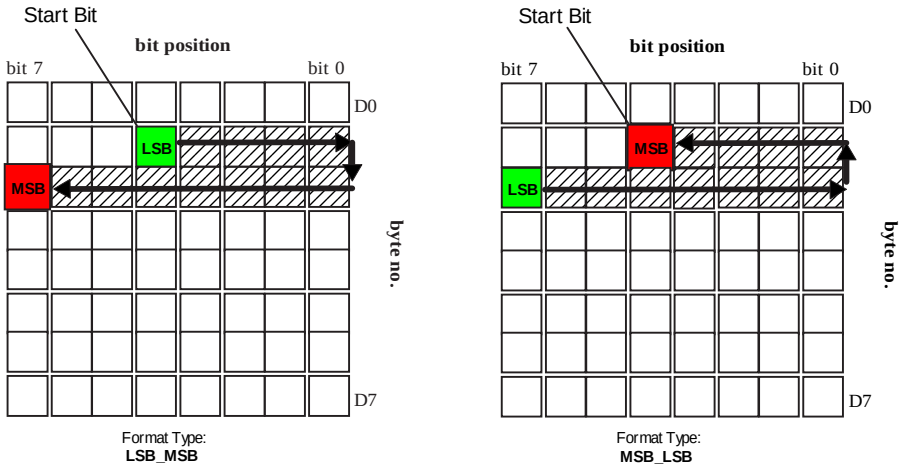


Figura F.1: CAN*software*: schematizzazione di come il CAN*software* interpreta i campi "Start Bit" e "Lenght".

MSB			LSB		Unsigned
XXXX	XXXX	XXXX	XXXX		
1100	0000	0100	0000	=	+C040h

Signed : il valore del segnale è codificato in modulo e segno. Il primo bit della sequenza (indicato con "S") se settato indica che l'intero è negativo.

MSB			LSB		Signed
SXXX	XXXX	XXXX	XXXX		
1100	0000	0100	0000	=	-4040h

Float : la sequenza codifica il numero in formato floating point secondo quanto previsto dalle specifiche IEEE-4 in cui il float è costituito da 4 byte.

MSB			LSB		Float
SEEE	EEEE	EMMM	MMMM		
1100	0000	0100	0000	=	-3

Con E si indicano i bit di esponente (8), con M quelli di mantissa (23), S è il bit del segno. CAN*software* esegue la conversione senza segnalare errore se nella sequenza di bit da convertire esiste almeno un bit di mantissa; il parametro"Lenght" deve quindi essere maggiore o uguale a 10 affinché non venga generato un errore. Se la sequenza è composta

da più di 4 byte il 33-mo bit ed i successivi non vengono considerati ed anche in questo caso l'elaborazione viene comunque eseguita.

Appendice G

Calcolo del "WorkLoad"

Il parametro di "WorkLoad" o "bus load" esprime in termini percentuali l'occupazione di banda del bus. CAN*software* fornisce all'utente due versioni di questo parametro (vedi figura G.1) uno istantaneo ed un valore mediato su una data finestra temporale di osservazione.

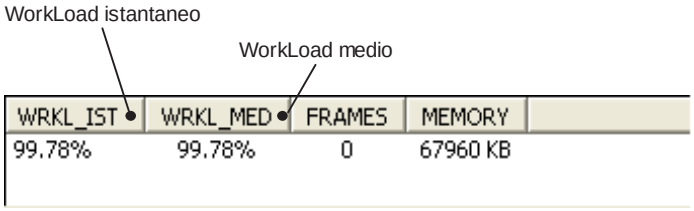


Figura G.1: CAN*software*: finestra di visualizzazione dei parametri di "WorkLoad".

G.1 "WorkLoad" istantaneo

Il coefficiente di "WorkLoad" viene calcolato utilizzando la seguente formula:

$$WorkLoad_{ist.} = 100\frac{T_1 + T_i}{T_2} \quad [\%]$$

dove:

T_1 : è la durata del frame. Viene calcolata tenendo conto del baudrate della rete CAN, della lunghezza in bit dei singoli campi che compongono il messaggio e del numero di bit di stuff presenti nel frame (il messaggio trasferito sul bus è codificato con la tecnica del bit stuffing e quindi la

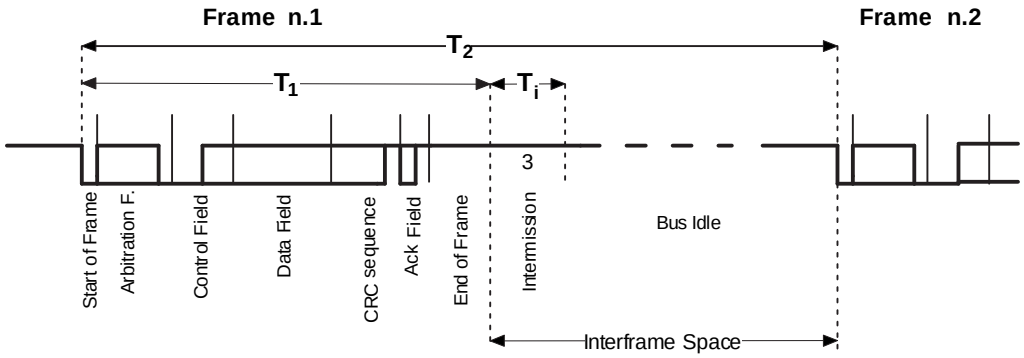


Figura G.2: CAN*software*: parametri per il calcolo del "WorkLoad".

durata effettiva di trasmissione dipende anche dal numero di stuff bit in esso presenti);

T_2 : è l'intervallo di tempo che passa tra l'arrivo di un frame ed il successivo;

T_i : è un valore costante di 3 "bit time" che corrisponde alla durata temporale dell' "Intermission Frame[1]".

Sicuramente il calcolo del tempo T_1 è l'operazione più complessa richiesta a CAN*software* nella determinazione del "WorkLoad". Oltre al fatto che i bit che compongono i campi in cui è suddiviso il frame hanno dimensione che varia con il tipo di messaggio[2] e vanno quindi ricalcolate ogni volta, deve anche essere calcolato il numero dei bit di stuff la cui determinazione passa attraverso il calcolo della sequenza di CRC[3].

$$T_1 = \frac{bit_{FRAME} + bit_{STUFF}}{Baudrate_{CAN}} \quad [s]$$

[1] In pratica si suppone che l'"Interframe Space" che segue l'invio di un frame DATA o REMOTE sul bus prima che la linea torni in stato di idle abbia durata minima; questo frame dura almeno 3 "bit time" e cioè la durata dell'"Intermission Frame".

[2] Ad esempio un frame di tipo "extended" ha un identificatore a 29 bit (anziché 11) e 2 bit in più nel campo di controllo, un DATA frame ha un numero di byte di dati variabile mentre un REMOTE frame non ha campo data ecc...

[3] Si ricorda che il protocollo CAN prevede che i segmenti che fanno parte della "Stuffing Area" siano codificati con il metodo del bit stuffing; all'interno di quest'area è compresa anche la sequenza di CRC e quindi, per poter calcolare il numero dei bit di stuff trasmessi in un frame c'è bisogno anche di ricavare dal messaggio destaffato il CRC.

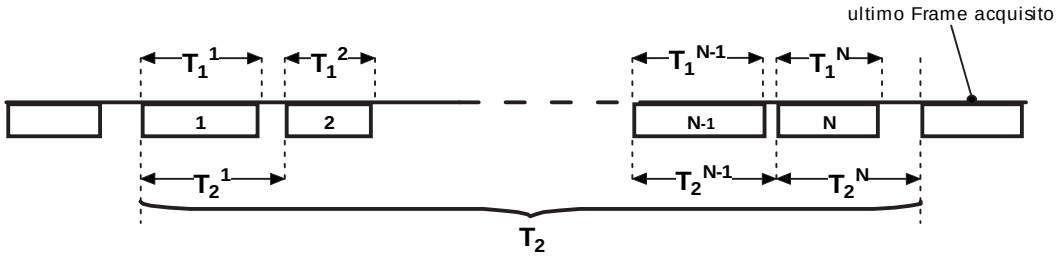


Figura G.3: CAN*software*: apertura della finestra temporale per il calcolo del parametro "WorkLoad" medio.

G.2 "WorkLoad" medio

Il "WorkLoad" istantaneo è un parametro che non ha "memoria" di quanto transita sulla rete con il passare del tempo; a questo problema si pone rimedio definendo un secondo coefficiente ottenuto in questo modo:

$$WorkLoad_{medio} = \frac{\sum_{k=1}^{N}(T_1^k + T_i)}{\sum_{k=1}^{N} T_2^k}$$

dove N è il numero di campioni che l'utente specifica attraverso il parametro "Number of samples" nella finestra "Parameters Dialog" a cui si accede dal menu "Setup". Specificando il numero N di campioni da considerare nella "media" non si fà altro che definire l'estensione di una finestra temporale di osservazione del bus (di durata T_2) ed all'interno di questa viene calcolato il nostro parametro.

