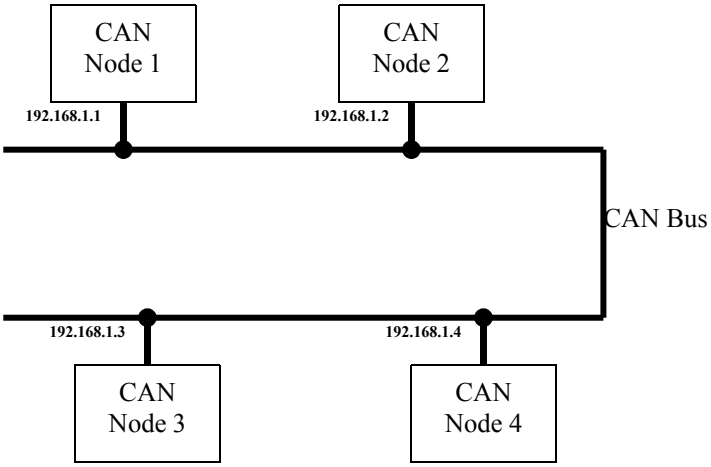
Data :dd.02.yyyy
Autore: Paolo Broggini
Final

Progetto

IP-Over-CAN



Realizzazione nell'ambito del lavoro di diploma
per lo studio

'Master in Advanced Computer Science'

Softool Microelectronics SA
CH-6653 Verscio Tel. +41 91 796 31 86 Fax +41 91 796 31 87

Sommario

1. Prefazione

1.1 Chi è la 'SOFTOOL Microelectronics SA'

Attiva da più di 20 anni, la *SOFTOOL Microelectronics SA* (www.softool.ch) ha da sempre focalizzato le proprie attività nella microelettronica, applicata nel campo dell'automazione industriale.
Grazie alle conoscenze sia nel ramo hardware sia nel software e l'esperienza acquisita nel corso degli anni, la *SOFTOOL* è in grado di offrire i seguenti servizi:

- Consulenza e pianificazione per lo sviluppo e specifiche di prodotti
- Design e analisi di sistemi basati su microprocessori
- Design e programmazione di software industriale
- Design e realizzazione di hardware industriale
- Produzione di piccole serie
- Assistenza al cliente per grosse produzioni

1.2 I sistemi 'Embedded'

Il termine 'Embedded', dall'inglese dedicato, sta ad indicare quei sistemi (basati su microprocessori) il cui ruolo è quello di svolgere un compito ben definito. A differenza di un Personal Computer la cui flessibilità permette di svolgere compiti anche radicalmente differenti fra loro (elaborazione testi, audio, video ecc.), un sistema 'Embedded' è pensato per svolgere un determinato compito, semplice o complicato che sia.
Un punto fondamentale che li differenzia da un PC sta nelle risorse messe a disposizione del software. Se da un lato i PC odierni hanno delle enormi capacità di elaborazione e memorizzazione impensabili solo fino a qualche anno fa, la maggior parte dei sistemi 'Embedded' hanno un set di risorse molto più limitate. Non è infatti raro trovare sistemi basati su microprocessori con velocità di clock di pochi MHz e capacità di RAM inferiori a 1Kb.
Oggigiorno i sistemi 'Embedded' li troviamo "nascosti" in una miriade di applicazioni (Auto, TV, telefoni cellulari, catene di produzione ecc.) e li usiamo continuamente senza quasi accorgerci della loro presenza. Da qui l'importanza di un sistema (possibilmente standard) per il loro controllo e/o monitoraggio.

1.3 Il Progetto IP-Over-CAN

L'utilizzo del protocollo TCP/IP nelle reti informatiche, spinto dall'evoluzione di Internet, è in continua espansione e risulta sempre più essere il "protocollo di riferimento" per tutte le reti informatiche.
L'implementazione di TCP/IP su CAN ne permette l'integrazione con la rete aziendale (tipicamente Ethernet).
L'obiettivo evidentemente non è quello di navigare in Internet "passando" per il CAN, bensì quello di realizzare un'infrastruttura di comunicazione uniforme.
Si pensi ad esempio, all'utilizzo di FTP per caricare l'aggiornamento del software applicativo di un'unità, oppure tramite un mini web-server effettuare la parametrizzazione di un'unità con un qualsiasi web browser presente praticamente su ogni PC, o altro ancora. Tutto questo sia all'interno dell'azienda stessa, sia all'esterno, grazie ad Internet.
Una completa integrazione fra una rete con TCP/IP (Ethernet) e CAN, è realizzabile unicamente se la comunicazione avviene in modo trasparente e senza alcun processo di gateway al punto di interconnessione. Per intenderci, ogni unità CAN intelligente che vuole comunicare con il mondo esterno deve implementare uno stack TCP/IP con i relativi servizi desiderati.
A questo punto è legittimo chiedersi se non sarebbe più semplice sostituire il bus di campo CAN con una rete Ethernet ? Sebbene si riscontra sempre più questa tendenza, le caratteristiche di Ethernet non rispondono alle esigenze di un sistema di controllo, in modo particolare alle problematiche legate alle esigenze di tempo reale, tipicamente nell'ordine dei mS. Usiamo quindi Ethernet per l'interconnessione globale, mentre per le aree specifiche affidiamoci a sistemi appositamente studiati. Se azzardiamo un paragone con l'ambiente PC, vediamo che per la comunicazione con le periferiche (Mouse, Scanner ecc.) non viene impiegato Ethernet, bensì USB o FireWire.
L'obiettivo è quindi la struttura rappresentata nella pagina seguente.

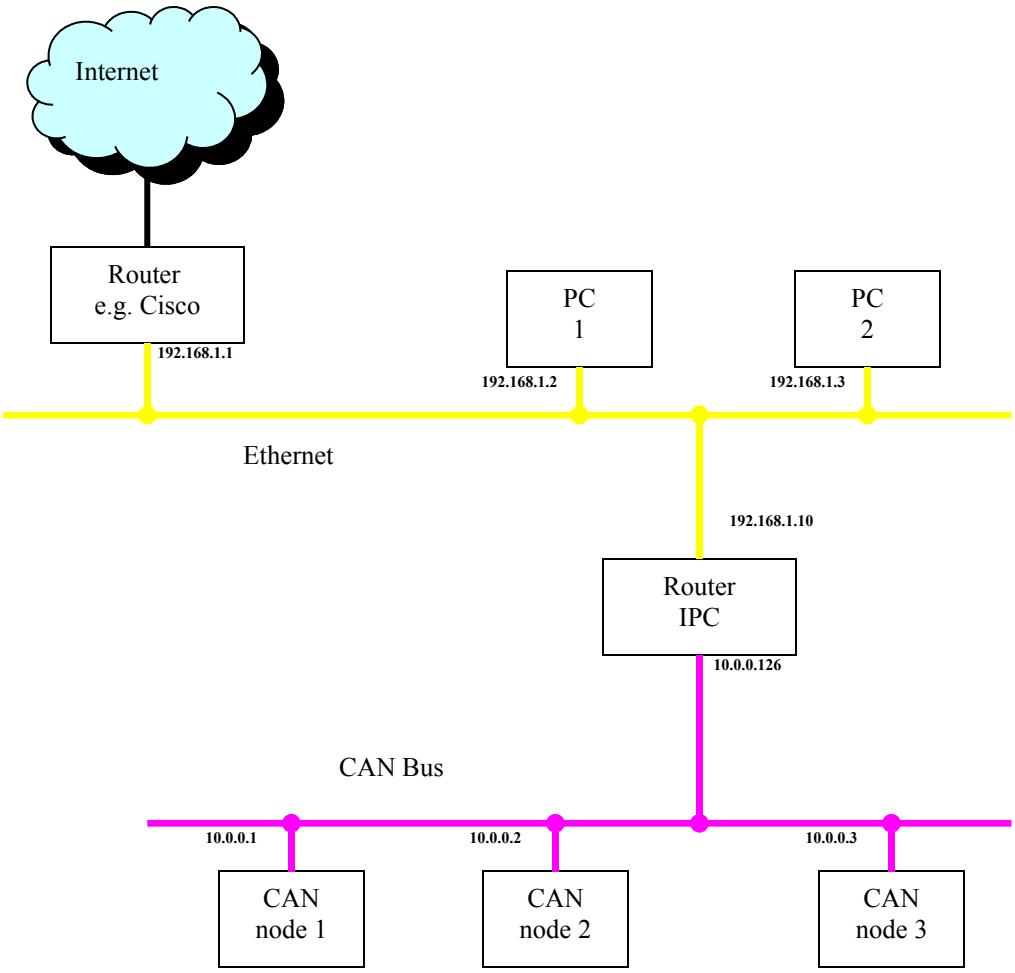

Internet
Router
e.g. Cisco
192.168.1.1
PC
1
192.168.1.2
PC
2
192.168.1.3
Ethernet
192.168.1.10
Router
IPC
10.0.0.126
CAN Bus
10.0.0.1
10.0.0.2
10.0.0.3
CAN
node 1
CAN
node 2
CAN
node 3

Figura 1

2. Introduzione

2.1 Obbiettivo

L'obbiettivo finale per questo lavoro è quello di poter monitorare e parametrizzare in tempo reale nodi CAN tramite browser WWW standard su stazioni di lavoro in rete locale. Infatti l'unico servizio realizzato in questo ambito è un mini Web-server.
E' importante sottolineare quanto sopra poiché come vedremo in seguito questo ha contribuito a determinare alcune scelte implementative.

Occorre comunque precisare che questo progetto non vuole essere un punto d'arrivo bensì di partenza. L'intera struttura è concepita al fine di facilitare ulteriori estensioni e complementi d'implementazione.

2.2 Ambiente di sviluppo

L'intero sviluppo è stato fatto su hardware Embedded basato sui microprocessori *Motorola* MC68xxx i quali comprendono un supporto integrato per lo sviluppo (Background Debug Mode). Il sistema BDM permette di leggere e/o modificare il contenuto di registri e memoria evitando così l'uso di debugger software.
Tutti i sorgenti sono compilati con i compilatori 'C' (v4.4) e Assembler (v7.0) della *Microtec Research*.

2.3 I Bus di campo

Nell'automazione industriale si fa sempre più uso di reti di comando e controllo, in particolare dei cosiddetti bus di campo. A dipendenza dei settori d'attività la tecnologia dei bus di campo può essere applicata al prodotto, oppure direttamente ai processi produttivi di un'azienda per essere poi integrata nel sistema informatico della stessa.
Di bus di campo ne esistono naturalmente diversi tipi, CAN, Profibus, Interbus, ecc. Saranno poi le specificità dei diversi prodotti a determinarne la scelta da parte dell'utilizzatore.

2.3.1 Caratteristiche

I bus di campo offrono meccanismi che consentono in modo rapido e affidabile, lo scambio di strutture di dati anche complesse.
I vantaggi principali dei bus di campo si possono così riassumere:

- facilità d'estensione
- flessibilità di configurazione
- riduzione massiccia dei cavi di collegamento
- riduzione globale dei costi

Con qualche svantaggio…

- maggior complessità dei dispositivi
- problemi di compatibilità fra prodotti di fornitori diversi

2.4 Il Bus di campo CAN

Il CAN (Controller Area Network) è un sistema di comunicazione seriale ad alte prestazioni, concepito originariamente quale bus per automobili nell'ambito della collaborazione tra la ditta Bosch e la Intel (1983-1985). Lo scopo dichiarato era quello di ridurre drasticamente il cablaggio elettrico nei veicoli, aumentando del frattempo sia la flessibilità di configurazione sia la sicurezza del trasporto delle informazioni.
La validità di questo sistema non ha tardato a dimostrarsi anche per soluzioni a problemi legati all'automazione industriale, ambiente nel quale si è rapidamente diffuso.

2.4.1 Caratteristiche

- Sistema di trasmissione seriale di tipo broadcast/multicast
- Trame di 8 byte con identificatori a 11 bit
- Arbitrazione basata sulla priorità del messaggio e quindi deterministico
- Flessibilità di configurazione
- Detezione e segnalazione d'errori a livello hardware
- Ritrasmissione automatica di messaggi corrotti
- Distinzione d'errori sporadici o permanenti

2.4.2 Modello OSI

La norma CAN definisce unicamente i livelli OSI 1 e 2 (*Physical Layer* e *Data Link Layer*). Per il livello 7 (*Application Layer*) esistono diverse alternative, non ancora standardizzate. Tra queste, le più diffuse sono il CAL (*CAN Application Layer*) definito dalla CiA (www.can-cia.de) e CANopen, un subset di CAL, pure definito dalla CiA. Va comunque detto che molte applicazioni utilizzano protocolli proprietari.

2.4.3 Introduzione a CANopen

CANopen è stato realizzato sotto forma di *profili*, per definire un meccanismo standardizzato di comunicazione tra apparecchi basati su CAN e, allo stesso tempo, definire la funzionalità di alcuni di essi.
Il concetto centrale di CANopen è basato sull'uso di un dizionario di oggetti (variabili, parametri…). Questo dizionario raccoglie i dati relativi alla comunicazione e all'applicazione. Nel dizionario sono appunto elencati tutti i parametri nonché la loro descrizione, il tipo di dati, la struttura ed il loro indirizzo, relativi ad una determinata applicazione.
Per accedere a questi dati sono previsti due meccanismi:

- **PDO – Process Data Object** : Canale usato per il trasferimento di dati relativi al processo.
- **SDO – Service Data Object** : Canale usato per il trasferimento di dati di servizio.

2.4.3.1 Caratteristiche di PDO:

- Trasferimento di dati Real-time (deterministico)
- Messaggi sincroni e Event-driven
- Identificatori CAN con alta priorità
- Ottimizzato per scambi veloci
- Trasmissione senza conferma di ricezione
- Massimo 8 byte per trasferimento
- Il formato della trama deve essere negoziato con il partner di comunicazione

2.4.3.2 Caratteristiche di SDO:

- Trasferimenti di dati che non necessitano di una relazione con il tempo
- Messaggi asincroni
- Identificatori CAN con bassa priorità
- Ottimizzato per il trasferimento di grandi quantità di dati
- Trasmissione con conferma di ricezione
- Trasferimento frammentato su più trame CAN
- Il formato delle trame usa un sistema di indicizzazione per riferirsi ai campi dati presenti nel dizionario

3. Protocolli

Questa sezione vuole essere una breve riassunto del formato delle trame dei diversi protocolli utilizzati.

3.1 Ethernet

I protocolli Ethernet e IEEE 802.3 hanno un formato delle rispettive trame molto simile fra di loro e possono co-esistere senza problemi sulla stessa LAN.

3.1.1 Formato Ethernet 2

Dest. address	Source address	Type	Data	CRC
6	6	2	46 – 1500	4

3.1.2 Formato IEEE 802.3

Dest. address	Source address	Len.	Dsap	Ssap	cntl	Org code	type	Data	CRC
6	6	2	1	1	1	3	2	38 – 1492	4

Come possiamo notare ambedue i formati usano un indirizzamento di destinazione e sorgente a 48-bit, compatibili fra di loro (indirizzi MAC). Il campo successivo è differente nei due formati. I'Ethernet *type* identifica il campo dati, mentre l'802.3 *Length* indica quanti bytes seguono fino al campo CRC (escluso). Fortunatamente, questi campi permettono la distinzione dei formati poiché nessuna lunghezza valida per il campo 802.3 *Length* risulta essere un *type* valido per Ethernet.
Il driver Ethernet per il router IPC è in grado ricevere trame in ambedue i formati, mentre per la trasmissione viene usato unicamente il formato Ethernet 2.

