Proteus 9.1: interfaccia del programma con il mondo esterno

L’ interazione del programma con il mondo esterno è garantita da una serie di interfacce bidirezionali di comunicazione seriale, ciascuna delle quali ha un proprio protocollo di gestione. Le interfacce disponibili per i controller Syel con sistema operativo Linux attualmente in produzione sono:

interfaccia	**CPU Broadcom BCM283X**	**CPU Allwinner A20**
• Porte seriali RS-232	COM1/COM2	COM1/COM2
• Porte seriali RS-485	COM3	COM3/COM4
• I2C	I2C-1	I2C-1/I2C-2
• SPI	spidev0.1	spidev0.0/spidev2.0
• Ethernet	ETH0/ETH0.1/WIFI0	ETH0/ETH0.1
• CAN	can0	can0

Porte seriali RS-232 e RS-485

Vengono viste dal sistema operativo e dall' utente nello stesso modo. L' unica differenza consiste nell' hardware di adattamento dei livelli dei segnali verso il mondo esterno, e nel fatto che mentre le porte RS-232 sono full-duplex, ovvero possono ricevere dati anche mentre li stanno trasmettendo, le porte RS-485 sono half-duplex, ovvero l' invio di dati mentre se ne stanno ricevendo crea un conflitto con perdita dei dati.

Mentre per quanto riguarda la trasmissione i dati vengono sempre inviati immediatamente, i dati in ricezione vengono messi in un buffer, ed esistono due modi per scodarli: il modo blocking e il modo no-blocking.
Nel modo no-blocking un comando di ricezione di un byte ritorna con il valore del primo dato scodato dal buffer di ricezione, oppure con -1 se non ve ne sono, mentre un comando di ricezione di un blocco di dati ritorna con il numero di bytes scodati oppure con 0 se il buffer di ricezione è vuoto.
Nel modo blocking un comando di ricezione sospende il task che lo ha inviato fino a quando il buffer di ricezione è vuoto. Questo metodo consente di alleggerire il carico della CPU evitando un polling continuo in attesa dell' arrivo dei dati, ma evidentemente è utilizzabile solo in un task che ha lo scopo di trattare i dati in arrivo, e che non blocchi il programma principale quando non ve ne sono.

Le funzioni per gestire le comunicazioni seriali RS-232 e RS-485 per LINUX sono:

funzioni di gestione del protocollo:

- **int com_open(struct COM *porta,S32 baud);**
- **int com_close(struct COM *porta);**
- **int com_enable(struct COM *com);**
- **int com_disable(struct COM *com);**
- **int com_block(struct COM *com,int block);**
- **int com_speed(struct COM *porta,S32 baud);**
- **int protocol_mode(struct COM *porta,char mode);**

funzioni di ricezione e trasmissione di dati (protocol mode 0):

- **int com_rx(struct COM *porta);**
- **int com_rxs(struct COM *com,char *buffer,int max);**
- **int com_tx(struct COM *porta,U8 c);**
- **int com_txs(struct COM *porta,U8 *c);**
- **int com_txsl(struct COM *porta,U8 *c,int len);**
- **int com_protocol(struct COM *com,int protocol);**

funzioni di controllo:

- **int com_rx_empty(struct COM *porta);**
- **int com_rx_size(struct COM *porta);**
- **int com_tx_empty(struct COM *porta);**
- **int com_tx_full(struct COM *porta);**
- **int com_tx_size(struct COM *porta);**

funzioni di ricezione e trasmissione di blocchi di dati (protocol mode 1):

- **int bload(struct COM *porta,void *buffer,int len);**
- **int bsave(struct COM *porta,void *buffer,int len);**
- **int com_rxlen(struct COM *porta);**
- **int com_rxcode(struct COM *porta);**
- **int onrx(struct COM *porta,void(*subroutine)());**

int com_open(struct COM *porta,S32 baud);

Apre una porta (COM1,COM2,COM3 o COM4) specificando la baud rate.
Le baud rates gestibili da Linux sono le seguenti:

50,75,110,134,150,200,300,600,1200,1800,2400,4800,9600,19200,38400,57600,115200,230400,460800,
500000,576000,921600,1000000,1152000,1500000,2000000,2500000,3000000,3500000,4000000

Dichiarando una baud rate diversa da questi valori, alla porta sarà assegnata una baud rate di 115200 baud.
Se la porta non esiste oppure se è già aperta la funzione ritorna con il valore -1 altrimenti ritorna con il valore 0.

int com_close(struct COM *porta);

Chiude una com precedentemente aperta. In genere questa funzione non viene mai usata.

int com_enable(struct COM *com);

Abilita la com specificata. Quando una porta viene aperta, essa viene automaticamente abilitata.
Questa funzione serve per riabilitare una porta precedentemente disabilitata con la funzione com_disable.
Questa funzione inoltre deve essere obbligatoriamente usata dopo l' esecuzione delle funzioni:

- com_block(..)
- protocol_mode(..)
- com_protocol(..)
- on_rx(..)

in quanto serve per attivare le funzioni suddette.

int com_disable(struct COM *com);

Disabilita l' attività di una com. Sebbene non necessaria, è bene usarla prima di eseguire le funzioni:

- com_block(..)
- protocol_mode(..)
- com_protocol(..)
- on_rx(..)