L'implementazione ha richiesto l'uso di due vettori[4] nei quali memorizzare gli ultimi valori dei coefficienti T_1^k e T_2^k in modo tale da eseguire il calcolo sempre sui dati riferiti agli *ultimi* N messaggi arrivati; solo in questo modo il parametro $WorkLoad_{medio}$ visualizzato finisce per essere un coefficiente di una qualche attendibilità ed utilità.

[4]La dimensione massima assegnata alle due strutture dati è attualmente di 1024 elementi ma niente vieta di estenderla.

Appendice H

Struttura dei file esportati

CAN*software* è in grado di esportare i messaggi CAN acquisiti, in file di formati adatti ad essere rielaborati con altri applicativi; vediamo la struttura dei file generati a seguito di queste operazioni.

H.1 Esportazione in formato ".csv"

Di seguito si riporta la prima parte del file testo generato dalla procedura di esportazione.

```
TIME: 22:43:17 DATE: 11/03/03 GENERAL INFO: DATI FORMATTATI

ACCEPTANCE FILTER PARAM: TimeStamp EN - Message EN Data/Remote -
Standard/Extended - ID:0x000 - Mask:0x000

BIT TIMING PARAM: No autobaud mode - 1 sample point Can Baudrate:
1000kHz -> nTQ:8 TQ:125ns BRP:2 PRS:3 PHS1:2 PHS2:2 SJW:1

Parametri operatore -> WorkLoad Samples: 3

Index;TimeStamps;D/R;S/E;CANId;DLC;D1;D2;D3;D4;D5;D6;D7;D8;ACK;FRM;
CRC;STF;DE;SE;OVRBUF;FSiZ;SFF
1;0,000097125;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
2;0,000155250;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
3;0,000213375;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
4;0,000271500;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
5;0,001958625;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
```

```
6;0,002016750;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
7;0,002074875;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
8;0,002133000;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
9;0,002191125;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
10;0,003878250;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
11;0,003936375;D;S;0;1;55;;;;;;;;;;;Y;Y;N;55;3
```

Nelle prime righe vengono riportate informazioni di commento ai dati: ora e data dell'esportazione ed i parametri selezionati dall'operatore per l'acquisizione dei dati (filtri e bit timing); a queste segue una riga di intestazione e successivamente le righe contenenti i dati veri e propri.

Nel formato CSV "Comma Separated Values" originario le virgole separano i vari campi questo consente ai programmi come *Microsoft Windows EXCEL* © di ricreare la struttura dei dati leggendo il file ed inizializzando una nuova cella ogni volta che viene trovata una virgola mentre ad ogni carattere LF+CR (Line Feed + Carriage Return) corrisponde un nuovo record e quindi una nuova riga nel foglio di calcolo.

Il file ".csv" generato da CAN*software* impiega il carattere ";" come separatore di campo per consentire ad Excel nella versione Italiana in cui con il carattere "," è impiegato come separatore decimale, di trattare in modo corretto i valori inseriti nel campo "TimeStamps".

Il file generato è un puro file di testo e quindi i dati in esso contenuti possono essere letti da qualunque editor ASCII.

H.2 Esportazione in formato ".dif"

Il formato DIF "Data Interchange Format" è un metodo di memorizzazione dei dati nato *program-independent*; dato che ogni foglio di calcolo ha un proprio formato nativo con cui salva i dati, DIF è nato per consentire il rapido scambio di informazioni tra questo tipo di applicativi precedentemente incompatibili.

Un foglio di calcolo è una matrice bidimensionale di celle che possono contenere dati numerici, testo, formule ed altro; la cella è individuata da un identificatore univoco costituito normalmente dalle coordinate della cella stessa. I file in formato ".dif" contengono tutte queste informazioni essenziali[1] ed in più hanno il vantaggio di essere semplici file ASCII in cui comandi e dati sono organizzati su righe, ciò rende molto più semplice la loro creazione ed editazione.

[1] I formati proprietari abbinano a queste informazioni "essenziali" altri dati caratterizzanti dell'applicativo e che arricchiscono il file di funzionalità specifiche.

Di seguito riportiamo una parte del file ".dif" generato da CAN*software* così come appare aprendolo con un semplice editor di testo

```
TABLE
0,1
"POLLON"
VECTORS
0,414
""
TUPLES
0,23
""
DATA
0,0
""
-1,0
BOT
1,0
"TIME: 13:59:13 DATE: 11/05/03" 1,0 ""
.
.
.
-1,0
BOT
1,0
"Index"
1,0
"TimeStamps"
1,0
"D/R"
1,0
"S/E"
1,0
"CANId"
1,0
"DLC"
1,0
"D1"
1,0
```

CAN*software* implementa l'esportazione in questo formato per mezzo della

classe "CFileDIF[2]". Di seguito si descrive la struttura del file esportato.

H.2.1 Organizzazione di un file ".dif"

Un file DIF consiste di un header seguito dai dati veri e propri, entrambe queste sezioni sono organizzate in righe fatte in questo modo:

```
Record type
Vector, Value
"string"
```

- Il campo `Record type` specifica il significato da attribuire al campo successivo.
- Il valore `Vector` influenza l'interpretazione del campo `Value` successivo ed indica la colonna del foglio di calcolo dove Value deve essere applicato.
- Se il dato è una stringa di caratteri questa viene inserita tra doppi apici nel campo `string`; una riga vuota viene indicata con doppi apici.

Valori possibili per il campo `Record type` sono ad esempio: TABLES, VECTOR, TUPLES, DATA, COMMENT, LABEL, UNITS ecc...

H.2.2 Header

L'header costituisce il preambolo del file ed è sempre composto almeno dai quattro `Record type` seguenti: TABLES, VECTOR, TUPLES e DATA. In particolar modo:

TABLE : individua l'inizio dell'header e nel campo stringa permette di qualificare l'applicazione generatrice del file o di inserire altre informazioni.

VECTORS : specifica il numero di colonne che compongono il foglio di calcolo.

TUPLES : è il numero di record (righe) totali del database.

DATA : marca la fine dell'header e quindi l'inizio della sezione dedicata ai dati (il contenuto delle celle del foglio di calcolo).

Il file generato da CAN*software* non introduce altri tipi di record.

[2]Le funzioni messe a disposizione dalla nostra classe non esauriscono completamente le potenzialità del formato DIF; se ne è limitata l'implementazione ai tipi di record ed alle caratteristiche richieste dalla natura dei dati che CAN*software* deve trattare.

H.2.3 Body

Il dati inseriti da CAN*software* sono stringhe, numeri o campi vuoti che il formato DIF codifica nella maniera seguente:

STRINGA:

```
1,0
"stringa"
```

NUMERO:

```
0,dato numerico
V
```

NULL:

```
1,0
""
```

Ogni record deve iniziare con la sequenza:

```
-1,0
BOT
```

mentre il file termina in questo modo:

```
-1,0
EOD
```

Quando un file DIF viene aperto in lettura i dati che vengono letti ed inseriti in celle adiacenti appartenenti allo stesso record, il passaggio al record successivo avviene quando si incontra la stringa `BOT` con cui inizia la riga successiva di dati; per questa serie di motivi è necessario specificare anche quando una cella non contiene informazioni a meno che tutte le celle che mancano per concludere il record non siano vuote.