3.2 ARP

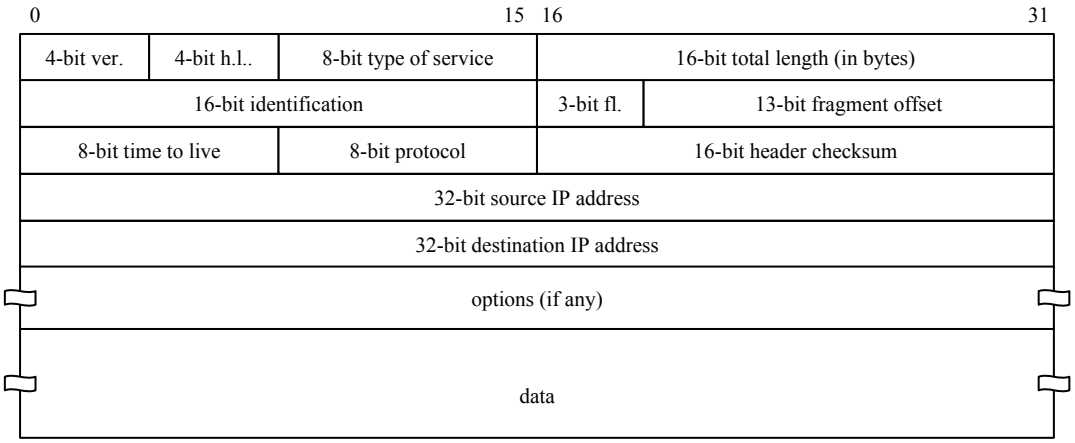
L'Address Resolution Protocol serve come noto a mappare gli indirizzi MAC con gli indirizzi IP.

Hard type	Prot type	Hard size	Prot size	Op	Sender Ethernet addr.	Sender IP addr.	Target Ethernet addr.	Target IP addr.
2	2	1	1	2	6	4	6	4

Hard type : Specifica il tipo di indirizzamento hardware (1 per Ethernet)
Prot type : Specifica il tipo di protocollo i qui indirizzi sono mappati (0x800 per IP)
Hard size : Numero di bytes usati dall'indirizzamento hardware
Prot size : Numero di bytes usati dall'indirizzamento del protocollo (6 per Ethernet e 4 per IP)
Op : Tipo di operazione (1= Arp request, 2= ARP reply, 3=RARP request, 4=RARP reply)

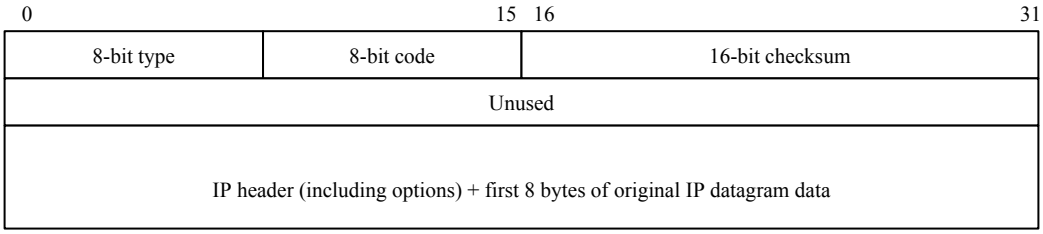
3.3 IP

Definizione dell'header IP.



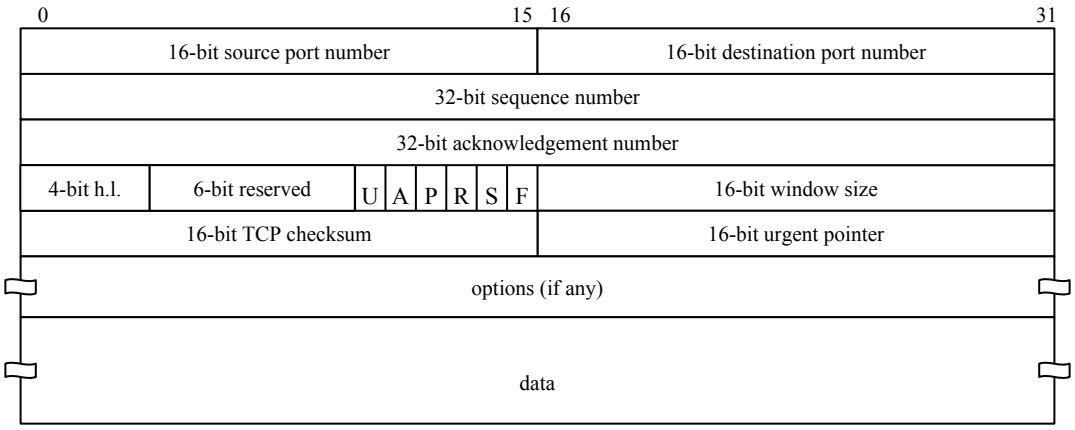
3.4 ICMP

Definizione dell'header ICMP.
Il campo *Unused*, a dipendenza del messaggio può assumere significato.



3.5 TCP

Definizione dell'header TCP.



4. Struttura globale

Essenzialmente l'intero progetto si suddivide in tre parti

1. trasporto di datagrammi IP su bus CAN
2. realizzazione di un router Ethernet – CAN
3. implementazione di un mini Web-server su nodo CAN

riassunti dallo schema sotto rappresentato.

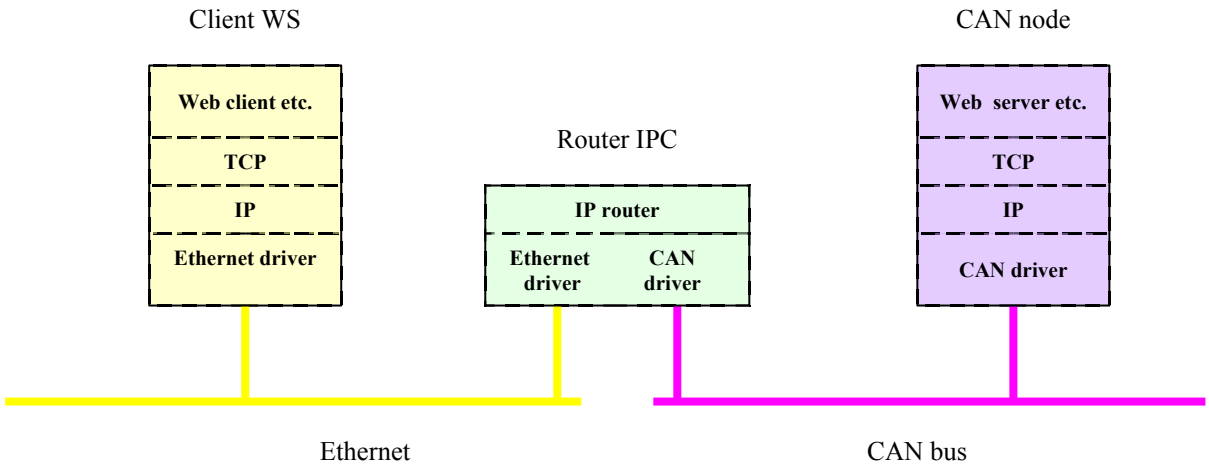


Figura 2

L'intero assieme di moduli sviluppati in questo progetto va pensato come un pacchetto di rete utilizzabile su ogni sistema embedded avente la necessità del supporto di rete.

4.1 Trasporto di datagrammi IP su CAN

Il problema principale di IPoCAN sta nel fatto che una trama CAN è composta da 8 Byte (pay-load). L'RFC di IP specifica che ogni modulo IP deve poter inoltrare datagrammi di almeno 68 Byte (IP header <= 60byte + frammentazione min. 8 Byte. Di conseguenza non è possibile incanalare direttamente IP su CAN bensì occorre passare attraverso un processo di frammentazione invisibile a IP.

La soluzione adottata consiste nell'implementazione del processo di frammentazione all'interno del livello applicativo di CAN (CanApp.c), come mostrato nella Figura 3 alla pagina seguente.
Il processo di frammentazione, eseguito in background dal livello CanApp, risulta così trasparente per IP. Quest'ultimo vede un canale di trasmissione con un MTU, nel caso specifico, di 1500 Byte (come per Ethernet).

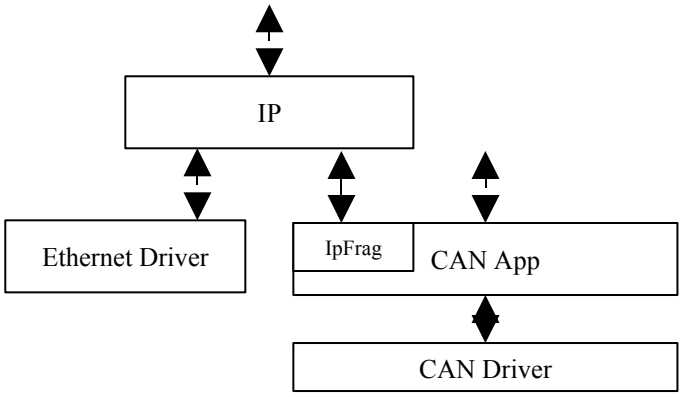
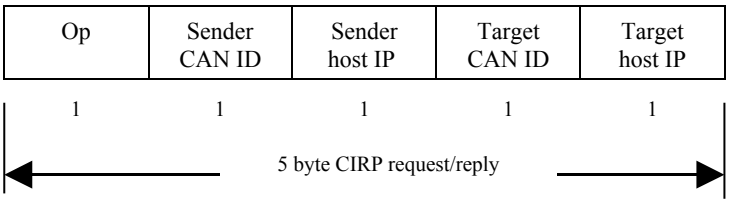


Figura 3

Un altro problema da risolvere, analogamente ad Ethernet, riguarda la mappatura degl'indirizzi IP con gli identificatori CAN. Una soluzione potrebbe essere l'implementazione di un meccanismo simile ad ARP.
Analizzando un po' più in dettaglio il problema vediamo che entro certe condizioni come questo non sia strettamente necessario.

Gli indentificatori CAN sono composti da 11-bit i quali permettono teoricamente un massimo di 2048 nodi.
Diversamente da Ethernet però gli identificatori CAN non servono unicamente ad indirizzare un nodo fisico, bensì determinano il tipo e la priorità del singolo messaggio. Attenendoci allo standard CANopen avremo quindi un massimo di 127 nodi CAN.
Se a livello di IP, per lo stesso segmento di bus CAN escludiamo la possibilità di avere più indirizzi di rete (nessuna operazione di sub-netting), possiamo mappare direttamente la parte Host dell'indirizzo IP con l'indirizzo fisico del nodo.

Come accennato sopra, se una mappatura diretta tra indirizzi IP e identificatori CAN non è accettabile occorre implementare un protocollo simile ad ARP. In fase di sviluppo, al fine di dimostrarne il funzionamento, è stato implementato un protocollo (CIRP, CAN Identifier Resolution Protocol) che mantenendo la limitazione di 127 nodi CAN utilizza una sola trama CAN:



Il campo *op* specifica il tipo di operazione, 1 => CIRP request, 2 => CIRP reply.

4.2 Architettura router IPC

4.2.1 Hardware

Il cuore di IPC è una scheda basata sul processore *Motorola* MC68EN360 il quale integra un Ethernet controller.

4.2.2 Software

Il router IPC comprende una porta Ethernet a 10Mb e una porta CAN. La struttura modulare dell'intero sistema permette l'aggiunta di altre porte (es: una seconda porta CAN o un canale serie) senza troppi sforzi.
Le tabelle di routing di IPC sono implementate in maniera statica e configurabili via canale serie.
L'intero pacchetto utilizza i servizi di Softask-360, in particolare le code di comunicazione.

4.2.2.1 Elenco moduli

- Eth360.c , Ethernet.h : Driver internet
- Candrv.h : Definizione per driver CAN
- CanApp.c, CanApp.h : Application layer di CAN (implementa frammentazione di IP)
- Arp.c, Arp.h : Protocollo ARP
- Cirp.c, Cirp.h : Protocollo CIRP (opzionale)
- Ip.c, Ip.h : Implementazione protocollo IP
- Icmp.c, Icmp.h : Implementazione protocollo ICMP
- NetUtil.c, NetUtil.h : Network utilities
- Ripc.c, Ripc.h : Main
- Config.h : Configurazione dell'intero sistema

4.3 Architettura nodo CAN

4.3.1 Hardware

L'implementazione hardware dei nodi CAN è stata fatta sulla scheda GMS-332.

4.3.2 Software

A differenza del router IPC il nodo CAN implementa anche il protocollo TCP al fine di supportare la realizzazione di un mini Web-server. L'implementazione di tale Web server su un nodo CAN risulta essere il punto d'arrivo di tutto il progetto. Sebbene lo scopo finale è quello di permettere la visualizzazione di pagine di stato e/o l'impostazione di parametri su un nodo CAN, nell'ambito di questa documentazione verranno illustrate solamente alcune pagine senza particolare utilità. Questo in ragione della stretta dipendenza dal tipo di applicazione realizzata sul nodo CAN.
L'intero pacchetto utilizza i servizi di Softask-332, in particolare le code di comunicazione.

4.3.2.1 Elenco moduli

- Candrv.h : Definizione per driver CAN
- CanApp.c, CanApp.h : Application layer di CAN (implementa frammentazione di IP)
- Cirp.c, Cirp.h : Protocollo CIRP (opzionale)
- Ip.c, Ip.h : Implementazione protocollo IP
- Icmp.c, Icmp.h : Implementazione protocollo ICMP
- NetUtil.c, NetUtil.h : Network utilities
- Tcp.c, Tcp.h : Implementazione protocollo TCP
- http.c, http.h, html.h : Implementazione protocollo http e mini Web-server
- Config.h : Configurazione

5. Il sistema operativo Softask-3xx

Softask è nato con le esigenze dei diversi progetti sviluppati in Softool, non da meno questo stesso progetto. Inizialmente non era altro che una "Synchronous Control Loop". Il sistema era basato su una tabella di puntatori a funzioni le quali venivano chiamate in modo ciclico e sequenziale. Le singole funzioni (tasks) implementavano una macchina a stati per suddividere i diversi compiti e permettere l'esecuzione delle altre funzioni.
In seguito sono stati implementati meccanismi rudimentali per la segnalazione di eventi e assegnazione di priorità ai diversi tasks.
La versione attuale ha ben poco a che vedere con la versione originale. Implementato per i microprocessori Motorola 68xxx e CPU32 la versione attuale realizza un vero "preemptible real time multitasking system".
Caratteristiche principali:

- Kernel scritto al 80% in assembler
- Risorse, Rom <26KB, Ram 541 Bytes + 106 Bytes per task creato
- Fino a 32 tasks attivi contemporaneamente
- Sfrutta la modalità 'Supervisor / User' dei processori Motorola
    - Il Kernel gira in 'Supervisor mode'
    - I tasks girano in 'User mode'
- Ogni task ha una sua area di stack configurabile in run-time
- 1 gruppo di 24 eventi broadcast utilizzabili liberamente
- Fino a 32 code di comunicazione di taglia fissa configurabile in run-time
- Funzionalità di 'task lock'
- Arbitrazione risorse tramite tokens (semafori)

In questa documentazione è allegato il file 'stask.h' contenente tutte le dichiarazioni delle funzioni utilizzabile dall'utente.