Appendice I

Codice di MICRO*software*

I.1 File: "t89c51cc01.h"

```
/*F*********************************************************
* FILE NAME             : t89c51cc01.h
* PURPOSE               : file intestazione registri ATMEL
*********************************************************/
#include "compiler.h"


// BYTE REGISTER

Sfr (P0 , 0x80);
Sfr (P1 , 0x90);
Sbit (P1_7, 0x90, 7);
Sbit (P1_6, 0x90, 6);
Sbit (P1_5, 0x90, 5);
Sbit (P1_4, 0x90, 4);
Sbit (P1_3, 0x90, 3);
Sbit (P1_2, 0x90, 2);
Sbit (P1_1, 0x90, 1);
Sbit (P1_0, 0x90, 0);

Sfr (P2 , 0xA0);

Sbit (P2_7 , 0xA0, 7);
Sbit (P2_6 , 0xA0, 6);
Sbit (P2_5 , 0xA0, 5);
```

```
Sbit (P2_4 , 0xA0, 4);
Sbit (P2_3 , 0xA0, 3);
Sbit (P2_2 , 0xA0, 2);
Sbit (P2_1 , 0xA0, 1);
Sbit (P2_0 , 0xA0, 0);

Sfr (P3 , 0xB0);

Sbit (P3_7 , 0xB0, 7);
Sbit (P3_6 , 0xB0, 6);
Sbit (P3_5 , 0xB0, 5);
Sbit (P3_4 , 0xB0, 4);
Sbit (P3_3 , 0xB0, 3);
Sbit (P3_2 , 0xB0, 2);
Sbit (P3_1 , 0xB0, 1);
Sbit (P3_0 , 0xB0, 0);

Sbit (RD , 0xB0, 7);
Sbit (WR , 0xB0, 6);
Sbit (T1 , 0xB0, 5);
Sbit (T0 , 0xB0, 4);
Sbit (INT1, 0xB0, 3);
Sbit (INT0, 0xB0, 2);
Sbit (TXD , 0xB0, 1);
Sbit (RXD , 0xB0, 0);

Sfr (P4 , 0xC0);

Sfr (PSW , 0xD0);

Sbit (CY  , 0xD0, 7);
Sbit (AC  , 0xD0, 6);
Sbit (F0  , 0xD0, 5);
Sbit (RS1 , 0xD0, 4);
Sbit (RS0 , 0xD0, 3);
Sbit (OV  , 0xD0, 2);
Sbit (UD  , 0xD0, 1);
Sbit (P   , 0xD0, 0);

Sfr (ACC , 0xE0);
Sfr (B , 0xF0);
```

```
Sfr (SP , 0x81);
Sfr (DPL , 0x82);
Sfr (DPH , 0x83);
Sfr16(DPTR, 0x82);

Sfr (PCON , 0x87);
Sfr (CKCON , 0x8F);
#define MSK_CKCON_CANX2 0x81;
#define MSK_CKCON_WDX2  0x41;
#define MSK_CKCON_PCAX2 0x21;
#define MSK_CKCON_SIX2 0x11;
#define MSK_CKCON_T2X2  0x09;
#define MSK_CKCON_T1X2 0x05;
#define MSK_CKCON_T0X2 0x03;


// TIMER REGISTER

Sfr (TCON , 0x88);    // 00H Timer/Counter control register
Sbit (TF1 , 0x88, 7); // timer 1 overflow flag
Sbit (TR1 , 0x88, 6); // timer 1 run control bit
Sbit (TF0 , 0x88, 5); // timer 0 overflow flag
Sbit (TR0 , 0x88, 4); // timer 0 run control bit
Sbit (IE1 , 0x88, 3);
Sbit (IT1 , 0x88, 2);
Sbit (IE0 , 0x88, 1);
Sbit (IT0 , 0x88, 0);

Sfr (TMOD , 0x89);    // 00H Timer/Counter mode control register

Sfr  (T2CON , 0xC8);
Sbit (TF2   , 0xC8, 7);
Sbit (EXF2  , 0xC8, 6);
Sbit (RCLK  , 0xC8, 5);
Sbit (TCLK  , 0xC8, 4);
Sbit (EXEN2 , 0xC8, 3);
Sbit (TR2   , 0xC8, 2);
Sbit (C_T2  , 0xC8, 1);
Sbit (CP_RL2, 0xC8, 0);

Sfr (T2MOD , 0xC9);
```

```
Sfr (TL0 , 0x8A);
Sfr (TL1 , 0x8B);
Sfr (TL2 , 0xCC);
Sfr (TH0 , 0x8C);
Sfr (TH1 , 0x8D);
Sfr (TH2 , 0xCD);
Sfr (RCAP2L , 0xCA);
Sfr (RCAP2H , 0xCB);
Sfr (WDTRST , 0xA6);
Sfr (WDTPRG , 0xA7);


// UART REGISTER

Sfr (SCON , 0x98);
Sbit (SM0  , 0x98, 7);
Sbit (FE   , 0x98, 7);
Sbit (SM1  , 0x98, 6);
Sbit (SM2  , 0x98, 5);
Sbit (REN  , 0x98, 4);
Sbit (TB8  , 0x98, 3);
Sbit (RB8  , 0x98, 2);
Sbit (TI   , 0x98, 1);
Sbit (RI   , 0x98, 0);

Sfr (SBUF , 0x99);
Sfr (SADEN , 0xB9);
Sfr (SADDR , 0xA9);


// ADC REGISTER

Sfr (ADCLK , 0xF2);
Sfr (ADCON , 0xF3);
#define MSK_ADCON_PSIDLE 0x40
#define MSK_ADCON_ADEN   0x20
#define MSK_ADCON_ADEOC  0x10
#define MSK_ADCON_ADSST  0x08
#define MSK_ADCON_SCH    0x07

Sfr (ADDL , 0xF4);
```

```
#define MSK_ADDL_UTILS  0x03

Sfr (ADDH , 0xF5);
Sfr (ADCF , 0xF6);


// FLASH EEPROM REGISTER

Sfr (FCON  , 0xD1);
#define MSK_FCON_FBUSY   0x01
#define MSK_FCON_FMOD    0x06
#define MSK_FCON_FPS     0x08
#define MSK_FCON_FPL     0xF0

Sfr (EECON  , 0xD2);
#define MSK_EECON_EEBUSY 0x01
#define MSK_EECON_EEE    0x02
#define MSK_EECON_EEPL   0xF0

Sfr (AUXR  , 0x8E);
#define MSK_AUXR_M0      0x20

Sfr (AUXR1  , 0xA2);
#define MSK_AUXR1_ENBOOT 0x20


// IT REGISTER

Sfr (IPL1 , 0xF8);
Sbit (POVRL , 0xF8, 2);
Sfr (IPH1 , 0xF7);
#define MSK_IPH1_POVRH 0x04

Sfr (IEN0  , 0xA8);
Sfr (IPL0 , 0xB8);
Sfr (IPH0 , 0xB7);
Sfr (IEN1  , 0xE8);

Sbit (EA   , 0xA8, 7);
Sbit (EC   , 0xA8, 6);
Sbit (ET2  , 0xA8, 5);
```

```
Sbit (ES   , 0xA8, 4);
Sbit (ET1  , 0xA8, 3);
Sbit (EX1  , 0xA8, 2);
Sbit (ET0  , 0xA8, 1);
Sbit (EX0  , 0xA8, 0);

Sbit (ETIM , 0xE8, 2);
Sbit (EADC , 0xE8, 1);
Sbit (ECAN , 0xE8, 0);


// PCA REGISTER

Sfr (CCON , 0xD8);
Sbit(CF  , 0xD8, 7);
Sbit(CR  , 0xD8, 6);
Sbit(CCF4, 0xD8, 4);
Sbit(CCF3, 0xD8, 3);
Sbit(CCF2, 0xD8, 2);
Sbit(CCF1, 0xD8, 1);
Sbit(CCF0, 0xD8, 0);

Sfr (CMOD , 0xD9);
Sfr (CH , 0xF9);
Sfr (CL , 0xE9);
Sfr (CCAP0H  , 0xFA);
Sfr (CCAP0L  , 0xEA);
Sfr (CCAPM0  , 0xDA);
Sfr (CCAP1H  , 0xFB);
Sfr (CCAP1L  , 0xEB);
Sfr (CCAPM1  , 0xDB);
Sfr (CCAP2H  , 0xFC);
Sfr (CCAP2L  , 0xEC);
Sfr (CCAPM2  , 0xDC);
Sfr (CCAP3H  , 0xFD);
Sfr (CCAP3L  , 0xED);
Sfr (CCAPM3  , 0xDD);
Sfr (CCAP4H  , 0xFE);
Sfr (CCAP4L  , 0xEE);
Sfr (CCAPM4  , 0xDE);
```

```
// CAN REGISTER

Sfr (CANGIT , 0x9B);
#define MSK_CANGIT_CANIT        0x80
#define MSK_CANGIT_OVRTIM       0x20
#define MSK_CANGIT_OVRBUF       0x10
#define MSK_CANGIT_SERG         0x08
#define MSK_CANGIT_CERG         0x04
#define MSK_CANGIT_FERG         0x02
#define MSK_CANGIT_AERG         0x01

Sfr (CANTEC , 0x9C);
Sfr (CANREC , 0x9D);
Sfr (CANTCON , 0xA1);
Sfr (CANMSG , 0xA3);
Sfr (CANTTCL , 0xA4);
Sfr (CANTTCH , 0xA5);
Sfr (CANGSTA , 0xAA);
#define MSK_CANGSTA_OVFG        0x40
#define MSK_CANGSTA_TBSY        0x10
#define MSK_CANGSTA_RBSY        0x08
#define MSK_CANGSTA_ENFG        0x04
#define MSK_CANGSTA_BOFF        0x02
#define MSK_CANGSTA_ERRP        0x01

Sfr (CANGCON , 0xAB);
#define MSK_CANGCON_ABRQ        0x80
#define MSK_CANGCON_OVRQ        0x40
#define MSK_CANGCON_TTC         0x20
#define MSK_CANGCON_SYNCTTC     0x10
#define TTC_EOF                 0x10
#define TTC_SOF         0x00
#define MSK_CANGCON_LISTEN      0x08
#define MSK_CANGCON_ENA         0x02
#define MSK_CANGCON_GRES        0x01

Sfr (CANTIML , 0xAC);
Sfr (CANTIMH , 0xAD);
Sfr (CANSTMPL , 0xAE);
Sfr (CANSTMPH , 0xAF);
```

```
Sfr (CANPAGE , 0xB1);
Sfr (CANSTCH , 0xB2);
#define MSK_CANSTCH_DLCW        0x80
#define MSK_CANSTCH_TxOk        0x40
#define MSK_CANSTCH_RxOk        0x20
#define MSK_CANSTCH_BERR        0x10
#define MSK_CANSTCH_SERR        0x08
#define MSK_CANSTCH_CERR        0x04
#define MSK_CANSTCH_FERR        0x02
#define MSK_CANSTCH_AERR        0x01

Sfr (CANCONCH , 0xB3);
#define MSK_CANCONCH_IDE        0x10
#define MSK_CANCONCH_DLC        0x0F
#define MSK_CANCONCH_CONF       0xC0
#define DLC_MAX                    8
#define CH_DISABLE              0x00
#define CH_RxENA                0x80
#define CH_TxENA                0x40
#define CH_RxBENA               0xC0

Sfr (CANBT1 , 0xB4);
#define MSK_CANBT1_BRP          0x7E
#define CAN_PRESCALER_MIN          0
#define CAN_PRESCALER_MAX         63

Sfr (CANBT2 , 0xB5);
#define MSK_CANBT2_SJW          0x60
#define MSK_CANBT2_PRS          0x0E
#define CAN_SJW_MIN                0
#define CAN_SJW_MAX                3
#define CAN_PRS_MIN                0
#define CAN_PRS_MAX                7

Sfr (CANBT3 , 0xB6);
#define MSK_CANBT3_PHS2         0x70
#define MSK_CANBT3_PHS1         0x0E
#define CAN_PHS2_MIN               0
#define CAN_PHS2_MAX               7
#define CAN_PHS1_MIN               0
#define CAN_PHS1_MAX               7
```

```
Sfr (CANSIT1 , 0xBA);
Sfr (CANSIT2 , 0xBB);
Sfr (CANIDT1 , 0xBC);
Sfr (CANIDT2 , 0xBD);
Sfr (CANIDT3 , 0xBE);
Sfr (CANIDT4 , 0xBF);
#define MSK_CANIDT4_RTRTAG      0x04

Sfr (CANGIE , 0xC1);
#define MSK_CANGIE_ENRX         0x20
#define MSK_CANGIE_ENERCH       0x08
#define MSK_CANGIE_ENBUF        0x04
#define MSK_CANGIE_ENERG        0x02

Sfr (CANIE1 , 0xC2);
Sfr (CANIE2 , 0xC3);
Sfr (CANIDM1 , 0xC4);
Sfr (CANIDM2 , 0xC5);
Sfr (CANIDM3 , 0xC6);
Sfr (CANIDM4 , 0xC7);
#define MSK_CANIDM4_RTRMSK      0x04
#define MSK_CANIDM4_IDEMSK      0x01

Sfr (CANEN1 , 0xCE);
Sfr (CANEN2 , 0xCF);
#endif
```

I.2 File: "compiler.h"

```
/*F*********************************************************
* FILE NAME             : compiler.h
* PURPOSE               : file intestazione con le direttive
*                       : al compilatore
*********************************************************/
#ifndef _COMPILER_H_
#define _COMPILER_H_

typedef unsigned char      Uchar;
typedef unsigned int       Uint16;
```

```
typedef int                Int16;
typedef float              Float16;
typedef unsigned long int  Uint32;
typedef long int           Int32;
typedef unsigned char      Bool;


#define FALSE 0
#define TRUE  1
#define OFF   0
#define ON    1
#define CLR   0
#define SET   1
#define LOW(U16)  ((Uchar)U16)
#define HIGH(U16) ((Uchar)(U16>>8))
#define TST_BIT_X(addrx,mask) (*addrx & mask)
#define SET_BIT_X(addrx,mask) (*addrx = (*addrx | mask))
#define CLR_BIT_X(addrx,mask) (*addrx = (*addrx & ~mask))
#define OUT_X(addrx,value)    (*addrx = value)
#define IN_X(addrx)           (*addrx)


#define SET_SFR_BIT(sfr_reg, bit_pos, bit_val)
{
  sfr_reg &= ~(1<<(bit_pos));
  sfr_reg |= ((bit_val)<<(bit_pos));
}


#define TST_SFR_BIT(sfr_reg, bit_pos)
  ((sfr_reg & (1<<(bit_pos)))>>(bit_pos))


// DEFINIZIONI PER IL COMPILATORE KEIL

#ifdef KEIL
#define Reentrant(x)   x reentrant
#define Sfr(x,y)       sfr x = y
#define Sfr16(x,y)     sfr16 x = y
#define Sbit(x,y,z)    sbit x = y ^ z
#define Interrupt(x,y) x interrupt y
```

```
#define At(x)          _at_ x

#ifdef __C51__
#define far  xdata
#endif

#endif


#ifdef TASKING
#include <keil.h>
#define far             _far
#define Reentrant(x)    _reentrant x
#define Sfr(x,y)        _sfrbyte x _at(y)
#define Sbit(x,y,z)     _sfrbit x _atbit(y,z)
#define Interrupt(x,y)  _interrupt(y) x
#endif
#endif /* _COMPILER_H_ */
```

I.3 File: "can_drv.h"

```
/*F*********************************************************
* FILE NAME             : can_drv.h
* PURPOSE               : intestazione funzioni gestione
*                       : CAN controller
*********************************************************/
#ifndef _CAN_DRV_H_
#define _CAN_DRV_H_


// VARIABILI PER LA GESTIONE DELLA CODA

extern Uchar Ins;        // indice inserimento
extern Uchar Est;        // indice di estrazione
extern Uchar ElemCodaRX; // numero di elementi in coda
extern Uchar OvrBufFlag; // flag di stato overflow
extern Uchar CodaRX[];   // coda
extern Uchar OvrRunFlag; // flag di stato overrun
```

```
// PROTOTIPI DELLE FUNZIONI DI INVIO DATI

extern void send_msg_to_pc (Uchar);  // invio dati al PC
extern void send_cmdOvrRun (void);   // controllo ed invio cmdOvrRun
extern void send_cmdOvrBuf (void);   // invio comando di OverBuffer


// PROTOTIPI DELLE FUNZIONI DI GESTIONE INTERRUZIONI

extern void can_drv_it_ovrbuf (void);  // overflow
extern void can_drv_it_rxok (void);    // ricezione frame
extern void can_drv_it_overrun (void); // overrun

#endif
```