6. Descrizione del software

Questa parte si occupa della descrizione schematica dell'implementazione dei vari protocolli. Ogni parte ha poi una breve descrizione delle funzioni disponibili (accessibili dagli altri moduli).
Nelle descrizioni che seguono si presuppone l'esatta conoscenza del lettore dei vari protocolli impiegati. Una loro descrizione esula da questo ambito e si rimanda alla bibliografia allegata.

La struttura di tutto il software permette la descrizione dei singoli moduli facendo astrazione dal tipo di applicazione (router IPC o nodo CAN). Infatti, per motivi di praticità tutto il software è stato sviluppato sull'hardware del router IPC e solo alla fine, con minimi cambiamenti a livello di 'include files', è stato portato anche sull'hardware del nodo CAN.

6.1 Struttura dei '*Memory Buffer*' MBuff

La struttura dei buffer di memoria è di cruciale importanza per il funzionamento dello stack TCP/IP.
Come noto, in questi sistemi i pacchetti di dati ricevuti/trasmessi devono transitare attraverso diversi livelli software (vedi modello OSI) prima di giungere a destinazione. Un punto fondamentale in termini di prestazioni di velocità sta nell'evitare un'operazione di copia dei dati per ogni livello attraversato. In questa realizzazione infatti, fra i diversi livelli viene passato solamente un puntatore.
Per realizzare quanto sopra occorre avere a disposizione una struttura la quale permette l'allocazione dei buffers dinamicamente.

L'implementazione in questo progetto consiste in un pool di buffer statici i quali vengono allocati/liberati dinamicamente. La dimensione del pool ed il numero di buffers a disposizione viene deciso in compilazione tramite il file config.h.
La scelta di utilizzare buffers statici, più dispendiosi in risorse di memoria, è stata presa al fine di evitare problemi di frammentazione della memoria, tipici dell'allocazione dinamica. Va sottolineato che molte applicazione embedded non possono permettersi "pause" per la deframmentazione della memoria.
L'approccio 'misto', ossia l'uso di buffers statici ma con dimensioni diverse è stato scartato per ragioni di semplicità.

La struttura di buffers è implementata in un file:
NetUtil.c (.h)

6.1.1 Descrittore MBuffer

Ogni Mbuff nel pool ha un suo descrittore, la cui struttura è la seguente:

```
typedef struct {
  UCHAR Data[MBUFFER_SIZE];  /* Data buffer               */
  USHORT Status;             /* TRUE = busy, FALSE = free */
  void *DataPtr;             /* Data pointer              */
  short Length;              /* Data field length         */
  USHORT Type;               /* Protocol type             */
} T_MBuff;

static T_MBuff MBuffPool[SIZE_BUFF_POOL];
```

SIZE_BUFF_POOL (config.h) definisce il numero massimo di MBuffers sopportati.
MBUFFER_SIZE (config.h) definisce la taglia dei buffers.

6.1.2 Funzioni disponibili

void InitMBuff(void)

Commento
Esegue l'inizializzazione dell'intero pool di buffers. Ogni buffer è liberato ed azzerato.

T_MBuff *GetMBuff(void)

Risultato
Ritorna un puntatore al primo buffer libero trovato nel pool o NULL se tutto occupato.

Commento
Il buffer viene segnato come occupato e DataPtr punta alla prima locazione del buffer.
GetMBuff() può essere chiamata anche da un interrupt handler.

void FreeMBuff(T_MBuff **MBuffPtr*)

Parametri
**MBuffPtr*
Puntatore all'MBuffer da liberare

Commento
Libera il buffer puntato da MBuffPtr.
Si comporta correttamente anche se MBuffPtr = NULL.
FreeMBuff() può essere chiamata anche da un interrupt handler.

6.2 Driver Ethernet EN-360

La completa comprensione del driver Eth360 presuppone l'approfondita conoscenza del microprocessore MC68EN360 e in particolar modo del controller Ethernet integrato. A questo scopo si rimanda al 'MC68360 USERS'S MANUAL' di *Motorola*.
Il controller Ethernet legge i dati da trasmettere, rispettivamente scrive quelli ricevuti in buffers locati liberamente in memoria, nel nostro caso negli MBuffers. Ogni MBuffer usato è referenziato da un BD (Buffer Descriptor del MC68360). La particolarità di questa architettura sta nel fatto che si possono allocare diversi BD per la trasmissione/ricezione, formando così una coda circolare di lunghezza programmabile.
Per tenere traccia dell'associazione fra i singoli BD e MBuffer occorrono due tabelle (una per la coda di ricezione e l'altra per la coda di trasmissione) le quali, con i loro puntatori, realizzano un'immagine delle code circolari gestite dal controller Ethernet.

L'intero pacchetto è contenuto in due files:
Ethernet.h
Eth360.c

L'implementazione delle code immagine è la seguente:

```
typedef struct {          /* Keeps track of the BD<->MBuffer association  */
  volatile T_BD *BdPtr;   /* SCC1 Buffer Descriptor pointer               */
  T_MBuff *MBuffPtr;      /* Memory buffer pointer                        */
} T_BdQueue;

static T_BdQueue Eth_RxQueue[MAX_RX_BUF]; /* Ethernet Rx queue    */
static T_BdQueue Eth_TxQueue[MAX_TX_BUF]; /* Ethernet Tx queue    */

static T_BdQueue *RxQPtr;                 /* Rx queue pointers    */
static T_BdQueue *TxQPtr;                 /* Tx queue pointers    */
```

MAX_RX_BUF e MAX_TX_BUF (file config.h) definiscono la lunghezza delle code di ricezione e trasmissione.

La struttura sotto riportata rappresenta l'interfaccia hardware. Contiene l'indirizzo MAC della porta e l'IP associato.

```
typedef struct {
  UINT Ip;                /* IP address   */
  UCHAR Mac[MAC_LEN];     /* Ethernet MAC address   */
} T_EthInterface;

T_EthInterface IfEth;     /* Hardware interface   */
```

La comunicazione con il livello superiore (Network layer) avviene in maniera differente per la trasmissione e per la ricezione.
Il Network layer chiama la funzione *Eth_Send(..)* per la trasmissione mentre per la ricezione si affida alla coda di comunicazione fornita da Softask. Gli MBuffers contenenti le trame ricevute vengono inseriti nella coda di comunicazione dall'interrupt handler *Eth_Isr()*. Quest'ultimo, provvede pure all'eliminazione dell'header Ethernet (Ethernet 2 o IEEE 802.3).
La variabile *Type* dell'MBuffer contiene il protocollo destinatario.

Di seguito è riportato lo schema della comunicazione:

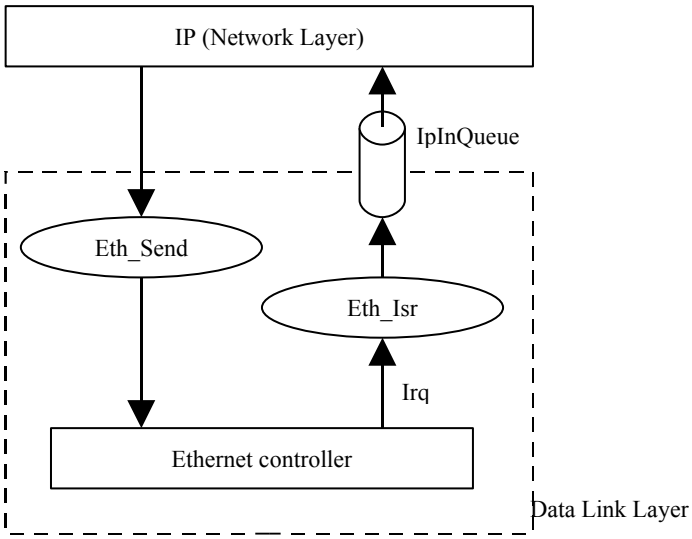


Figura 4

6.2.1 Funzioni disponibili

void Eth_Init(void)

Commento
Esegue l'inizializzazione dell'Ethernet controller.
Crea le code di ricezione/trasmissione con le rispettive immagini ed assegna l'indirizzo MAC (preso da IfEth).

USHORT Eth_ClearError(void)

Risultato
Valore 16bit contenente i bit di errore segnalati dal driver.

Commento
Usata per richiedere lo stato della comunicazione. Eventuali errori vengono azzerati.

short Eth_SendFrame(T_MBuff **MBuffPtr*, UCHAR **MacPtr*, USHORT *Type*)

Parametri
**MBuffPtr*
Puntatore all'MBuffer da trasmettere
**MacPtr*
Puntatore all'indirizzo MAC di destinazione
Type
Tipo di dati inviati (ARP, IP..)

Risultato
Ritorna OK se termina con successo o ERROR se il BD in cima alla coda non era libero.

Commento
Invia il contenuto di un Mbuffer.
La funzione aggiunge l'header Ethernet al contenuto di MBuffer, per cui quest'ultimo deve avere sufficientemente spazio libero (14 Bytes).
L'MBuffPtr viene associato al BD in cima alla coda e quello precedentemente assegnato (e quindi trasmesso) viene liberato.

6.3 Comunicazione CAN

La comunicazione CAN si suddivide in due blocchi principali:

- CAN Driver i82527 (CanDrv.c)
- CAN application (CanApp.c)

Va sottolineato che questa struttura è una derivazione da un'implementazione esistente (usata in varie applicazioni), opportunamente modificata per adattarla alle esigenze di di questo progetto. In modo particolare si è implementato il processo di frammentazione necessario al trasporto di IP.

Tutto il pacchetto utilizza i seguenti moduli:

82527.h
CanDrv.c (.h)
CanApp.c (.h)

Qui sotto è rappresentata un visione schematica della struttura di comunicazione tra CAN e il livello IP:

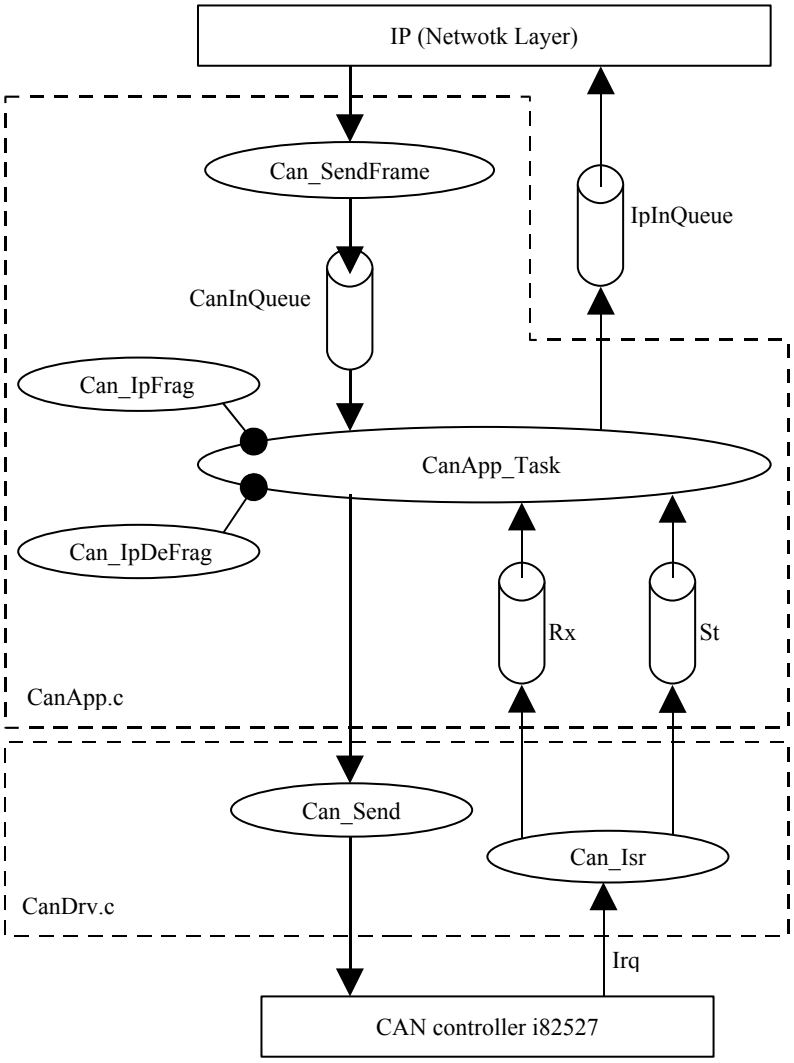


Figura 5

6.3.1 Driver CAN i82527

Questo modulo mette a disposizione un pacchetto di funzioni per la comunicazione e la gestione del CAN controller Intel i82527. L'architettura del driver fornisce il supporto all'implementazione di protocolli di livello superiore, come CANopen.
Un'analisi dettagliata del driver CAN esula dall'ambito di questa descrizione poiché come gia accennato non si tratta di un'implementazione realizzata specificatamente per questo progetto. Inoltre, per un'esatta comprensione occorrerebbero approfondite conoscenze del CAN controller i82527. Ci limiteremo quindi ad una breve descrizione dell'interfacciamento con *CanApp_Task()*.

Come vediamo dalla Figura 5, la comunicazione verso il livello superiore avviene tramite le due code di CanApp. La prima coda (Rx - *CanRxQueue*) riporta tutte le trame CAN ricevute, mentre la seconda (St - *CanStQueue*) riporta informazioni di stato generate dal controller CAN (Bus warning, Bus Off, Message Lost). Entrambe le code sono alimentate dall'interrupt handler *Can_Isr()* il quale legge le trame CAN dal controller e ne cattura i cambiamenti di stato.
In questa versione inoltre, l'interrupt handler *Can_Isr()* gestisce anche l'avanzamento del processo di frammentazione in trasmissione (trasmissione 'interrupt driven').

Per l'invio di trame CAN abbiamo a disposizione la seguente funzione:

USHORT Start_Trans(short *CanN*, char *Ext*, T_CanDataMsg **Pqmsg*)

Parametri
CanN
Identificatore chip CAN (0 o 1, il driver sopporta 2 CAN controller)
Ext
Tipo di indirizzamento utilizzato. 0 = standard (11-bit), 1 = extended (29-bit)
**Pqmsg*
Puntatore a T_CanDataMsg contenente la trama da inviare e il 'MessageBox' da utilizzare.

Risultato
Ritorna ERROR se il 'Message Box' è occupato, altrimenti OK.

Commento
Invia il campo dati contenuto nella struttura puntata da ***Pqmsg* in una trama CAN.

6.4 Livello CanApp

CanApp realizza il livello applicativo di CAN. Come gia accennato, non c'è una norma standardizzata per il livello applicativo di CAN. Questa implementazione risulta essere compatibile con CANopen per quello che riguarda la mappatura degli identificatori CAN ma non implementa lo scambio di informazioni dei dizionari d'oggetti. Per contro lo scambio tra i nodi di messaggi specifici dell'applicazione, avviene usando i messaggi PDO di CANopen. Questa scelta permette di realizzare nodi CAN che possono convivere con sistemi basati su CANopen, pur essendo nel contempo "leggeri" dal punto di vista delle risorse e nella complessità d'implementazione.
La frammentazione dei datagrammi IP, necessaria per il loro trasporto su CAN, avviene pure a questo livello. Infatti come vediamo in Figura 5, il livello IP comunica esclusivamente con CanApp.
In questo ambito analizzeremo in dettaglio il processo di frammentazione. Si presuppone pertanto la conoscenza del protocollo CAN poiché il tutto si basa su alcune sue peculiarità.
E' comunque bene ricordare che, sebbene non documentato in questa sede, nell'applicazione reale il bus CAN non serve solamente a trasportare IP bensì principalmente allo scambio di messaggi relativi al tipo di applicazione.