I.4 File: "can_drv.c"

```
/*F*********************************************************
* FILE NAME             : can_drv.c
* PURPOSE               : funzioni di gestione del CAN
*                       : controller
*********************************************************/
#include "config.h"
#include "compiler.h"
#include "t89c51cc01.h"
#include "can_lib.h"
#include "can_drv.h"
#include "serial_drv.h"


// DICHIARAZIONE VARIABILI E STRUTTURE DATI GLOBALI

Uchar Ins;
Uchar Est;
Uchar ElemCodaRX;
Uchar OvrBufFlag;
Uchar CodaRX[DIM_CODARX];
Uchar OvrRunFlag;
```

```
/*P*********************************************************
* FUNCTION NAME         : can_drv_it_overbuf
* PURPOSE               : funzione di gestione overflow
*                       : richiamata da int.7
*********************************************************/
void can_drv_it_ovrbuf (void)
{
  CANGIT &= ~MSK_CANGIT_OVRBUF;  // interrupt ack
  OvrBufFlag = 1;
}


/*P*********************************************************
* FUNCTION NAME         : can_drv_it_rxok
* PURPOSE               : funzione di gestione ricezione
*                       : dato richiamata da int.7
*********************************************************/
void can_drv_it_rxok (void)
{
  Uchar Channel;
  Channel = FindFirstChIt();
  CAN_SET_CHANNEL (Channel);
  CANSTCH = CANSTCH & 0xDF;
  CodaRX[Ins] = CodaRX[Ins] | Channel;

  ElemCodaRX++;
  if (Ins == CHANNEL_BUF_MODE)
    Ins = 0;
  else
    Ins++;
}


/*P*********************************************************
* FUNCTION NAME         : can_drv_it_overrun
* PURPOSE               : funzione di gestione overrun
*                       : richiamata da int.9
*********************************************************/
void can_drv_it_overrun (void)
{
  Uchar ovr;
```

```
  if ((ElemCodaRX == DIM_CODARX) || OvrBufFlag)
    OvrRunFlag = 1;
  else
  {
    ovr = CodaRX[Ins] & 0x10;
    if (!ovr)
      CodaRX[Ins] |= 0x10;
    else
      OvrRunFlag = 1;
  }
  CANGIT &= ~MSK_CANGIT_OVRTIM;
}


/*P**********************************************************
* FUNCTION NAME         : send_cmdOvrRun
* PURPOSE               : invio comando di overrun
**********************************************************/
void send_cmdOvrRun (void)
{
  Uchar command;
  OvrRunFlag = 0;
  command = cmdCommand;
  command |= cmdOvrRunFlag;
  putchar (command);
}


/*P**********************************************************
* FUNCTION NAME         : can_sendOvrBuf
* PURPOSE               : invio comando di overflow
**********************************************************/
void send_cmdOvrBuf (void)
{
  Uchar command,timeL,timeH;
  timeL = CANTIML;
  timeH = CANTIMH;
  if (timeH == 0x00)
    if (timeL <= 10)
       send_cmdOvrRun();
  timeL = timeL + 2;
```

```
  command = cmdCommand;
  command |= cmdOvrBufFlag;
  putchar (command);
  putchar (timeL);
  putchar (timeH);
}


/*P*********************************************************
* FUNCTION NAME         : send_msg_to_pc
* PURPOSE               : invio CAN message
*********************************************************/
void send_msg_to_pc (Uchar command)
{
  Uchar dlc;
  Uchar ch;
  Uchar ovr;
  Uchar CANCONCH2;
  ch  = CodaRX[Est] & 0x0F;
  ovr = CodaRX[Est] & 0x10;
  CAN_SET_CHANNEL (ch);
  dlc = CANCONCH & MSK_CANCONCH_DLC;
  CANCONCH2 = CANCONCH & 0x1F;

  if ((ElemCodaRX == 1) && OvrBufFlag)
    CANCONCH2 |= cmdOvrBufFlag;

  if (ovr)
    CANCONCH2 |= cmdOvrRunFlag;

  putchar (CANCONCH2);

// invio ID
  putchar (CANIDT4);
  if (CANCONCH2 & MSK_CANCONCH_IDE)
    putchar (CANIDT3);
  putchar (CANIDT2);
  putchar (CANIDT1);

// invio CANSTCH
  putchar (CANSTCH);
```

```
// invio DATI
  if (command & cmdMessageEnable)
  {
    if (!(CANIDT4 & MSK_CANIDT4_RTRTAG))
    {
      while (dlc)
      {
        putchar (CANMSG);
        dlc--;
      }
    }
  }

// invio CANSTAMP
  if (command & cmdTimeEnable)
  {
    putchar (CANSTMPL);
    putchar (CANSTMPH);
  }


// AGGIORNAMENTO
  CodaRX[Est] = 0;
  if (Est == CHANNEL_BUF_MODE)
    Est = 0;
  else
    Est++;

  ElemCodaRX--;
  if (!OvrBufFlag)
    ENABLE_CHANNEL_BUFFER;
}
```

I.5 File: "autobaud.a51"

```
/*F*********************************************************
* FUNCTION NAME         : autobaud.a51
* PURPOSE               : funzione di autobaud
**********************************************************/
```

```
NAME AUTOBAUD;

         PUBLIC ASM_CAN_AUTOBAUD;

USING 0


CANBT1    DATA 0B4H
CANBT2    DATA 0B5H
CANBT3    DATA 0B6H
CANSTCH   DATA 0B2H
CANGIT    DATA 09BH
CANGCON   DATA 0ABH
CANCONCH  DATA 0B3H

BSEG AT 20H.0
f_timeout:  DBIT  1
f_new_conf: DBIT  1


START SEGMENT CODE
RSEG START



/*P***********************************************************
* FUNCTION NAME         : ASM_CAN_AUTOBAUD
* PURPOSE               : testa la configurazione di
*                       : bit timing impostata
***********************************************************/

ASM_CAN_AUTOBAUD:
        Push    AR0
        Push    AR1
        Push    AR2
        Push    AR3

AUTOBAUD_INIT:
        Mov R0, #02h ; prs  = 2
        Mov R1, #01h ; phs1 = 1
        Mov R2, #01h ; phs2 = 1
```

```
        Mov R3, #00h ; brp  = 0

Orl  CANGCON, #08h ; set bit AUTOBAUD
        Call   LOAD_CAN_BIT_TIMING


TEST_STUFF_ERROR:
        Mov A, CANGIT
        Jb Acc.3, NEXT_CONFIG ; test SERG

TEST_TIMEOUT:
        Jb f_timeout, NEXT_CONFIG

TEST_RX_OK:
        Mov A, CANSTCH
        Jb Acc.5, END_ASM_CAN_AUTOBAUD ; test RXOK

UPDATE_TIMEOUT:
        Jnb f_new_conf, TEST_STUFF_ERROR
        Mov A, CANGIT
        Jnb Acc.5, TEST_STUFF_ERROR ; test OVRTIM
        Setb f_timeout

        Jmp TEST_STUFF_ERROR


NEXT_CONFIG:
        Anl CANGCON, #0FDh ; CAN_CONTROLLER_DISABLE
        Setb f_new_conf
        Clr f_timeout

        Cjne R0, #08h, phs2

        Cjne R3, #20h, brp
        Jmp AUTOBAUD_INIT

brp:
        Inc R3
        Mov R0, #02h ; prs  = 2
        Mov R1, #01h ; phs1 = 1
        Mov R2, #01h ; phs2 = 1
```

```
        Jmp LOAD_NEW_CONFIG

phs2:
        Mov A, R0
        Cjne A, AR2, INC_PHS2
        Cjne A, AR1, INC_PHS1
        Inc R0

LOAD_NEW_CONFIG:
        Call LOAD_CAN_BIT_TIMING
        Jmp TEST_RX_OK

INC_PHS2:
        Inc R2
        Jmp LOAD_NEW_CONFIG

INC_PHS1:
        Inc R1
        Jmp LOAD_NEW_CONFIG

END_ASM_CAN_AUTOBAUD:
        Pop AR3
        Pop AR2
        Pop AR1
        Pop AR0
Ret


/*P*********************************************************
* FUNCTION NAME         : LOAD_CAN_BIT_TIMING
* PURPOSE               : carica la nuova configurazione di
*                       : bit timing
**********************************************************/
LOAD_CAN_BIT_TIMING:
        Mov CANBT1, #00h
        Mov CANBT2, #00h
        Mov CANBT3, #00h
        Mov A, R3
        Rl A
        Mov CANBT1, A
        Mov A, R0
```

```
        Rl A
        Mov CANBT2, A
        Mov A, R2
        Swap A
        Mov CANBT3, A
        Mov A, R1
        Rl A
        Orl CANBT3, A

        Orl CANGCON, #02h
        Mov CANGIT, #00h;
        Mov CANSTCH, #00h;
Ret

END
```

I.6 File: "config.h"

```
/*F*********************************************************
* FUNCTION NAME         : config.h
* PURPOSE               : parametri di configurazione
*********************************************************/
#ifndef _CONFIG_H_
#define _CONFIG_H_

// FREQUENZA DI OSCILLAZIONE IN HZ

#define FOSC                    16000000


// PARAMETRI GENERAZIONE SERIAL BAUDRATE

#define BAUDRATE_115200         9
#define BAUDRATE_57600          17


// CANALI IMPIEGATI IN RECEIVE MODE

#define CHANNEL_BUF_MODE        14
```

```
#define DIM_CODARX              15


// CODICI COMANDI RICEVUTI DA CANsoftware

#define cmdActiveMonitor        0x01
#define cmdActiveSim            0x03
#define cmdSMPEnable            0x01
#define cmdMessageEnable        0x02
#define cmdTimeEnable           0x04
#define cmdAutobaudEnable       0x08
#define cmdIdeTag               0x10
#define cmdDisconnect           0x02


// CODICI COMANDI INVIATI A CANsoftware

#define cmdCommand              0x09
#define cmdOvrBufFlag           0x40
#define cmdOvrRunFlag           0x20
#define cmdAckToDISflag         0x02
#define cmdAckToAMflag         0x04
#define cmdAckToSIMflag         0x10


// FUNZIONI DI GESTIONE DELLE INTERRUZIONI

extern void fct_can_it (void);
extern void fct_tim_ovf_it (void);


// FUNZIONE DI GESTIONE AUTOBAUD

extern ASM_CAN_AUTOBAUD();
```

I.7 File: "can_lib.h"

```
/*F*********************************************************
* FUNCTION NAME         : can_lib.h
* PURPOSE               : intestazione funzioni di libreria
```

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


// MASCHERE E PARAMETRI VARI

#define NB_CHANNEL     15
#define NB_DATA_MAX     8

#define STD_ID_10_4     0
#define STD_ID_3_0      1
#define EXT_ID_28_21    0
#define EXT_ID_20_13    1
#define EXT_ID_12_5     2
#define EXT_ID_4_0      3

#define MSK_CTRL_RTR   0x80
#define MSK_CTRL_IDE   0x10
#define MSK_CTRL_DLC   0x0F

#define CONF_IDE       0x10
#define CONF_NOIDE     0
#define CONF_RTR       0x20
#define CONF_NORTR     0
#define CONF_BUFFER    0x01
#define CONF_NOBUFFER  0
#define CONF_MSK_IDE   0x80
#define CONF_NOMSK_IDE 0
#define CONF_MSK_RTR   0x4
#define CONF_NOMSK_RTR 0


// DICHIARAZIONE DELLE MACRO

#define CAN_CONTROLLER_ENABLE  { CANGCON |= MSK_CANGCON_ENA;}
#define CAN_CONTROLLER_DISABLE { CANGCON &= ~MSK_CANGCON_ENA;}
#define CAN_CONTROLLER_RESET   { CANGCON |= MSK_CANGCON_GRES;}
#define CAN_LISTEN_ENABLE      { CANGCON |= MSK_CANGCON_LISTEN;}
#define CAN_LISTEN_DISABLE     { CANGCON &= ~MSK_CANGCON_LISTEN;}
#define CAN_IT_ENABLE          { ECAN = 1;}
#define CAN_IT_DISABLE         { ECAN = 0;}
#define CANTIM_IT_ENABLE       { ETIM = 1;}
```

```
#define CANTIM_IT_DISABLE      { ETIM = 0;}
#define CAN_RX_IT_ENABLE       { CANGIE |= MSK_CANGIE_ENRX;}
#define CAN_RX_IT_DISABLE      { CANGIE &= ~MSK_CANGIE_ENRX;}
#define CAN_TX_IT_ENABLE       { CANGIE |= MSK_CANGIE_ENTX;}
#define CAN_TX_IT_DISABLE      { CANGIE &= ~MSK_CANGIE_ENTX;}
#define CAN_ERCH_IT_ENABLE     { CANGIE |= MSK_CANGIE_ENERCH;}
#define CAN_ERCH_IT_DISABLE    { CANGIE &= ~MSK_CANGIE_ENERCH;}
#define CAN_ERG_IT_ENABLE      { CANGIE |= MSK_CANGIE_ENERG;}
#define CAN_ERG_IT_DISABLE     { CANGIE &= ~MSK_CANGIE_ENERG;}
#define CAN_BUF_IT_ENABLE      { CANGIE |= MSK_CANGIE_ENBUF;}
#define CAN_BUF_IT_DISABLE     { CANGIE &= ~MSK_CANGIE_ENBUF;}


// ABILITAZIONE E DISABILITAZIONE INTERRUPT

#define CAN_CHANNEL_IT_ENABLE(channel)
  {(channel < CHANNEL_8)?(CANIE2|=(1<<channel)):
   (CANIE1|=(1<<(channel-8)));}

#define CAN_CHANNEL_IT_DISABLE(channel)
  {(channel < CHANNEL_8)?(CANIE2&=~(1<<channel)):
   (CANIE1&=~(1<<(channel-8)));}


// SELEZIONE DEL CANALE ATTIVO

#define CAN_SET_CHANNEL(channel) (CANPAGE = (channel << 4))
#define CAN_GET_CHANNEL          (CANPAGE >> 4)


// ABILITAZIONE CANALI

#define ENABLE_CHANNEL_RX
  (CANCONCH |= (MSK_CANCONCH_CONF & CH_RxENA))

#define ENABLE_CHANNEL_BUFFER
  (CANCONCH |= (MSK_CANCONCH_CONF & CH_RxBENA))

#define ENABLE_CHANNEL_TX
  (CANCONCH |= (MSK_CANCONCH_CONF & CH_TxENA))
#define DISABLE_CHANNEL
```

```
  (CANCONCH |= (MSK_CANCONCH_CONF & CH_DISABLE))

#define STD_ID(ident) ((unsigned long)ident<<16)


// DEFINIZIONE DI TIPO

typedef Uchar can_data_t [NB_DATA_MAX];
typedef union
{
  Uint32 ext;
  Uint16 std;
  Uchar  tab[4];
}
can_id_t;

typedef struct
{
  can_id_t id;
  Uchar ctrl;
  Uchar * pt_donne;
}
can_msg_t;

typedef enum
{
 CHANNEL_0, CHANNEL_1, CHANNEL_2, CHANNEL_3 ,
 CHANNEL_4, CHANNEL_5 ,CHANNEL_6 , CHANNEL_7 ,
 CHANNEL_8, CHANNEL_9 , CHANNEL_10, CHANNEL_11,
 CHANNEL_12, CHANNEL_13, CHANNEL_14, NO_CHANNEL
}
channel_t;

typedef enum
{
  CONF_DLC_0, CONF_DLC_1, CONF_DLC_2, CONF_DLC_3,
  CONF_DLC_4, CONF_DLC_5, CONF_DLC_6, CONF_DLC_7,
  CONF_DLC_8
}
dlc_t;
```

```
#define conf_rx conf
#define conf_tx conf


// DICHIARAZIONE FUNZIONI PROTOTIPO

extern can_id_t   can_rx_filt, can_rx_msk;
extern can_id_t   can_tx_id;
extern Uchar * pt_candata_tx;
extern can_msg_t * pt_st_can_rx;
extern bdata Uchar conf;
extern void CanSetBRP  (Uchar);
extern void CanSetSJW  (Uchar);
extern void CanSetPRS  (Uchar);
extern void CanSetPHS2 (Uchar);
extern void CanSetPHS1 (Uchar);
extern Uchar FindFirstChIt (void);

#define NO_CHANNEL  0xFF

extern void RazAllMailbox (void);
extern void WaitTX (Uchar num_channel);
extern void ConfChannel_Rx();
extern void SendCanMsg ();
extern void driver_can_it (void);
extern void ReadCanMsg (Uchar nextconf);

#define CHANNEL_DISABLE       0x01
#define CHANNEL_RX_ENABLE     0x02
#define CHANNEL_RXB_ENABLE    0x04
```

I.8 File: "can_lib.c"

```
/*F*********************************************************
* FUNCTION NAME         : can_lib.c
* PURPOSE               : funzioni di libreria per la
*                       : gestione del CAN controller
**********************************************************/
#include "compiler.h"
#include "config.h"
```

```
#include "t89c51cc01.h"
#include "can_lib.h"
#include "can_drv.h"

can_id_t   can_rx_filt, can_rx_msk;


// VARIABILE USATA PER LA CONFIGURAZIONE DEI CANALI

bdata Uchar conf;
Sbit( msk_ide, conf, 7);  /* just for TX      */
Sbit( msk_rtr, conf, 6);  /* just for RX      */
Sbit( rtr    , conf, 5);  /* commun Rx and TX */
Sbit( ide    , conf, 4);  /* commun RX and TX */
Sbit( buffer , conf, 0);  /* just for TX      */


// VARIABILE PER LA GESTIONE DELLE FUNZIONI SUL BIT

bdata Uchar b_var_read;
Sbit( bit_read_7, b_var_read, 7);
Sbit( bit_read_6, b_var_read, 6);
Sbit( bit_read_5, b_var_read, 5);
Sbit( bit_read_4, b_var_read, 4);
Sbit( bit_read_3, b_var_read, 3);
Sbit( bit_read_2, b_var_read, 2);
Sbit( bit_read_1, b_var_read, 1);
Sbit( bit_read_0, b_var_read, 0);

#define NEW_CONF_CH_DISABLE     bit_read_0
#define NEW_CONF_CH_RX_ENABLE   bit_read_1
#define NEW_CONF_CH_RXB_ENABLE  bit_read_2


// bit_var = CANSTCH;

#define IT_TXOK bit_6
#define IT_RXOK bit_5
#define IT_BERR bit_4
#define IT_SERR bit_3
#define IT_CERR bit_2
```