Per poter transitare sul bus CAN, i datagrammi IP devono essere suddivisi in pacchetti di grandezza inferiore o uguale a 8 Bytes. Al fine di ridurre al minimo l'impatto sulle prestazioni forzatamente degradate si è cercato di minimizzare il più possibile l'overhead introdotto dalla frammentazione. A questo proposito sono state fissate le seguenti condizioni:

1. Massimo 127 nodi per segmento CAN (CANopen)
2. Il concetto di connessione affidabile è delegato ad un livello superiore (TCP)
3. La frammentazione in contemporanea di due o più datagrammi IP fra due nodi distinti non è permessa
4. La frammentazione in contemporanea di due o più datagrammi IP provenienti da due o più nodi verso un terzo, è permessa

Il processo di frammentazione, una volta avviato, prosegue autonomamente fino al termine del trasferimento. La trasmissione dei singoli pacchetti avviene tramite interrupt (interrupt driver).

Formato delle trame CAN per la frammentazione

Processo di frammentazione

Implementazione

Di seguito sono rappresentati i tipi di messaggi che transitano nella code di comunicazione verso *CanApp_Task()*.

```
typedef packed struct {    /* STATUS MESSAGES, 8 byte size      */
  USHORT Cod;              /* message code                      */
  USHORT Str;              /* status register                   */
  USHORT Mob;              /* message object number             */
  USHORT Mobst;            /* Ctrl. 0,1  merged by Mob_Status   */
  UCHAR  Chip;             /* Chip number                       */
} T_CanStMsg;
```

T_CanStMsg contiene messaggi di stato generati dal CAN controller ed inviati dall'interrupt handler *Can_Isr()*.
Contiene una copia dello 'Status Register' del controller nonché lo stato del rispettivo 'Message Box'.

```
typedef packed struct {    /* DATA MESSAGES, 16 byte size       */
  USHORT Cod;              /* message code                      */
  UCHAR  Dlc;              /* data length code                  */
  UCHAR  Mob;              /* message object number             */
  UINT  Id;                /* CAN Identifier                    */
  UCHAR  Data[8];          /* data field                        */
} T_CanDataMsg;
```

T_CanDataMsg contiene le trame CAN accettate dal controller, anch'esso inviato da *Can_Isr()*.

```
typedef struct {
  UCHAR DestId;            /* CAN node destination ID   */
  T_MBuff *MBuffPtr;       /* MBuff pointer             */
} T_CanIpMsg;
```

T_CanIpMsg contiene i messaggi inviati dal livello IP tramite la coda ad esso riservata.

La struttura sotto riportata rappresenta l'interfaccia hardware. Contiene l'indirizzo del nodo e l'IP associato.
A meno che non venga utilizzato il protocollo CIRP, i campi *Ip* e *Id* contengono lo stresso valore.

```
typedef struct {
  UINT Ip;                 /* IP address        */
  UINT Id;                 /* CAN identifier    */
} T_CANInterface;
```

6.4.1 Funzioni disponibili

short Can_SendFrame(T_MBuff **MBuffPtr*, UCHAR *CanId*, USHORT *Type*)

Parametri
**MBuffPtr*
Puntatore all'MBuffer da trasmettere
CanId
Indirizzo del nodo CAN di destinazione
Type
Tipo di dati inviati (CIRP_TYPE o IP_TYPE)

Risultato
ERROR se la coda (CanInQueue) di comunicazione verso *CanApp_Task()* è piena oppure se il parametro *Type* specifica un protocollo non valido, altrimenti OK.

Commento
Chiamata dal livello IP.
Type = IP_TYPE
Crea un messaggio del tipo T_CanIpMsg e lo inserisce nella coda (CanInQueue) di CanApp_Task(). Usata per inviare un datagramma IP al nodo specificato con *CanId*.
Type = CIRP_TYPE
Invia un 'CIRP request' al nodo specificato con *CanId*.

void CanApp_Task(void)

Commento
CanApp_Task() è il cuore della comunicazione CAN. Agisce da 'hub' instradando i diversi messaggi provenienti dal driver i82527 e da/verso i vari servizi di livello superiore, e implementando parzialmente lo standard CANopen.
Nella fase d'inizializzazione esegue le seguenti operazioni:

- Inizializza il CAN controller i82527
- Crea le code di comunicazione *CanRxQueue*, *CanStQueue* e *CanInQueue*
- Apre i 'Message Box' per la ricezione:
  - 1 MsgBox per i messaggi PDO
  - 1 MsgBox per i messaggi NMT (Network Message T…..)
  - 1 MsgBox per i messaggi EMERGENCY
- Apre i 'Message Box' per la trasmissione
  - 1 MsgBox per la frammentazione di IP
  - 1 MsgBox per tutti gli altri messaggi

Terminata la fase d'inizializzazione, il task si mette in ascolto interrogando ciclicamente le code con un periodo di 100mS, nell'ordine: *CanStQueue*, *CanInQueue CanRxQueue*.
Il motivo per cui vengono aperti due 'Message Box' è dovuto al conflitto di risorse che può avvenire al momento dell'invio di un qualsiasi messaggio durante il processo di frammentazione che, come abbiamo visto, è eseguito tramite interrupt.

6.5 ARP - Implementazione

ARP (Address Resolution Protocol) gestisce la traslazione degl'indirizzi IP (32-bit) con i corrispondenti indirizzi MAC, in questo caso Ethernet (48-bit).
Quando un host vuole inviare un datagramma IP ad un altro host, innanzitutto cerca l'indirizzo MAC dell'host di destinazione nella ARP cache (*T_ArpCache ArpCache[..]*), tabella dove vengono mappati gli indirizzi IP con i rispettivi indirizzi MAC. Se l'indirizzo richiesto viene trovato il datagramma IP può essere inviato. Diversamente, se non esiste una corrispondenza le funzioni di ARP "congelano" il datagramma IP (*T_PostedFrame ArpFreezer[..]*), inviano un 'ARP request', e, quando un 'ARP reply' corrispondente è stato ricevuto inviano il datagramma IP.
La copia di indirizzi MAC - IP ricevuti con 'ARP request/reply' viene poi memorizzata nella ARP cache così che sarà disponibile ad IP per futuri scambi. Ogni record nella ARP cache resta valido per 2 min. dall'ultima volta che IP lo ha richiesto.

Come vediamo dal diagramma in Figura 6, il livello IP interagisce con ARP al fine di ottenere gli indirizzi MAC necessari al trasporto su Ethernet.

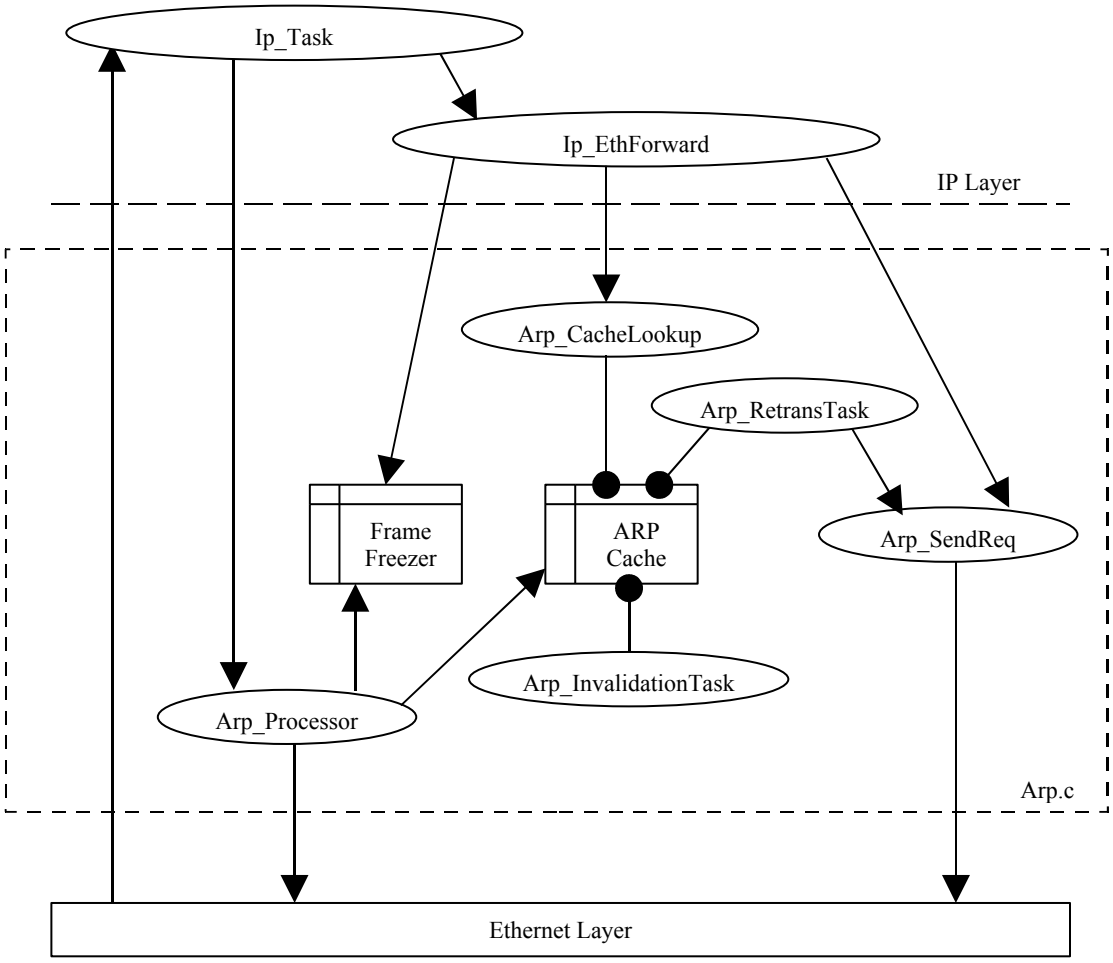


Figura 6

L'intero protocollo è contenuto in due files:
Arp.h
Arp.c

L'implementazione della cache di ARP è la seguente:

```
typedef struct {
  UCHAR Status;           /* Entry status         */
  UCHAR ExpTimer;         /* Entry expiring timer */
  USHORT Retry;           /* Request timeout      */
  UINT Ip;                /* IP address           */
  UCHAR Mac[MAC_LEN];     /* MAC address          */
} T_ArpCache;

T_ArpCache ArpCache[ARP_CH_SIZE];    /* ARP cache      */
```

I valori possibili per la variabile *Status* sono:
INVALID : Record non valido
BOOKED : Record riservato, in attesa di un ARP reply
VALID : Record valido

ARP_CH_SIZE (config.h) definisce la dimensione della cache.

I datagrammi IP messi in attesa di un ARP reply sono memorizzati nel seguente modo:

```
typedef struct {
  T_MBuff *MBuffPtr;      /* MBuff containing the IP datagram    */
  UINT GWayIp;            /* Gateway IP address (next-hop)       */
  short Status;           /* TRUE = busy, FALSE = free           */
} T_PostedFrame;

static T_PostedFrame ArpFreezer[ARP_FREEZER_SIZE];   /* Queue of posted datagrams */
```

ARP_FREEZER_SIZE (config.h) definisce la dimensione della cache.

6.5.1 Funzioni disponibili

void ARP_CacheInvalidation_Task(void)

Commento
Task periodico a bassa priorità usato per invalidare la cache.
Ogni 15sec. spazzola l'intera cache e per ogni record segnato come VALID, decrementa la variabile *ExpTimer*.
Quando quest'ultima è uguale a 0, il record è segnato come INVALID.

short ARP_SetCacheEntry(UINT *IpAddr*, UCHAR **MacPtr*)

Parametri
IpAddr
Indirizzo IP a 32-bit
**MacPtr*
Puntatore all'indirizzo MAC

Risultato
ERROR se non ottiene l'*ARP_TOKEN* ,altrimenti OK.

Commento
Memorizza la copia di indirizzi in un record libero della cache segnandolo come VALID.
Se l'intera cache risulta occupata, viene soprascritto il record con il valore più basso della variabile *ExpTimer*.
L'accesso alla cache è arbitrato da Softask tramite la richiesta del token *ARP_TOKEN*.

short ARP_BookCacheEntry(UINT *IpAddr*)

Parametri
IpAddr
Indirizzo IP a 32-bit

Risultato
ERROR se non ottiene l'*ARP_TOKEN* ,altrimenti OK.

Commento
Memorizza l'indirizzo IP in un record libero segnandolo come BOOKED.
Se l'intera cache risulta occupata, viene soprascritto il record con il valore più basso della variabile *ExpTimer*.
L'accesso alla cache è arbitrato da Softask tramite la richiesta del token *ARP_TOKEN*.

T_ArpCache *ARP_CacheLookUp(UINT *IpAddr*)

Parametri
IpAddr
Indirizzo IP a 32-bit

Risultato
Ritorna un puntatore al record contenente l'indirizzo IP richiesto, altrimenti NULL.

Commento
Ricerca nella cache il record (segnato come VALID) corrispondente all'indirizzo IP richiesto.
La variabile *ExpTimer* viene ricaricata con ARP_ENTRY_TOUT.
L'accesso alla cache è arbitrato da Softask tramite la richiesta del token *ARP_TOKEN*.

void ARP_Processor(T_MBuff **MBuffPtr*)

Parametri
**MBuffPtr*
Puntatore all'MBuffer contenente la trama ARP.

Commento
Gestisce i messaggi ARP (reply/request) provenienti dal driver Ethernet.
Nel caso di un 'ARP_request' corrispondente al proprio IP, memorizza nella cache gli indirizzi dell'host remoto ed invia una risposta contenente il proprio indirizzo MAC.
Nel caso di un 'ARP_reply' aggiorna la cache, ricerca nel *ArpFreezer[..]* datagrammi con l'indirizzo IP corrispondente così da inviarli all'host remoto.

short ARP_SendRequest(UINT *TargetIp*)

Parametri
TargetIp
Indirizzo IP di destinazione

Risultato
ERROR se non è possibile allocare un MBuffer, altrimenti OK.

Commento
Invia un 'ARP_request' contenente l'indirizzo *TargetIp*.

void ARP_Retransmission_Task(void)

Commento
Gestisce il processo di ritrasmissione di richieste ARP senza risposta.
L'intero processo è implementato come task, il quale normalmente resta addormentato (suspended). All'invio di un ARP request (*ARP_SendRequest(..)*) viene svegliato e finché nella ARP cache trova uno o più records con la variabile *Status* = BOOKED resta sveglio controllando ciclicamente (200mS) lo *Status* di ogni record. Se non vi è stata nessuna risposta dopo 5.5 sec. viene inviato un altro ARP request, così come 30 sec. più tardi. Se dopo 31 sec. non è stata ricevuta alcuna risposta i datagrammi in attesa in *ArpFreezer[..]* vengono eliminati.
Quando non vi sono più richieste in attesa il task si rimette nello stato addormentato.

6.6 CIRP - Implementazione

CIRP (Can Identifier Resolution Protocol) come il suo omologo ARP, serve per la traslazione degl'indirizzi IP (32-bit) con i corrispondenti indirizzi CAN (per CANopen, 7-bit).
Come abbiamo gia visto al punto "Trasporto di datagrammi IP su CAN" il suo uso è facoltativo, in questa sede è stato implementato a titolo dimostrativo. Le trame transitano con l'identificatore NMT di CANopen e con il codice 255, il quale non risulta essere utilizzato da CANopen.
L'implementazione di CIRP segue esattamente gli stessi schemi di ARP, per cui per la sua descrizione si rimanda alla descrizione di ARP. L'unico punto differente sta nel formato degli indirizzi, al posto degli indirizzi MAC a 48-bit vengono usati gli indirizzi CAN a 7-bit.
Di seguito sono elencati le dichiarazioni delle funzioni.