```
#define IT_FERR bit_1
#define IT_AERR bit_0


// bit_var = CANGIT;

#define IT_OVRBUF bit_4
#define IT_SERG bit_3
#define IT_CERG bit_2
#define IT_FERG bit_1
#define IT_AERG bit_0


// VARIABILE UTILIZZATA DA FindFirstChIt()

bdata Uchar bit_var;
Sbit( bit_7, bit_var, 7);
Sbit( bit_6, bit_var, 6);
Sbit( bit_5, bit_var, 5);
Sbit( bit_4, bit_var, 4);
Sbit( bit_3, bit_var, 3);
Sbit( bit_2, bit_var, 2);
Sbit( bit_1, bit_var, 1);
Sbit( bit_0, bit_var, 0);


/*P*********************************************************
* FUNCTION NAME         : RazAllMailbox
* PURPOSE               : ripulisce tutti i canali
*********************************************************/
void RazAllMailbox (void)
{
  Uchar num_channel, num_data;
  for (num_channel = 0;
    num_channel < NB_CHANNEL;
    num_channel++)
  {
    CANPAGE  = num_channel << 4;
    CANCONCH = 0;
    CANSTCH  = 0;
```

```
    CANIDT1  = 0;
    CANIDT2  = 0;
    CANIDT3  = 0;
    CANIDT4  = 0;
    CANIDM1  = 0;
    CANIDM2  = 0;
    CANIDM3  = 0;
    CANIDM4  = 0;

    for (num_data = 0; num_data < NB_DATA_MAX; num_data++)
    {
      CANMSG = 0;
    }
  }
}


/*P**********************************************************
* FUNCTION NAME         : CanSetBRP
* PURPOSE               : il byte che viene passato è usato
*                       : per la configurazione del campo
*                       : BRP di CANBT1
**********************************************************/
void CanSetBRP (Uchar prescaler)
{

  if ((prescaler <= CAN_PRESCALER_MAX) &&
      (prescaler >= CAN_PRESCALER_MIN)   )
  {
    CANBT1 = prescaler << 1;
  }

}


/*P**********************************************************
* FUNCTION NAME         : CanSetSJW
* PURPOSE               : il byte che viene passato è usato
*                       : per la configurazione del campo
*                       : SJW di CANBT2
**********************************************************/
```

```
void CanSetSJW (Uchar sjw)
{
   if ((sjw <= CAN_SJW_MAX) &&
       (sjw >= CAN_SJW_MIN)   )
  {
    CANBT2 &= ~MSK_CANBT2_SJW;
    CANBT2 |= (sjw << 5);
  }
}


/*P*********************************************************
* FUNCTION NAME         : CanSetPRS
* PURPOSE               : il byte che viene passato è usato
*                       : per la configurazione del campo
*                       : PRS di CANBT2
*********************************************************/
void CanSetPRS (Uchar prs)
{
  if ((prs <= CAN_PRS_MAX) &&
      (prs >= CAN_PRS_MIN)   )
  {
    CANBT2 &= ~MSK_CANBT2_PRS;
    CANBT2 |= (prs << 1);
  }
}


/*P*********************************************************
* FUNCTION NAME         : CanSetPHS2
* PURPOSE               : il byte che viene passato è usato
*                       : per la configurazione del campo
*                       : PHS2 di CANBT3
*********************************************************/
void CanSetPHS2 (Uchar phs2)
{
  if ((phs2 <= CAN_PHS2_MAX) &&
      (phs2 >= CAN_PHS2_MIN)   )
  {
    CANBT3 &= ~MSK_CANBT3_PHS2;  /* RAZ of field PHS2.  */
    CANBT3 |= (phs2 << 4);
```

```
  }
}


/*P*********************************************************
* FUNCTION NAME         : CanSetPHS1
* PURPOSE               : il byte che viene passato è usato
*                       : per la configurazione del campo
*                       : PHS1 di CANBT3
*********************************************************/
void CanSetPHS1 (Uchar phs1)
{
  if ((phs1 <= CAN_PHS1_MAX) &&
      (phs1 >= CAN_PHS1_MIN)   )
  {
    CANBT3 &= ~MSK_CANBT3_PHS1;
    CANBT3 |= (phs1 << 1);
  }
}


/*P*********************************************************
* FUNCTION NAME         : FindFirstChIt
* PURPOSE               : restituisce l'indice del primo
*                       : canale che ha generato una
*                       : interruzione
*********************************************************/
Uchar FindFirstChIt (void)
{
  Uchar num_channel = NO_CHANNEL;

  if (CANSIT2)
  {
    bit_var = CANSIT2;
    if(bit_0)
    {
      num_channel = CHANNEL_0;
    }
    else if (bit_1)
    {
      num_channel = CHANNEL_1;
```

```
    }
    else if (bit_2)
    {
      num_channel = CHANNEL_2;
    }
    else if (bit_3)
    {
      num_channel = CHANNEL_3;
    }
    else if (bit_4)
    {
      num_channel = CHANNEL_4;
    }
    else if (bit_5)
    {
      num_channel = CHANNEL_5;
    }
    else if (bit_6)
    {
      num_channel = CHANNEL_6;
    }
    else
    {
      num_channel = CHANNEL_7;
    }
  }
  else if (CANSIT1)
  {
    bit_var = CANSIT1;
    if(bit_0)
    {
      num_channel = CHANNEL_8;
    }
    else if (bit_1)
    {
      num_channel = CHANNEL_9;
    }
    else if (bit_2)
    {
      num_channel = CHANNEL_10;
    }
```

```
    else if (bit_3)
    {
      num_channel = CHANNEL_11;
    }
    else if (bit_4)
    {
      num_channel = CHANNEL_12;
    }
    else if (bit_5)
    {
      num_channel = CHANNEL_13;
    }
    else if (bit_6)
    {
      num_channel = CHANNEL_14;
    }
  }
  return(num_channel);
}
```

I.9 File: "serial_drv.h"

```
/*F*********************************************************
* FUNCTION NAME         : serial_drv.h
* PURPOSE               : file di intestazione funzioni di
*                       : gestione della porta seriale
*********************************************************/
void putchar (Uchar);       // funzione invio dato a seriale
Uchar getchar (void);       // funzione lettura dato da seriale
void putstring (Uchar*);    // invio stringa
void putbitstring (Uchar);
```

I.10 File: "serial_drv.c"

```
/*F*********************************************************
* FUNCTION NAME         : serial_drv.c
* PURPOSE               : funzioni di gestione della porta
*                       : seriale
*********************************************************/
```

```
#include "config.h"
#include "compiler.h"
#include "t89c51cc01.h"

bdata Uchar byte_elem;
Sbit( bit_7b , byte_elem, 7);
Sbit( bit_6b , byte_elem, 6);
Sbit( bit_5b , byte_elem, 5);
Sbit( bit_4b , byte_elem, 4);
Sbit( bit_3b , byte_elem, 3);
Sbit( bit_2b , byte_elem, 2);
Sbit( bit_1b , byte_elem, 1);
Sbit( bit_0b , byte_elem, 0);



/*P***********************************************************
* FUNCTION NAME         : putchar
* PURPOSE               : invio byte a seriale
***********************************************************/
void putchar (Uchar c)
{
  while (!TI);
  TI = 0;
  SBUF = c;
}


/*P***********************************************************
* FUNCTION NAME         : getchar
* PURPOSE               : preleva byte a seriale
***********************************************************/
Uchar getchar (void)
{
  Uchar c;
  while (!RI);
  c = SBUF;
  RI = 0;
  return (c);
}
```

```
/*P*********************************************************
* FUNCTION NAME         : RitornoCarrello
* PURPOSE               : invio caratteri di ritorno
*                       : carrello
*********************************************************/
void RitornoCarrello (void)
{
  putchar (0x0A);
  putchar (0x0D);
}


/*P*********************************************************
* FUNCTION NAME         : putstring
* PURPOSE               : invio stringa di caratteri
*********************************************************/
void putstring (Uchar* Stringa)
{
  Uchar i;
  i = 0;
  while (Stringa[i] != '\0')
  {
    putchar(Stringa[i]);
    i++;
  }
  RitornoCarrello();
}


/*P*********************************************************
* FUNCTION NAME         : putbitstring
* PURPOSE               :
*********************************************************/
void putbitstring (Uchar c)
{
byte_elem = c;

if (bit_7b)
  putchar (0x31);
else
```

```
  putchar (0x30);

if (bit_6b)
  putchar (0x31);
else
  putchar (0x30);

if (bit_5b)
  putchar (0x31);
else
  putchar (0x30);

if (bit_4b)
  putchar (0x31);
else
  putchar (0x30);

if (bit_3b)
  putchar (0x31);
else
  putchar (0x30);

if (bit_2b)
  putchar (0x31);
else
  putchar (0x30);

if (bit_1b)
  putchar (0x31);
else
  putchar (0x30);

if (bit_0b)
  putchar (0x31);
else
  putchar (0x30);