6.6.1 Funzioni disponibili

void CIRP_CacheInvalidation_Task(void)
short CIRP_SetCacheEntry(UINT *IpAddr***, UCHAR** *CanId***)**
short CIRP_BookCacheEntry(UINT *IpAddr***)**
UCHAR *CIRP_CacheLookUp(UINT *IpAddr***)**
void CIRP_Processor(char *CanN***, T_CanDataMsg ***RxMsgPtr***)**
short CIRP_SendRequest(UINT *TargetIp***)**
void CIRP_Retransmission_Task(void)
T_PostedFrame *CIRP_GetFreezerEntry(void)

6.7 IP - Implementazione

Come ben sappiamo il ruolo primario di IP (Internet Protocol) è di instradare i datagrammi verso le rispettive destinazioni. Il suo compito è di cruciale importanza per tutti i protocolli e le applicazioni di livello superiore.
Con questa versione, che implementa IPv4, si è realizzato una implementazione "light" di IP pur mantenendone le sue funzionalità vitali. I punti sacrificati sono i seguenti:

- Il contenuto del campo TOS (Type-of-service) è ignorato
- Il campo delle opzioni è pure ignorato (per altro raramente utilizzato)
- Il processo di frammentazione non è implementato, solo il riassemblaggio di datagrammi frammentati è implementato
- Il processo di routing avviene per mezzo di tabelle statiche (nessun protocollo di routing)

Per evitare il processo di frammentazione, i livelli superiori (TCP e applicazione) faranno in modo di inviare pacchetti di dati che non superino l'MTU delle interfacce implementate, ossia <= 1500 Bytes (compresi gli headers dei vari livelli).

Qui sotto abbiamo una rappresentazione schematica della comunicazione di IP con i vari livelli:

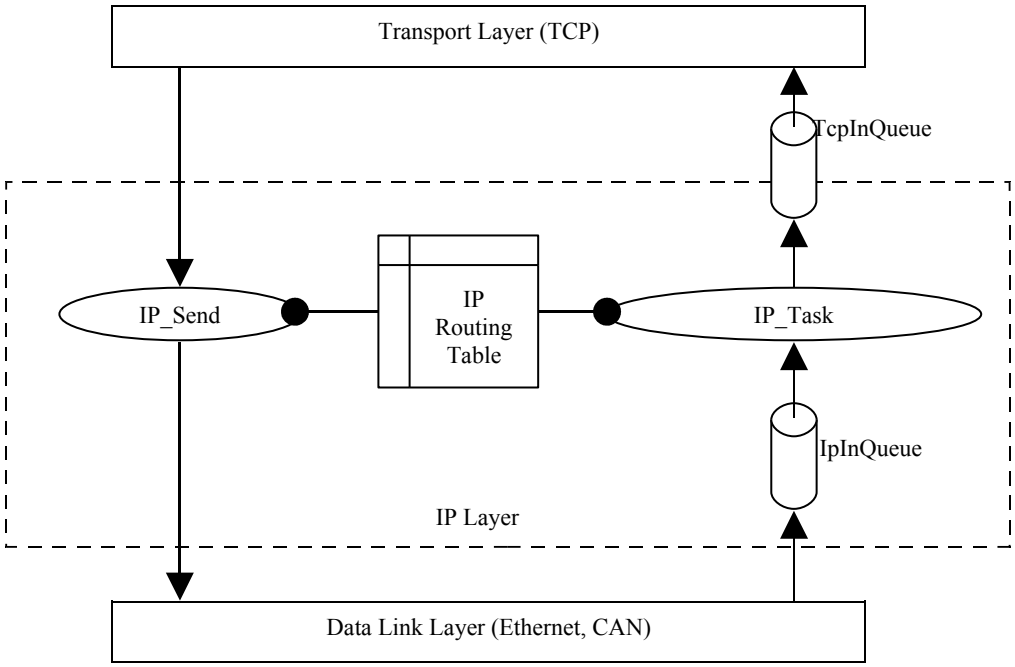


Figura 7

Come vediamo dallo schema in Figura 7, *IP_Task()* usa le code di comunicazione di Softask per la ricezione dal Data Link Layer e per la trasmissione verso il livello superiore (flusso asincrono). *IP_Send(..)* invece fornisce l'interfaccia al livello superiore per l'invio di dati (flusso sincrono). Sia *IP_Send(..)* che *IP_Task()* si affidano alla tabella di routing per le decisioni d'instradamento.
Anche se non risulta dallo schema, IP fa naturalmente capo ai servizi di ICMP per l'invio/ricezione di messaggi di controllo.

L'intero protocollo è contenuto in tre files:

Ip.h
Ip.c
Netuil.c (*Ip_CalcChecksum (..)*)

La coda di input del task IP è dichiarata come segue:

```
IpInQueue = sc_qcreate(IP_QUEUE_LEN,sizeof(T_MBuff *));
```

Tabella di Routing

L'implementazione della tabella di routing è la seguente:

```
typedef struct {        /* ROUTE DESCRIPTOR               */
  UINT DestIp;          /* Destination IP address         */
  UINT Mask;            /* Subnet mask                    */
  UINT GWayIp;          /* Gateway IP address (next-hop)  */
  UCHAR Flags;          /* Route entry flags              */
  UCHAR If;             /* Interface type                 */
} T_RouteDesc;

T_RouteDesc RouteTab[ROUTE_TAB_SIZE];   /* Routing table   */
```

ROUTE_TAB_SIZE (config.h) definisce il numero massimo di route.

Il campo *If* specifica il tipo d'interfaccia, può avere i seguenti valori:

IF_LO0	: Local loopback
IF_ETH0	: Ethernet interface 0
IF_CAN0	: CAN interface 0
IF_CAN1	: CAN interface 1

Il campo Flags può avere tre flags:

U	: record attivo (route is Up)
H	: il campo DestIp indica un Host
G	: il record indica un Gateway

Come esempio riportiamo le tabella usata durante lo sviluppo:

Router IPC

Note	Destinazione	Maschera	Gateway	Flags	Interfaccia
(1)	192.168.1.12	255.255.255.255	192.168.1.12	UH	IF_ETH0
(2)	10.0.0.126	255.255.255.255	10.0.0.126	UH	IF_CAN0
(3)	192.168.1.0	255.255.255.224	192.168.1.12	U	IF_ETH0
(4)	10.0.0.0	255.255.255.128	10.0.0.126	U	IF_CAN0
(5)	127.0.0.0	255.255.255.255	127.0.0.1	U	IF_LO0
(6)	0.0.0.0	0.0.0.0	192.168.1.1	UG	IF_ETH0

Nodo CAN

Note	Destinazione	Maschera	Gateway	Flags	Interfaccia
(2)	10.0.0.1	255.255.255.255	10.0.0.1	UH	IF_CAN0
(4)	10.0.0.0	255.255.255.128	10.0.0.1	U	IF_CAN0
(5)	127.0.0.1	255.255.255.255	127.0.0.1	UH	IF_LO0
(6)	0.0.0.0	0.0.0.0	10.0.0.126	UG	IF_CAN0

Legenda note

1. Interfaccia di rete Ethernet
2. Interfaccia di rete CAN
3. Rete Ethernet
4. Rete CAN
5. Interfaccia di loopback
6. Default gateway

Tutte le decisione di routing sia nel router IPC che nei nodi CAN, avvengono in base alle tabelle sopra rappresentate e secondo la strategia seguente:

1. Ricerca di un indirizzo Host corrispondente (UH)
2. Ricerca di un indirizzo di rete corrispondente (U e (Addr & Mask) == DestIp)
3. Ricerca di un default gateway (UG)

Tabella di riassemblaggio

```
typedef struct {          /* REASSEMBLY DESCRIPTOR       */
  UINT SrcIp;             /* For ident. purpose (RFC791) */
  UINT DestIp;            /* For ident. purpose (RFC791) */
  USHORT Id;              /* For ident. purpose (RFC791) */
  UCHAR PCol;             /* For ident. purpose (RFC791) */
  USHORT Tm;              /* Timeout timer               */
  T_MBuff *HeadMBuffPtr;  /* Header MBuffer pointer      */
  T_MBuff *DataMBuffPtr;  /* Data MBuffer pointer        */
} T_ReasDesc;

T_ReasDesc ReasTab[REAS_TAB_SIZE];/* IP Reassembly table  */
```

REASS_TAB_SIZE (config.h) definisce quanti processi di riassemblaggio si possono avere contemporaneamente.

I campi *SrcIp*, *DestIP*, *Id*, *PCol* servono per identificare i singoli frammenti. *HeadMBuffPtr* è un puntatore all'header del primo frammento della serie (quello con *Offset* = 0).
Il timer *Tm* serve come timeout per liberare le risorse in caso di frammenti mancanti. Viene aggiornato dalla funzione *Ip_UpdateTimers()*, la quale viene chiamata dall'interrupt periodico di sistema.

6.7.1 Funzioni disponibili

Accesso alla tabella di routing

void Ip_InitRouteTable(void)

Commento
Inizializza la tabella di routing.
Mette tutti i campi a 0, ad eccezione *Mask* che è messo a 255.255.255.255

T_RouteDesc *Ip_GetRoute(UINT *ReqIp*)

Parametri
ReqIp
Indirizzo IP richiesto

Risultato
Ritorna il puntatore alla route corrispondente se esiste, altrimenti NULL.

Commento
Cerca nella tabella una route corrispondente all'indirizzo IP richiesto. L'algoritmo di ricerca usa nell'ordine le seguenti regole di precedenza:

1. Ricerca di un indirizzo di Host corrispondente
2. Ricerca di un indirizzo di rete corrispondente
3. Ricerca di un default gateway

Un indirizzo di Host corrispondente ha sempre la precedenza su un indirizzo di rete il quale ha la precedenza su un default gateway.

short Ip_AddRoute(T_RouteDesc **RoutePtr*)

Parametri
**RoutePtr*
Puntatore al descrittore contenente le impostazioni da aggiungere alla tabella

Risultato
ERROR se la tabella è tutta occupata ,altrimenti OK.

Commento
Aggiunge la route puntata da *RoutePtr* alla tabella di routing.

short Ip_RemoveRoute(UINT *DestIp*, UINT *Mask*, UINT *GWayIp*)

Parametri
DestIp
Indirizzo IP a 32-bit
Mask
Maschera di subnet
GWayIp
Indirizzo IP del gateway da usare

Risultato
ERROR se non è stata trovata una route con le caratteristiche specificate, altrimenti OK.

Commento
Cerca nella tabella di routing una route con le caratteristiche specificate e la rimuove.

Interfacciamento con IP

void Ip_Send(UINT *SrcIp*, UINT *DestIp*, T_MBuff **MBuffPtr*)

Parametri
SrcIp
Indirizzo IP sorgente, inserito nel campo 'Source IP address' dell'header IP
DestIp
Indirizzo IP di destinazione, inserito nel campo 'Destination IP address' dell'header IP
**MBuffPtr*
Puntatore all'MBuffer contenente i dati da inviare

Commento
Questa è la funzione d'interfaccia per i livelli superiori (flusso sincrono). Viene chiamata da ICMP e TCP per l'invio dati.
Aggiunge l'header IP ai dati contenuti nell'MBuffer e spedisce il tutto chiamando la funzione interna *Ip_Routing(..)* che, in base alle informazioni raccolte nella tabella di routing, chiamerà a sua volta le funzione d'invio delle rispettive interfacce (*Ip_xxxForward(..)*). Quest'ultime hanno pure il compito di calcolare il *checksum*. dell'header.
Se chiamata con il parametro *SrcIp = 0*, lo stesso verrà impostato dalla funzione *Ip_Routing(..)*.
L'header aggiunto è il normale header IP (20 Bytes) senza opzioni, con il campo *TOS = 0* e il campo *TTL* impostato a 32.

void Ip_Task(void)

Commento
Questo è il task che gestisce il flusso di informazioni proveniente dal livello inferiore (flusso asincrono). Assieme con *Ip_Send(..)* rappresenta l'interfaccia del livello IP (Network Layer).
Nella sua fase d'inizializzazione crea i tasks di servizio per i protocolli ARP e CIRP (se attivi) ed apre la sua coda di ricezione (*IpInQueue*). Terminata l'inizializzazione resta in ascolto sulla coda di ricezione e in base al tipo di protocollo esegue le seguenti operazioni:

IP_TYPE
- Verifica il checksum
- Chiama la funzione *Ip_Routing(..)*

ARP_TYPE
- Chiama la funzione *Apr_Processor(..)* se ARP è attivato

CIRP_TYPE
- Chiama la funzione *Cirp_Processor(..)* se CIRP è attivato

void Ip_UpdateTimers(void)

Commento
Funzione di servizio che implementa i timeout per i processi di riassemblaggio.
Deve essere chiamata dall'interrupt periodico di Softask.

Il seguente diagramma illustra le relazioni fra le diverse funzioni appartenenti al modulo IP:

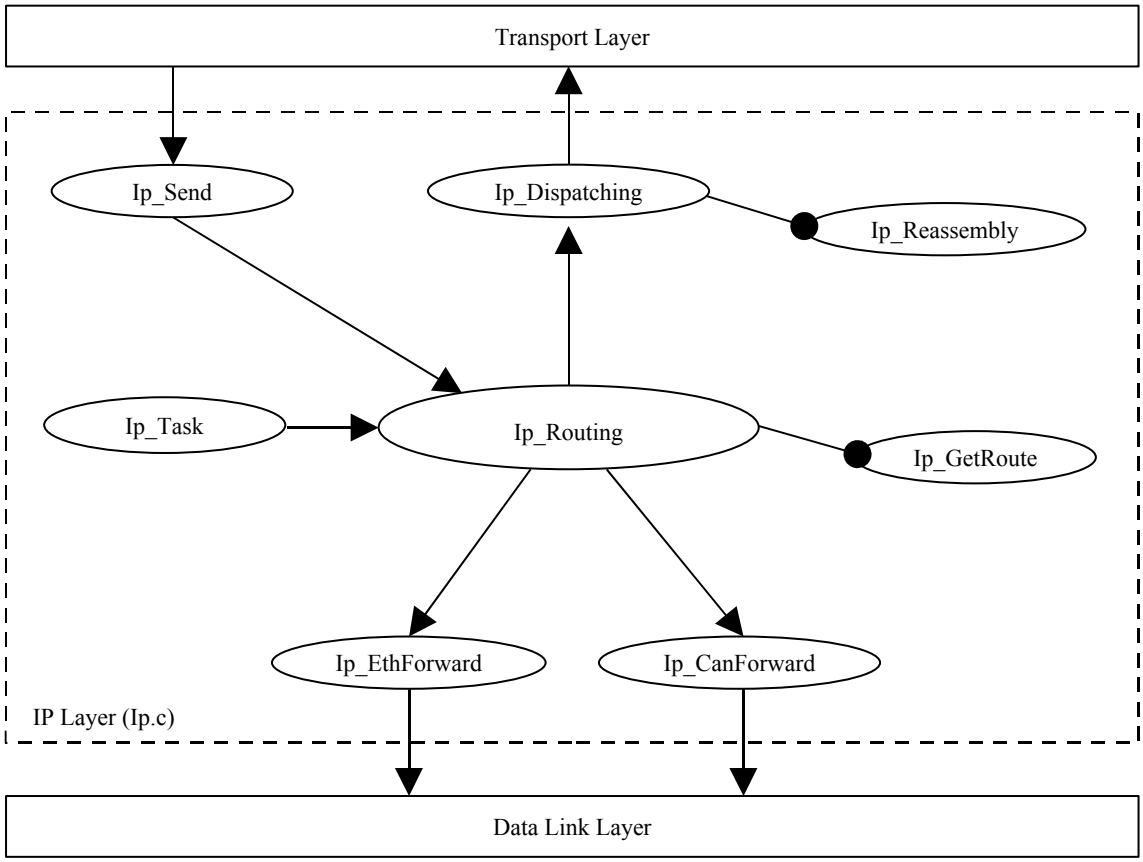


Figura 8

6.8 ICMP - Implementazione

ICMP (Internet Control Message Protocol) è un protocollo di servizio per lo stack TCP/IP. ICMP è usato per comunicare errori o messaggi di servizio tra sistemi IP ed è parte integrante di ogni implementazione IP.
Attualmente sono implementati i seguenti messaggi:

ICMP_ECHO
ICMP_ECHOREPLY
ICMP_DEST_UNREACH
    ICMP_HOST_UNREACH
    ICMP_PROT_UNREACH
ICMP_TIME_EXCEEDED

L'intero protocollo è contenuto in tre files:

Icmp.h
Icmp.c
Netuil.c (*Ip_CalcChecksum (..)*)

6.8.1 Funzioni disponibili

void Icmp_Receive(UINT *SrcIp*, UINT *DestIp*, T_MBuff **MBuffPtr*)

Parametri

SrcIp
: Indirizzo IP sorgente

DestIp
: Indirizzo IP di destinazione

**MBuffPtr*
: Puntatore all'MBuffer contenente i dati ricevuti

Commento

Gestisce i messaggi ICMP ricevuti.
Controlla il checksum del messaggio ed esegue le operazioni legate al tipo di messaggio. Attualmente solamente i messaggi ICMP_ECHO/REPLY sono implementati. Per ICMP_TIME_EXCEEDED e ICMP_DEST_UNREACH viene visualizzato un messaggio sullo stdout.

void Icmp_SendMessage(UCHAR *Type*, UCHAR *Code*, UINT *Param*, T_MBuff **MBuffPtr*)

Parametri

Type
: Tipo di messaggio, corrisponde al campo *Type* nell'header ICMP

Code
: Codice del messaggio, corrisponde al campo *Code* nell'header ICMP

Param
: Opzionale, dipende dal tipo di messaggio. Anch'esso contenuto nell'header ICMP

**MBuffPtr*
: Puntatore all'MBuffer contenente il datagramma che ha generato il messaggio ICMP

Commento

Genera un messaggio ICMP.
Usato per inviare quei messaggi ICMP che necessitano di contenere l'header IP più 8 Bytes del datagramma causante la generazione del messaggio.

void Ping_Test(UINT *DestIp***, UINT** *nSend***)**

Parametri
DestIp
Indirizzo IP di destinazione
nSend
Numero di ripetizioni.

Commento
Implementa una semplice versione del programma PING.
Genera un messaggio di ICMP_ECHO con 26 Bytes di dati all'Host specificato con *DestIp* ed attende per 3 secondi una sua risposta con ICMP_ECHOREPLY..

6.9 TCP - Implementazione

TCP (Transmission Control Protocol) è il servizio di trasporto usato dalle applicazioni quali Telnet, http, Ftp ecc. E' usuale definire TCP come un "reliable, connection-oriented, byte stream service".
Sebbene, come per IP, si è cercato di realizzare una versione semplificata, la sua complessità a livello d'implementazione non è cosa di poco conto poiché al fine di ottenere una versione utilizzabile, buona parte delle caratteristiche di TCP non possono essere tralasciate. La difficoltà nell'implementare un protocollo come TCP sta nel fatto che TCP cerca di fare parecchie cose contemporaneamente:

- Apertura di una connessione ('Passive Opening' e 'Active Opening')
- Trasmissione dei dati in full-duplex
- Gestire i datagrammi persi, duplicati e/o fuori sequenza
- Gestire problemi sulla rete
- Implementare il controllo di flusso
- Gestire quantità diverse di dati
- Chiusura della connessione (anche 'Half closure')

Tutte queste operazioni vengono inoltre eseguite su diverse connessioni, per cui occorre tenere un'immagine dello stato di ognuna di esse.

Nell'ambito di questo progetto, il cui scopo finale è la realizzazione di un mini Web-server (protocollo HTTP) e non di meno per questioni di tempi di realizzazione, alcune parti di TCP sono state tralasciate. In particolare:

- 'Active Opening' non interamente implementato (da terminare). Basta il processo di 'Passive Open'
- L'unica opzione (header TCP) supportata è l'MSS (Maximum Segment Size)
- 'Urgent Mode' non implementato
- Diversi algoritmi d'ottimizzazione non implementati (Nagle, 'Congestion avoidance', 'Slow start', 'Keepalives')

L'intero protocollo è contenuto in tre files:
Tcp.h
Tcp.c
Netuil.c (*Tcp_CalcChecksum (..)*)

TCP tiene traccia delle applicazioni in ascolto per mezzo della tabella:

```
typedef struct {
  USHORT ServerPort;            /* Server port       */
  USHORT ServerQueueId;         /* Server queue Id   */
} T_ServerMap;

static T_ServerMap ServerMap[MAX_SERVER_APP];
```

MAX_SERVER_APP (config.h) definisce quanti servizi possono essere aperti contemporaneamente.

Al tentativo di apertura di una connessione da parte di un Host remoto (Passive open), TCP controlla se la porta di destinazione contenuta nell'header TCP corrisponde ad un record della tabella *ServerMap[]*. Se trova una corrispondenza la connessione viene accettata, altrimenti viene rifiutata. Le applicazioni che si vogliono registrare su TCP useranno la funzione *OpenService(..)*.
Se una connessione è accettata viene allocato un descrittore di socket (Socket TCP Control Block) nella tabella:

```
static T_SockTcb SocketTcb[MAX_SOCKET];
```

MAX_SOCKET (config.h) definisce quante connessioni possono essere aperte simultaneamente.

Ogni descrittore contiene tutte le informazioni inerenti alla connessione. Sia TCP che le applicazioni useranno poi le funzioni d'accesso ai sockets per aprire/chiudere le connessioni o inviare dati.

A differenza di IP, il livello TCP usa una coda di comunicazione sia per la ricezione che per la trasmissione. La coda di comunicazione è dichiarata come segue:

```
typedef struct {
  T_SockTcb *SockPtr;        /* Socket pointer              */
  USHORT SockCmd;            /* Socket Command (SOCK_CMD)   */
  UINT SrcIp;                /* IP source address           */
  UINT DestIp;               /* IP Destination address      */
  T_MBuff *MBuffPtr;         /* MBuffer pointer             */
} T_TcpQMsg;

TcpInQueue = sc_qcreate(TCP_QUEUE_LEN,sizeof(T_TcpQMsg));
```

Il campo *SockCmd* può assumere i seguenti valori:

- TCP_OPEN Inviato dall'applicazione, apre una connessione ('Active Open' non completato)
- TCP_CLOSE Inviato dall'applicazione, chiude una connessione
- TCP_SEND Inviato dall'applicazione, per trasmettere dei dati
- TCP_RECV Inviato da IP, indica la ricezione di dati

Qui sotto abbiamo una rappresentazione schematica della comunicazione di TCP con i vari livelli:

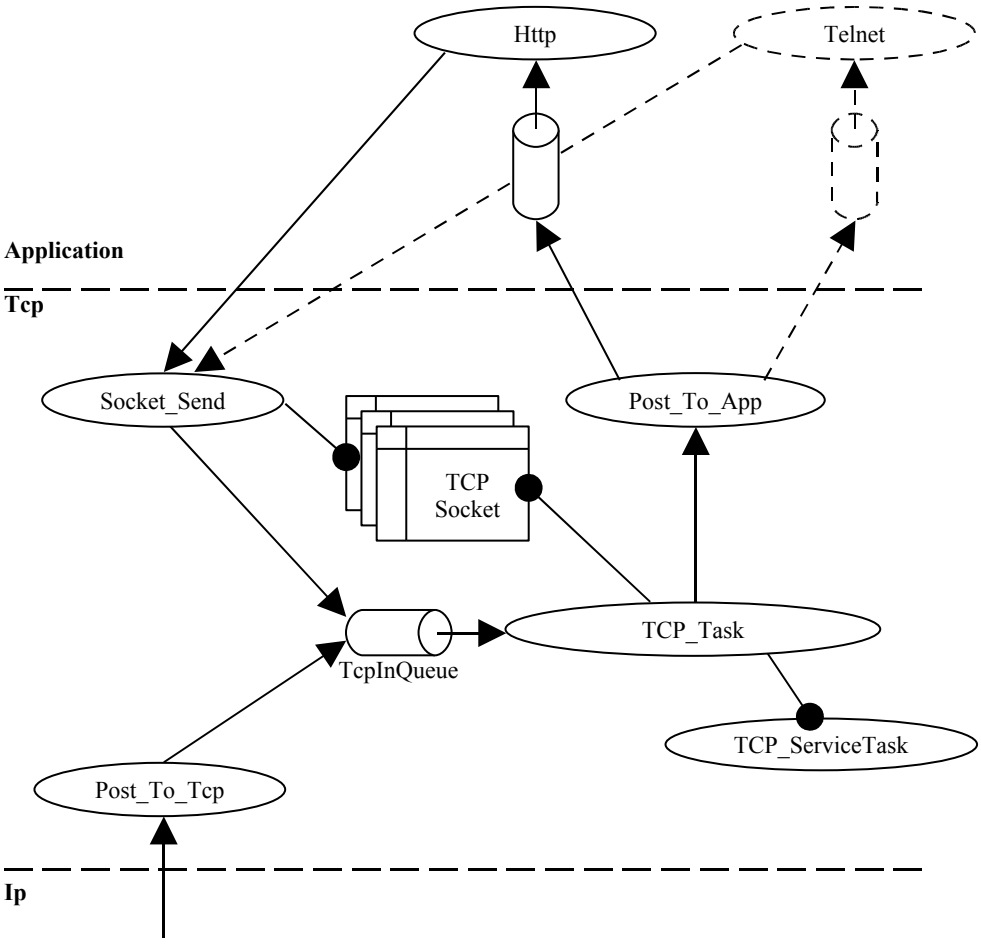


Figura 9

6.9.1 Funzioni disponibili

USHORT OpenService(USHORT *Port*, USHORT *AppQueueId*)

Parametri
Port
Numero della porta del servizio
AppQueueId
Identificatore (Softask) della coda di comunicazione

Risultato
Ritorna ERROR se il servizio non può essere aperto poiché la tabella *ServerMap[]* è piena, altrimenti OK.

Commento
Apre il servizio associato a *Port* (Telnet = 23, HTTP = 80 ecc.). TCP accetterà le connessioni in 'Passive open' su questa porta.

USHORT CloseService(USHORT *Port*)

Parametri
Port
Numero della porta del servizio

Risultato
Ritorna ERROR se la porta specificata con *Port* non è stata trovata, altrimenti OK.

Commento
Chiude il servizio associato a *Port* precedentemente aperto con *OpenService(..)*. TCP rigetterà i tentativi di connessione su questa porta.

T_SockTcb *Socket_Open(UINT *RemIp*, USHORT *RemPort*, UINT *LocIp*, USHORT *LocPort*, USHORT *AppQueueId*)

Parametri
RemIp
Indirizzo IP dell'Host remoto
RemPort
Numero della porta utilizzata dall'Host remoto
LocIp
Indirizzo Ip dell'Host locale
LocPort
Numero della porta del servizio locale a cui ci si vuole connettere
AppQueueId
Identificatore (Softask) della coda di comunicazione associata al servizio specificato con *LocPort*.
Se AppQueueId = 0, l'identificatore viene preso dalla *ServerMap[]* (Passive opening).

Risultato
Ritorna NULL se non ci sono risorse libere, altrimenti un puntatore al descrittore contenuto nella tabella *SocketTcb[]*.

Commento
Alloca ed inizializza le risorse necessarie all'apertura di una connessione ma non la apre.

void Socket_Close(T_SockTcb *_SockPtr_)

Parametri
*_SockPtr_
Puntatore al descrittore del socket

Commento
Chiude il socket puntato da *SockPtr* e libera tutte le risorse ad esso associate.

USHORT Socket_Send(T_SockTcb *_SockPtr_, USHORT _Cmd_, T_MBuff *_MBuffPtr_)

Parametri
*_SockPtr_
Puntatore al descrittore del socket
Cmd
Comando da inviare:
- TCP_OPEN Apre una connessione ('Active Open', parzialmente implementato)
- TCP_CLOSE Chiude una connessione ('Active Close')
- TCP_SEND Invia i dati contenuti nell'MBuffer

*_MBuffPtr_
Puntatore ad un MBuffer. A dipendenza di *Cmd*, può essere NULL.

Risultato
Ritorna ERROR se la coda di TCP è piena, altrimenti OK.

Commento
Invia un comando (*Cmd*) al socket puntato da *SockPtr*.
Costruisce un messaggio T_TcpQMsg con le informazioni sopra elencate e lo inserisce nella coda di TCP (*TcpInQueue*).

USHORT GetSocket_State(T_SockTcb *_SockPtr_)

Parametri
*_SockPtr_
Puntatore al descrittore del socket

Risultato
Stato della connessione.

Commento
Ritorna lo stato (corrispondente al diagramma di transizione di TCP) della connessione descritta da *SockPtr*.

USHORT Post_To_Tcp(UINT _SrcIp_, UINT _DestIp_, T_MBuff *_MBuffPtr_)

Parametri
SrcIp
Indirizzo IP sorgente
DestIp
Indirizzo IP di destinazione
*_MBuffPtr_
Puntatore all'MBuffer contenente i dati

Risultato

Ritorna ERROR se la coda di TCP è piena, altrimenti OK.

Commento
Chiamata dal livello IP per passare il datagramma a TCP.
Costruisce un messaggio T_TcpQMsg con le informazioni sopra elencate e lo inserisce nella coda di TCP (*TcpInQueue*).

USHORT Post_To_App(T_SockTcb **SockPtr*)

Parametri
**SockPtr*
Puntatore al descrittore del socket

Risultato
Ritorna ERROR se la coda dell'applicazione è piena, altrimenti OK.

Commento
Invia all'applicazione (registrata con *OpenService(..)*) i dati ricevuti da TCP.
Costruisce un messaggio T_TcpQMsg con le informazioni sopra elencate e lo inserisce nella coda dell'applicazione.

void Tcp_Task(void)

Commento
Questo è il task principale di TCP. Gestisce il flusso di informazioni proveniente dai livelli adiacenti.
TCP_Task() si avvale di un task di servizio (*TCP_ServiceTask()*, vedi Figura 9) il quale implementa le seguenti funzionalità:

- Ritrasmissione di datagrammi "Not Acknowledged"
- Trasmissione di datagrammi posticipati
- "Delayed acknowledge"
- "Window advertising"
- Timeout connessione

TCP_ServiceTask() viene eseguito ciclicamente in background con un periodo di 200mS. Il valore di 200mS è imposto dalla specifica per il meccanismo di "Delayed acknowledge".