RitornoCarrello();
}
```

I.11 File: "main.c"

```
/*F*********************************************************
* FILE NAME             : main.c
*********************************************************/
#include "config.h"     // conf. parametri dispositivo
#include "compiler.h"   // macro comp., ridef. keywords
#include "t89c51cc01.h" // intestazione x atmel T89C51CC01
#include "serial_drv.h" // gestione porta seriale
#include "can_lib.h"    // API gestione can controller
#include "can_drv.h"

Uchar ini_array[14];


/*P*********************************************************
* FUNCTION NAME         : init_serial2
* PURPOSE               : inizializzazione seriale
*                       : baudrate generator TIMER2
*********************************************************/
void init_serial2 (void)
{
  Uchar BAUDRATE;
  T2CON = 0;
  PCON = PCON & 0X3F;
  SCON = 0x50;

  BAUDRATE = BAUDRATE_115200; // sel. 115.2Kbit/s

  TH2 = HIGH(65536-BAUDRATE);
  TL2 = LOW(65536-BAUDRATE);
  RCAP2L=LOW(65536-BAUDRATE);
  RCAP2H=HIGH(65536-BAUDRATE);

  TCLK = 1; // attivo seriale in TX
  RCLK = 1; // attivo seriale in RX

  TI = 1; // pronto per TX dato
  RI = 0; // pronto per RX dato

  TR2 = 1; // attivo TIMER2
```

```
}


/*P*********************************************************
* FUNCTION NAME         : init_micro
* PURPOSE               : inizializzazione microcontrollore
*********************************************************/
void init_micro (void)
{
  CKCON = 0X01;
}


/*P*********************************************************
* FUNCTION NAME         : init_global_var
* PURPOSE               : inizializzazione strutture dati
*********************************************************/
void init_global_var (void)
{
  Uchar i;
  Ins = 0;
  Est = 0;
  ElemCodaRX = 0;
  OvrBufFlag = 0;
  for (i=0; i<DIM_CODARX; i++)
  {
    CodaRX[i] = 0;
  }
  OvrRunFlag = 0;
}


/*P*********************************************************
* FUNCTION NAME         : decode_init_sequence
* PURPOSE               : inizializzazione del CAN controller
*********************************************************/
void decode_init_sequence(void)
{
  Uchar command;
  channel_t ch;
```

```
CAN_CONTROLLER_RESET;
CAN_IT_DISABLE;
RazAllMailbox();
CAN_LISTEN_ENABLE;

command = ini_array[8];

for(ch = CHANNEL_0; ch <= CHANNEL_BUF_MODE; ch++)
{
  CAN_SET_CHANNEL(ch);    // seleziono il canale 'ch'
  CANIDM1 = ini_array[0]; // configura reg. CANIDM
  CANIDM2 = ini_array[1];
  CANIDM3 = ini_array[2];
  CANIDM4 = ini_array[3];
  CANIDT1 = ini_array[4]; // configura reg. CANIDT
  CANIDT2 = ini_array[5];
  CANIDT3 = ini_array[6];
  CANIDT4 = ini_array[7];
  ENABLE_CHANNEL_BUFFER;

  if (command & cmdIdeTag)
    CANCONCH |= 0x10;     // ID standard
  else
    CANCONCH &= 0xEF;     // ID esteso

  CAN_CHANNEL_IT_ENABLE(ch);

  CANCONCH |= 0x08;
}

if (command & cmdSMPEnable)
  CANBT3 |= 0x01;         // abilito SMP
else
  CANBT3 &= 0xFE;         // disabilito SMP

CanSetBRP(ini_array[9]);  // configura parametri di
CanSetSJW(ini_array[10]); // bit timing del CAN contr.
CanSetPRS(ini_array[11]);
CanSetPHS1(ini_array[12]);
CanSetPHS2(ini_array[13]);
```

```
  CAN_IT_ENABLE;            // abilita CAN interrupt

  if (command & cmdTimeEnable)
    CANTIM_IT_ENABLE;
  else
    CANTIM_IT_DISABLE;

  CAN_BUF_IT_ENABLE;
  CAN_RX_IT_ENABLE;

  EA = 1;
  CAN_CONTROLLER_ENABLE;
}


/*P**********************************************************
* FUNCTION NAME         : autobaud
* PURPOSE               : ricava per approssimazioni successive
*                       : il baudrate della rete
**********************************************************/
void autobaud (void)
{
  Uchar i;
  Uchar CANBTapp;

  CAN_CONTROLLER_RESET;
  CAN_IT_DISABLE;
  RazAllMailbox();
  CAN_SET_CHANNEL(CHANNEL_0);
  CANIDT1  = 0;
  CANIDT2  = 0;
  CANIDT3  = 0;
  CANIDT4  = 0;
  CANIDM1  = 0;
  CANIDM2  = 0;
  CANIDM3  = 0;
  CANIDM4  = 0;
  ENABLE_CHANNEL_RX;
  ASM_CAN_AUTOBAUD();

  CAN_CONTROLLER_DISABLE;
```

```
  CANBTapp = CANBT1 & MSK_CANBT1_BRP;
  ini_array[9]  = CANBTapp >> 1;

  ini_array[10] = 0;

  CANBTapp = CANBT2 & MSK_CANBT2_PRS;
  ini_array[11] = CANBTapp >> 1;

  CANBTapp = CANBT3 & MSK_CANBT3_PHS1;
  ini_array[12] = CANBTapp >> 1;

  CANBTapp = CANBT3 & MSK_CANBT3_PHS2;
  ini_array[13] = CANBTapp >> 4;

  // INVIO DATI DI BIT TIMING CALCOLATI

  for (i=9; i<=13; i++)
    putchar (ini_array[i]);
}


/*M*********************************************************
* FUNCTION NAME         : main
* PURPOSE               : gestione sistema
*********************************************************/
void main (void)
{
  Uchar ini_command;
  Uchar hsk_command;
  Uchar ch;

  while (1)
  {
    EA= 0;
    init_global_var();
    init_micro();
    init_serial2();
    hsk_command = 0;

    // ATTESA RICEZIONE COMANDO DI ATTIVAZIONE
```

```
do
{
  ini_command = getchar();
}
while (ini_command != cmdActiveMonitor);

// INVIO RISPOSTA
hsk_command = cmdCommand;
hsk_command |= cmdAckToAMflag;
putchar (hsk_command);

// RICEZIONE COMANDI DI CONFIGURAZIONE
for (ch = 0; ch < 14; ch++)
{
  ini_array[ch] = getchar();
}
ini_command = ini_array[8];

// AUTOBAUD?
if (ini_command & cmdAutobaudEnable)
  autobaud();

decode_init_sequence();

// INVIA CAN MESSAGE FINO ALLA RICEZIONE DI UN
// COMANDO DI DISCONNESSIONE
while(!(hsk_command & cmdDisconnect))
{
  if (OvrRunFlag && (ElemCodaRX == 0))  send_cmdOvrRun();

  if (ElemCodaRX)
  {
    send_msg_to_pc(ini_command);
  }
  else
  {
    if (OvrBufFlag)
    {
      if (OvrRunFlag) send_cmdOvrRun();
      EA = 0;
      OvrBufFlag = 0;
```

```
  if (ini_command & cmdTimeEnable) send_cmdOvrBuf();
          decode_init_sequence();
}
      }

      // RICEVUTO NUOVO COMANDO?
      if (RI)
      {
        hsk_command = SBUF;
        RI = 0;
      }
    }

    // PROCEDURA DI DISCONNESSIONE
    EA = 0;
    while (ElemCodaRX)
    {
      send_msg_to_pc(ini_command);
    }

    // INVIO ACK A DISCONNESSIONE
    hsk_command = cmdCommand;
    hsk_command |= cmdAckToDISflag;
    putchar (hsk_command);
    while(!TI);
  }
}


/*I*********************************************************
* FUNCTION NAME         : interrupt 7
* PURPOSE               : routine di sevizio
*********************************************************/
Interrupt(int_7(void), 7)
{
  if (CANGIT & 0x10)
    can_drv_it_ovrbuf();  // OVERBUFFER
  else
    can_drv_it_rxok();    // RXOK
}
```

```
/*I*********************************************************
* FUNCTION NAME         : interrupt 9
* PURPOSE               : routine di sevizio
*********************************************************/
Interrupt(int_9(void), 9)
{
  can_drv_it_overrun(); // OVERRUN
}
```