Nella fase d'inizializzazione crea il task di servizio ed apre la sua coda di ricezione (*TcpInQueue*). Terminata l'inizializzazione resta in ascolto sulla coda di ricezione. I messaggi letti da quest'ultima vengono discriminati in base al contenuto del campo *SockCmd* di T_ TcpQMsg. Il livello IP, per mezzo di *Post_To_App(..)*, imposta *SockCmd = TCP_RECV*, mentre le applicazioni, tramite *Socket_Send(..)*, impostano TCP_OPEN, TCP_CLOSE o TCP_SEND a dipendenza dell'operazione richiesta.
Il diagramma a pagina seguente (Figura 10) illustra il flusso delle operazioni di *TCP_Task()*.

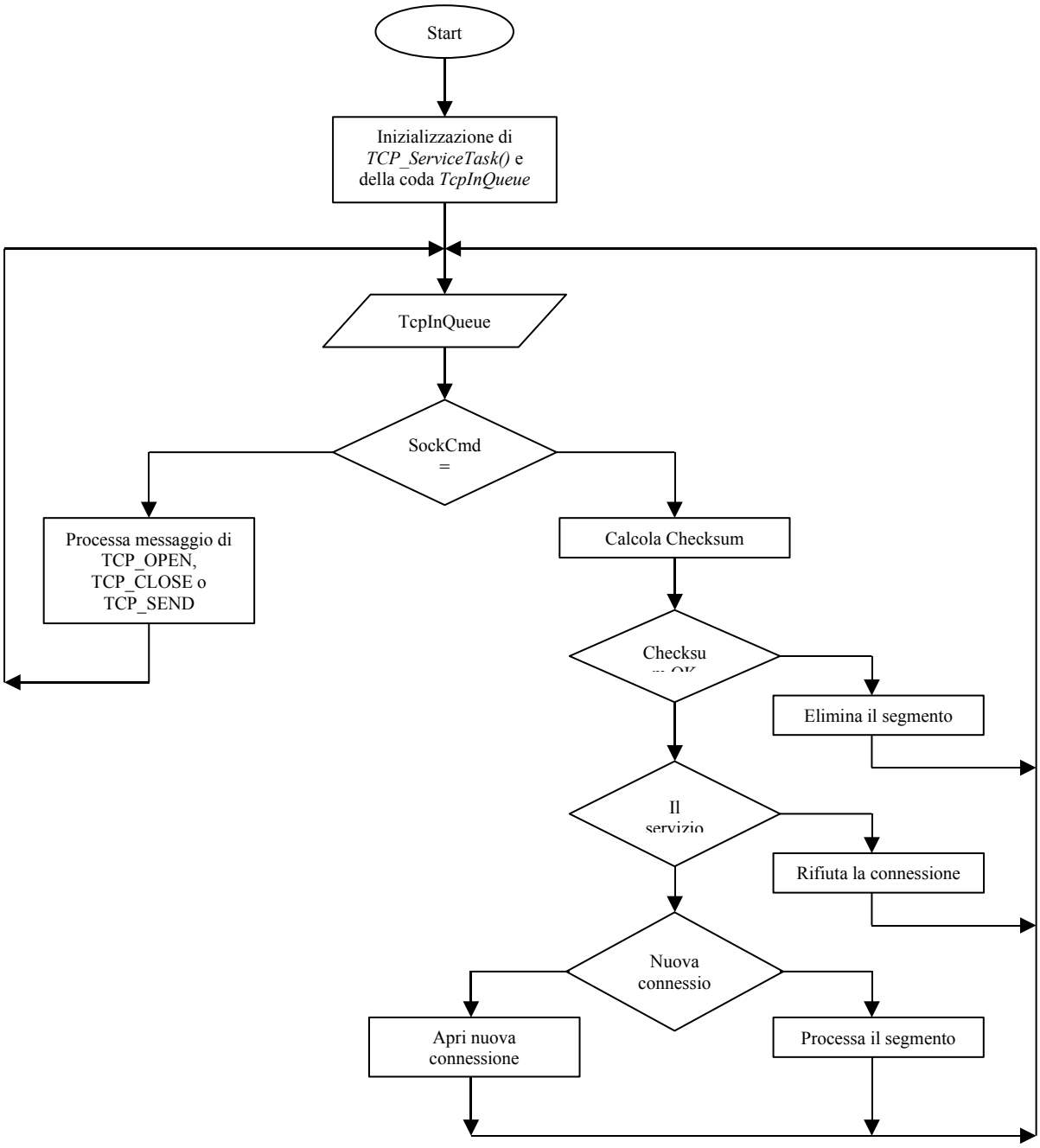


Figura 10 - flusso delle operazioni di *TCP_Task()*

6.10 HTTP - Implementazione

HTTP (HyperText Transfer Protocol) è il protocollo usato per la comunicazione con i Web-servers.
Per la realizzazione del mini Web-server in questo progetto si è scelto d'implementare la versione 1.0 di HTTP, in ragione delle semplici operazioni richieste e per la minor complessità d'implementazione.
Il mini Web-server qui presentato ha le seguenti caratteristiche:

- Esecuzione dei metodi GET e POST
- Multi sessione
  Permette l'accesso contemporaneo di più utenti
- Generazione delle pagine in run-time
  Tutte le pagine web sono implementate in codice, ognuna di esse corrisponde ad una chiamata di funzione. Questo metodo, un poco rudimentale, semplifica notevolmente la generazione di pagine dal contenuto dinamico a discapito della comodità nella creazione di pagine web.
- Possibilità di arbitrare l'accesso con la creazione pagine ad accesso esclusivo
  Permette di identificare, con un ragionevole grado di sicurezza, un utente garantendogli l'accesso esclusivo a determinate pagine. Con 'accesso esclusivo' non si intende accesso a pagine private bensì la garanzia di essere l'unico utente, in un determinato istante, ad avere l'autorizzazione d'accesso.

La necessità di arbitrare l'accesso per determinate pagine può, a prima vista, non essere evidente. Se pensiamo ad un sistema Embedded che offre la possibilità di modificare determinati parametri in tempo reale tramite accesso Web, dobbiamo poter garantire che un solo utente alla volta possa accedere a questi parametri.
Il problema con il protocollo HTTP 1.0 è che per ogni pagina richiesta viene aperta una connessione e, dopo il suo trasferimento, viene immediatamente chiusa. Come conseguenza il Web-server non è in grado di stabilire se due richieste successive provengono dallo stesso utente.
In questa realizzazione il problema è stato risolto in due fasi:

1. L'accesso ad una serie di pagine protette avviene tramite una pagina di registrazione
2. All'indirizzo URL delle pagine protette viene aggiunto un codice (gettone) generato in run-time nella fase precedente.
   Es: l'indirizzo della pagina "http://10.0.0.1/page" viene trasformato in "http://10.0.0.1/page&af4e"

Così facendo, per un secondo utente o per un utente non registrato, risulta perlomeno difficile accedere direttamente ad una pagina protetta.
L'accesso come utente registrato sottostà ad una funzione di timeout. Dopo un determinato tempo di inattività (2min.) l'accesso scade e l'utente è obbligato a registrarsi di nuovo.

Per illustrare un Web-server occorrono evidentemente delle pagine html che però sono strettamente legate al tipo di applicazione. A scopo dimostrativo sono state realizzate alcune pagine molto semplici per illustrare il procedimento di registrazione ma senza alcuna utilità pratica. L'output generato lo si trova nell'Annesso D.

Il pacchetto HTTP è contenuto in quattro files:

html.c (.h)	: contiene le pagine html di dimostrazione
http.c (.h)	: implementa il protocollo HTTP

Come abbiamo specificato sopra, il mini Web-server realizzato è multi sessione. Inoltre certi browsers (come Netscape) utilizzano più segmenti TCP per inviare richieste con il metodo POST. Queste considerazioni ed una eventuale futura implementazione della versione HTTP 1.1, impongono di tenere traccia di ogni sessione (un po' come i socket di TCP) con dei descrittori di sessione:

```
typedef struct {
  T_SockTcb *SockPtr;        /* Associated socket pointer */
  T_MBuff *MBuffPtr;         /* Data buffer               */
  USHORT BodyLength;         /* Copy of 'Content-length'  */
  USHORT SessionFlag;        /* HTTP Session flags        */
} T_HttpSession;

static T_HttpSession HttpSession[MAX_OPEN_SESSION];
```

MAX_OPEN_SESSSION (config.h) definisce quante sessioni possono essere aperte simultaneamente.

Non avendo a disposizione un 'file system' occorre una sorta di dizionario che traduca i nomi delle pagine richieste con le rispettive funzioni:

```
struct S_Page {
  char PageName[16];                          /* Page name max. 16 char    */
  char NameLen;                               /* Page name length          */
  char *PostPtr;                              /* Pointer to POST string    */
  char PageFlag;                              /* Page flag field           */
  void (*PagePtr)(T_MBuff *, struct S_Page *); /* Function page pointer    */
};
typedef struct S_Page T_Page ;

static T_Page PageDict[MAX_WEB_PAGES];
```

MAX_WEB_PAGES (config.h) definisce il numero massimo di pagine supportate.

Il campo *PageFlag* contiene il flag *P_PROTECTED_FL* che specifica se la pagina necessita l'accesso esclusivo.
Le pagine html vanno poi inserite nel dizionario all'interno della funzione *Http_InitPageDict()*.

La funzionalità di timeout è realizzata con la variabile *Http_LoginTimeout* la quale deve essere decrementata fino a zero dall'interrupt periodico di Softask.

6.10.1 Funzioni disponibili

short Tokenized_scanf(char **String*, char **Token*, const char **Fmt*, ...)

Parametri
**String*
Stringa in cui cercare il token
**Token*
Stringa contenente il token da cercare
**Fmt*
Stringa contenente i caratteri di conversione (come *scanf(..)*)

Risultato
Ritorna OK se il token è stato convertito correttamente, altrimenti ERROR.

Commento
Usata per estrarre il valore di una variabile contenuta in una stringa.
Es: Tokenized_scanf("…Var=25…", "Var=", "%4hx", &TmpShort) => TmpShort = 25

void Http_RefWithKey(char **String*, char **Ref*, char **LinkLabel*)

Parametri
**String*
Stringa di destinazione
**Ref*
Nome della pagina
**LinkLabel*
Stringa che appare come link

Commento
Genera un link Html.
Aggiunge il codice d'accesso al nome della pagina e costruisce il link.
Es: Http_RefWithKey(Str, "PageName", "LinkName")
Str = LinkName

void Http_AddPage(T_Page *_DictPtr_, const char *_PageName_, void *_FctPtr_, char _Flag_)

Parametri
*_DictPtr_
Puntatore al descrittore nel dizionario
*_PageName_
Nome della pagina
*_FctPtr_
Puntatore alla funzione
*_Flag_
Attributi della pagina

Commento
Costruisce un descrittore d pagina T_Page e lo inserisce nel dizionario alla locazione *DictPtr*.

void Http_Task(void)

Commento
Questo è il task che implementa HTTP.
Nella fase d'inizializzazione esegue le operazioni seguenti:

- Inizializza il dizionario delle pagine
- Apre la sua coda di comunicazione (*HttpInQueue*)
- Si registra su TCP come server in ascolto (*OpenService(..)*)

Terminata l'inizializzazione resta in ascolto sulla coda di ricezione dando seguito alle richieste ricevute.

ANNESSO A - Componenti Hardware

MC68332

32-Bit Modular Microcontroller

1 Introduction

The MC68332, a highly-integrated 32-bit microcontroller, combines high-performance data manipulation capabilities with powerful peripheral subsystems. The MCU is built up from standard modules that interface through a common intermodule bus (IMB). Standardization facilitates rapid development of devices tailored for specific applications.

The MCU incorporates a 32-bit CPU (CPU32), a system integration module (SIM), a time processor unit (TPU), a queued serial module (QSM), and a 2-Kbyte static RAM module with TPU emulation capability (TPURAM).

The MCU can either synthesize an internal clock signal from an external reference or use an external clock input directly. Operation with a 32.768-kHz reference frequency is standard. The maximum system clock speed is 20.97 MHz. System hardware and software allow changes in clock rate during operation. Because MCU operation is fully static, register and memory contents are not affected by clock rate changes.

High-density complementary metal-oxide semiconductor (HCMOS) architecture makes the basic power consumption of the MCU low. Power consumption can be minimized by stopping the system clock. The CPU32 instruction set includes a low-power stop (LPSTOP) command that efficiently implements this capability.

1.1 Features

- Central Processing Unit (CPU32)
  - — 32-Bit Architecture
  - — Virtual Memory Implementation
  - — Table Lookup and Interpolate Instruction
  - — Improved Exception Handling for Controller Applications
  - — High-Level Language Support
  - — Background Debugging Mode
  - — Fully Static Operation
- System Integration Module (SIM)
  - — External Bus Support
  - — Programmable Chip-Select Outputs
  - — System Protection Logic
  - — Watchdog Timer, Clock Monitor, and Bus Monitor
  - — Two 8-Bit Dual Function Input/Output Ports
  - — One 7-Bit Dual Function Output Port
  - — Phase-Locked Loop (PLL) Clock System
- Time Processor Unit (TPU)
  - — Dedicated Microengine Operating Independently of CPU32
  - — 16 Independent, Programmable Channels and Pins
  - — Any Channel can Perform any Time Function
  - — Two Timer Count Registers with Programmable Prescalers
  - — Selectable Channel Priority Levels
- Queued Serial Module (QSM)
  - — Enhanced Serial Communication Interface
  - — Queued Serial Peripheral Interface
  - — One 8-Bit Dual Function Port
- Static RAM Module with TPU Emulation Capability (TPURAM)
  - — 2-Kbytes of Static RAM
  - — May be Used as Normal RAM or TPU Microcode Emulation RAM

MC68360

Product Brief

MC68360 QUad Integrated Communication Controller (QUICC™)

INTRODUCTION

The MC68360 QUad Integrated Communication Controller (QUICC™) is a versatile one-chip integrated microprocessor and peripheral combination that can be used in a variety of controller applications. It particularly excels in communications activities. The QUICC (pronounced “quick”) can be described as a next-generation MC68302 with higher performance in all areas of device operation, increased flexibility, major extensions in capability, and higher integration. The term "quad" comes from the fact that there are four serial communications controllers (SCCs) on the device; however, there are actually seven serial channels: four SCCs, two serial management controllers (SMCs), and one serial peripheral interface (SPI).

QUICC Key Features

The following list summarizes the key MC68360 QUICC features:

- CPU32+ Processor (4.5 MIPS at 25 MHz)
  - — 32-Bit Version of the CPU32 Core (Fully Compatible with the CPU32)
  - — Background Debug Mode
  - — Byte-Misaligned Addressing
- Up to 32-Bit Data Bus (Dynamic Bus Sizing for 8 and 16 Bits)
- Up to 32 Address Lines (At Least 28 Always Available)
- Complete Static Design (0–25-MHz Operation)
- Slave Mode To Disable CPU32+ (Allows Use with External Processors)
  - — Multiple QUICCs Can Share One System Bus (One Master)
  - — MC68040 Companion Mode Allows QUICC To Be an MC68040 Companion Chip and Intelligent Peripheral (22 MIPS at 25 MHz)
  - — Also Supports External MC68030-Type Bus Masters
  - — All QUICC Features Usable in Slave Mode
- Memory Controller (Eight Banks)
  - — Contains Complete Dynamic Random-Access Memory (DRAM) Controller
  - — Each Bank Can Be a Chip Select or Support a DRAM Bank
  - — Glueless Interface to DRAM Single In-Line Memory Modules (SIMMs), Static Random-Access Memory (SRAM), Electrically Programmable Read-Only Memory (EPROM), Flash EPROM, etc.
  - — Four CAS lines, Four WE lines, One OE line
  - — Boot Chip Select Available at Reset (Options for 8-, 16-, or 32-Bit Memory)
- Four General-Purpose Timers
  - — Superset of MC68302 Timers
  - — Four 16-Bit Timers or Two 32-Bit Timers
  - — Gate Mode Can Enable/Disable Counting
- Two Independent DMAs (IDMAs)
  - — Single Address Mode for Fastest Transfers
  - — Buffer Chaining and Auto Buffer Modes
  - — Automatically Performs Efficient Packing
  - — 32-Bit Internal and External Transfers
- System Integration Module (SIM60)
  - — Bus Monitor
  - — Double Bus Fault Monitor
  - — Spurious Interrupt Monitor
  - — Software Watchdog

— Periodic Interrupt Timer
— Low Power Stop Mode
— Clock Synthesizer
— Breakpoint Logic Provides On-Chip Hardware Breakpoints
— External Masters May Use On-Chip Features Such As Chip Selects
— On-Chip Bus Arbitration with No Overhead for Internal Masters

• Interrupts
— Seven External IRQ Lines
— 12 Port Pins with Interrupt Capability
— 16 Internal Interrupt Sources
— Programmable Priority Between SCCs
— Programmable Highest Priority Request

• Communications Processor Module (CPM)
— RISC Controller
— 224 Buffer Descriptors
— Supports Continuous Mode Transmission and Reception on All Serial Channels
— 2.5 Kbytes of Dual-Port RAM
— 14 Serial DMA (SDMA) Channels
— Three Parallel I/O Registers with Open-Drain Capability
— Each Serial Channel Can Have Its Own Pins (NMSI Mode)

• Four Baud Rate Generators
— Independent (Can Be Connected to Any SCC or SMC)
— Allows Changes During Operation
— Autobaud Support Option

• Four SCCs
— Ethernet/IEEE 802.3 Optional on SCC1 (Full 10-Mbps Support) (Available only on the MC68EN360)
— HDLC/SDLC™ (All Four Channels Supported at 2 Mbps)
— HDLC Bus (Implements an HDLC-Based Local Area Network (LAN))
— AppleTalk ®
— Signaling System #7
— Universal Asynchronous Receiver Transmitter (UART)
— Synchronous UART
— Binary Synchronous Communication (BISYNC)
— Totally Transparent (Bit Streams)
— Totally Transparent (Frame Based with Optional Cyclic Redundancy Check (CRC))
— Profibus (RAM Microcode Option)
— Asynchronous HDLC (RAM Microcode Option) to Support PPP (Point to Point Protocol)
— DDCMP™ (RAM Microcode Option)
— V.14 (RAM Microcode Option)
— X.21 (RAM Microcode Option)

• Two SMCs
— UART
— Transparent
— Can Be Connected to the Time-Division Multiplexed (TDM) Channels

• One SPI
— Superset of the MC68302 SCP
— Supports Master and Slave Modes
— Supports Multimaster Operation on the Same Bus

• Time-Slot Assigner

• Supports Two TDM Channels
— Each TDM Channel Can Be T1, CEPT, PCM Highway, ISDN Basic Rate, ISDN Primary Rate, User Defined
— 1- or 8-Bit Resolution
— Allows Independent Transmit and Receive Routing, Frame Syncs, Clocking
— Allows Dynamic Changes
— Can Be internally Connected to Six Serial Channels (Four SCCs and Two SMCs)

• Parallel Interface Port
— Centronics ™ Interface Support

— Supports Fast Connection Between QUICCs

82527

SERIAL COMMUNICATIONS CONTROLLER CONTROLLER AREA NETWORK PROTOCOL

- **Supports CAN Specification 2.0**
  - -Standard Data and Remote Frames
  - -Extended Data and Remote Frames
- **Programmable Global Mask**
  - -Standard Message ldentifier
  - -Extended Message ldentifier
- **15 Message Objects of 8-Byte Data Length**
  - -14 Tx/Rx Buffers
  - -1 Rx Buffer with Programmable Mask
- **Flexible CPU Interface**
  - -8-Bit Multiplexed
  - -16-Bit Multiplexed
  - -8-Bit Non-Multiplexed (Synchronous/Asynchronous)
  - -Serial Interface
- **Programmable Bit Rate**
- **Programmable Clock Output**
- **Flexible Interrupt Structure**
- **Flexible Status Interface**
- **Configurable Output Driver**
- **Configurable Input Comparator**
- **Two 8-Bit Bidirectional I/O Ports**
- **44-Lead PLCC Package**
- **44-Lead QFP Package**
- **Pinout Compatibility with the 82526**

The 82527 serial communications controller is a highly integrated device that performs serial communication according to the CAN protocol. It performs all serial communication functions such as transmission and reception of messages, message filtering, transmit search, and interrupt search with minimal interaction from the host microcontroller, or CPU.

The 82527 is Intel's first device to support the standard and extended message frames in CAN Specification 2.0 Part B. It has the capability to transmit, receive, and perform message filtering on extended message frames. Due to the backwardly compatible nature of CAN Specification 2.0, the 82527 also fully supports the standard message frames in CAN Specification 2.0 Part A.

The 82527 features a powerful CPU interface that offers flexibility to directly interface to many different CPUs. It can be configured to interface with CPUs using an 8-bit multiplexed, 16-bit multiplexed, or 8-bit non-multiplexed address/data bus for Intel and non-Intel architectures. A flexible serial interface (SPI) is also available when a parallel CPU interface is not required.

The 82527 provides storage for 15 message objects of 8-byte data length. Each message object can be configured as either transmit or receive except for the last message object. The last message object is a receive-only buffer with a special mask design to allow select groups of different message identifiers to be received.

The 82527 also implements a global masking feature for message filtering. This feature allows the user to globally mask any identifier bits of the incoming message. The programmable global mask can be used for both standard and extended messages.

The 82527 PLCC offers hardware, or pinout, compatibility with the 82526. It is pin-to-pin compatible with the 82526 except for pins 9, 30, and 44. These pins are used as chip selects on the 82526 and are used as CPU interface mode selection pins on the 82527.

The 82527 is fabricated using Intel's reliable CHMOS III 5V technology and is available in either 44-lead PLCC or 44-lead QFP for the automotive temperature range (b40 §C toa125 §C).

ANNESSO B - Bibliografia

- RFC 791,792,793,1945
- “TCP/IP Illustrated, Vol. 1” (Stevens) ISBN 0-201-63346-9
- “TCP/IP Illustrated, Vol. 2” (Wright, Stevens) ISBN 0-201-63354-X
- “TCP/IP LEAN Web Servers for Embeded Systems” (Bentham) ISBN 1-929629-11-7
- “Mastering Local Area Networks” (Anderson, Minasi) ISBN 0-7821-2258-2
- “Reti di commando e controllo industriale” (Colla, Mondada) ISBN 88-900298-1-1
- “CAN Grundlagen und praxis” (Lawrenz) ISBN 3-7785-2263-9
- “MC68360 User’s manual” Motorola
- “MC68332 User’s manual” Motorola
- “82527 Serial Communications Controller Architectural Overview” Intel

ANNESSO C - Sorgenti

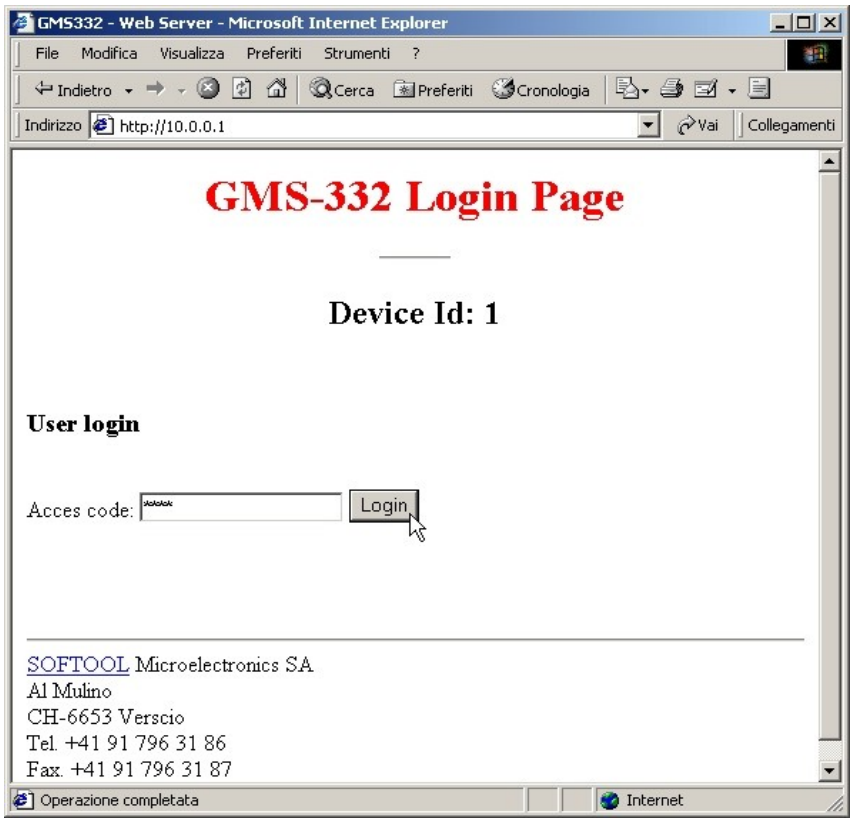
Si ricorda che i listati allegati in questo annesso sono proprietà intellettuale della *Softool Microelectronics SA*. Sono stati inclusi nella documentazione a titolo informativo. Tuttavia la loro duplicazione e/o riproduzione con qualsiasi mezzo, con qualsiasi forma e per qualsiasi scopo è vietata previo il consenso scritto di *Softool Microelectronics SA*.

Elenco dei moduli allegati:

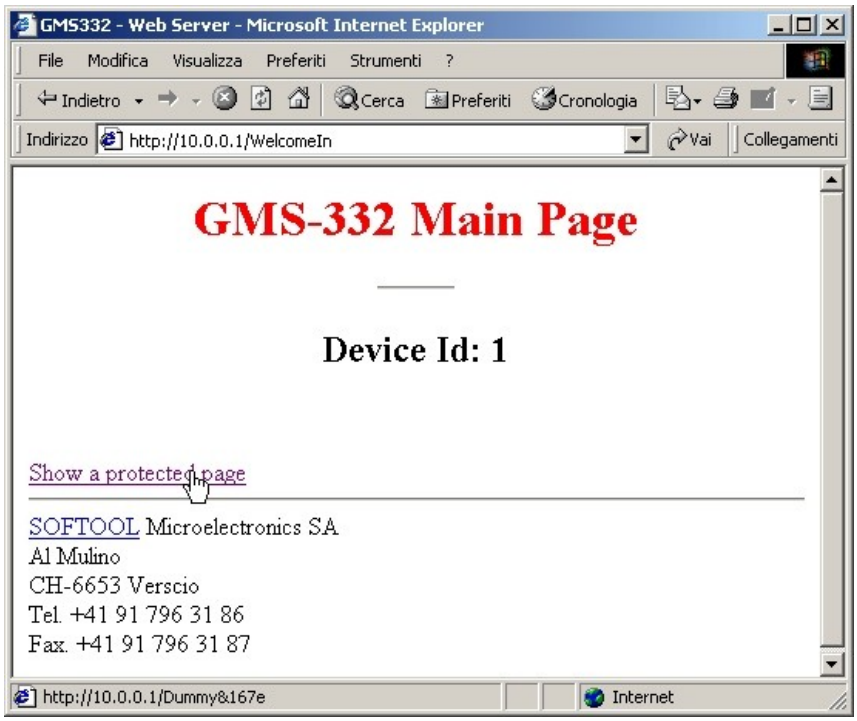
- stask.h : Definizione di Softaks
- Config.h : Configurazione dell'intero sistema
- NetUtil.h : Definizione Network utilities
- NetUtil.c : Network utilities
- Ethernet.h : Definizione per driver Ethernet
- Eth360.c : Driver internet per MC68EN360
- Candrv.h : Definizione per driver CAN
- CanApp.h : Definizione per CanApp
- CanApp.c : Application layer di CAN (implementa frammentazione di IP)
- Arp.h : Definizione per ARP
- Arp.c : Protocollo ARP
- Cirp. h : Definizione per CIRP
- Cirp.c : Protocollo CIRP
- Ip. h : Definizione protocollo IP
- Ip.c : Implementazione protocollo IP
- Icmp. h : Definizione protocollo ICMP
- Icmp.c : Implementazione protocollo ICMP
- Tcp. h : Definizione protocollo TCP
- Tcp.c : Implementazione protocollo TCP
- http. h : Definizione protocollo HTTP
- http.c : Implementazione protocollo HTTP
- html.h : Definizione pagine html
- html.c : Implementazione implementazione pagine html
- Ripc.c : Estratto main() router IPC

ANNESSO D – Pagine Web

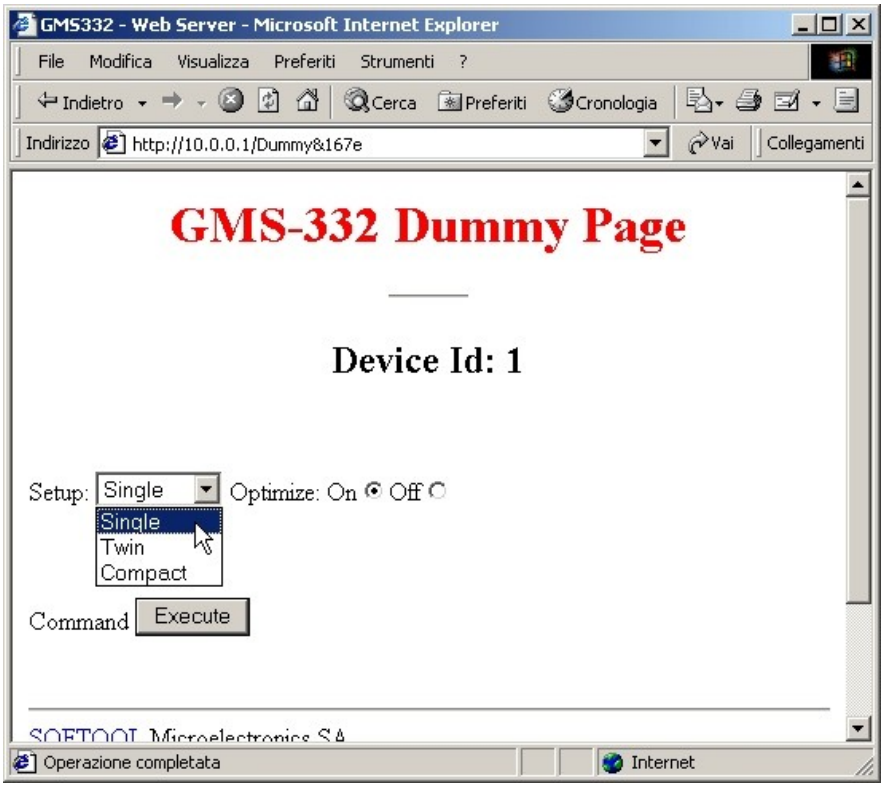
1. Pagina di registrazione (*Http_P_Index()*).



2. Pagina principale per le pagine protette(*Http_P_WelcomIn()*). Notare il link alla pagina protetta con la chiave '167e'



3. Pagina protetta (*Http_P_Dummy()*). Notare la chiave inserita nel URL



4. Accesso negato (*Http_P_WelcomIn()*). Generata alla registrazione con una password errata.