Dopo l' esecuzione delle suddette funzioni il comando com_enable le attiverà e riabiliterà la com.

int com_block(struct COM *com,int block);

Se il parametro block vale 1 viene attivata la modalità di ricezione block_mode on.
Se il parametro block vale 0 viene attivata la modalità di ricezione block_mode off.

int com_speed(struct COM *porta,S32 baud);

Cambia la baud rate di una porta precedentemente aperta con com_open.
Le baud rates gestibili da Linux sono le seguenti:

50,75,110,134,150,200,300,600,1200,1800,2400,4800,9600,19200,38400,57600,115200,230400,460800,
500000,576000,921600,1000000,1152000,1500000,2000000,2500000,3000000,3500000,4000000

Dichiarando una baud rate diversa da questi valori, alla porta sarà assegnata una baud rate di 115200 baud.

int protocol_mode(struct COM *porta,char mode);

Nella versione corrente di Proteus esistono solo due modi di protocollo:
Mode 0 (default all' apertura della porta) con gestione della trasmissione e ricezione a livello di bytes o di stringhe di bytes.
Mode 1 con gestione della trasmissione e ricezione a blocchi di dati formattati (bload, bsave).

int com_rx(struct COM *porta);

Scoda un byte dal buffer di ricezione e ne ritorna il valore come un intero nel range 0-255
Se il buffer di ricezione è vuoto ritorna con il valore -1.

int com_rxs(struct COM *com,char *buffer,int max);

Scoda al massimo max bytes dal buffer di ricezione e li mette in buffer.
Ritorna con il numero di bytes scodati (con 0 se il buffer di ricezione è vuoto) .
Da notare che questa funzione ritorna con i bytes finora ricevuti, per cui se la ricezione dei dati è ancora in corso non è detto che i bytes ricevuti contengono tutto il messaggio in corso di ricezione.

int com_tx(struct COM *porta,U8 c);

Trasmette un byte (il carattere c) sulla porta specificata.

int com_txs(struct COM *porta,U8 *c);

Trasmette la stringa ascii contenuta nel vettore C, fino al carattere finale 0 escluso.

int com_txsl(struct COM *porta,U8 *c,int len);

Trasmette len dati della stringa contenuta nel vettore c.
A differenza della funzione com_txs() la stringa può contenere caratteri nulli.

int com_protocol(struct COM *com,int protocol);

Specificando come argomento l' indirizzo di una funzione, per ogni byte ricevuto verrà chiamata tale funzione, che ha come argomenti la com che ha ricevuto il dato ed il valore del dato.
Le funzioni com_rx e com_rxs in questo caso sono disabilitate.
Se l' argomento protocol della funzione com_protocol vale 0 viene riabilitato il modo di ricezione al byte, riabilitando le funzioni com_rx() e com_rxs().

Esempio:
Questo esempio mostra l' utilizzo di com_protocol per gestire la ricezione di blocchi di dati racchiusi tra parentesi quadre.

```
int nrx3=0;    //numero di blocchi ricevuti
int sizerx3=0; //dimensione in bytes dell' ultimo blocco ricevuto
char rx3[100]; //dati dell' ultimo blocco ricevuto
// ----------------------------------------+
//  protocollo ricezione COM3
// ----------------------------------------+
void myprotocol3(COM *com, char c)
{
   static int status=0;
   static int n3=0;
   switch(status)
   {
       case (0):
       if (c=='[') {status=1;n3=0;}
       break;
       case (1):
       if (c==']') {status=0;nrx3++;sizerx3=n3;}
       else rx3[n3++]=c;
       break;
   }
}

int main(…)
{
….
   com_open(COM3,115200);
   com_protocol(COM3,myprotocol3);
   com_block(COM3,1);        //** non indispensabile ma snellisce il carico CPU
   com_enable(COM3);         //con protocollo al byte
…
}
```

int com_rx_empty(struct COM *porta);

ritorna con il valore 1 se il buffer di ricezione è vuoto, altrimenti ritorna con il valore 0

int com_rx_size(struct COM *porta);

ritorna con il numero di bytes attualmente presenti nel buffer di ricezione e non ancora scodati.

int com_tx_empty(struct COM *porta);

ritorna con il valore 1 se tutti i bytes sono stati inviati, altrimenti ritorna con il valore 1 se l' invio è ancora in corso.

int com_tx_full(struct COM *porta);

ritorna con il valore 1 se il buffer di trasmissione è pieno

int com_tx_size(struct COM *porta);

ritorna con il numero di bytes attualmente presenti nel buffer di trasmissione e non ancora inviati.

int bload(struct COM *porta,void *buffer,int len);

Se per la porta è stato dichiarato protocol_mode(porta,1):
Questa funzione mette in buffer l' eventuale blocco di dati ricevuti e destinati a lei, e riabilita la ricezione di altri blocchi di dati. Ritorna con la lunghezza del blocco di dati ricevuti, oppure con 0 se non è arrivato niente. Questo modo è particolarmente utile se usato su porte RS-485 e permette di inviare e ricevere blocchi di dati di lunghezza non superiore a 255 bytes da numerose periferiche collegate sullo stesso bus. Un blocco è considerato valido se il suo checksum è giusto e l' indirizzo del destinatario corrisponde ai bit 4-7 di com->centr dichiarato per la com. Il tipo di messaggio è contenuto nei 4 bit meno significativi di tale indirizzo.
Se è stata dichiarata la callback onrx(com,callback) la funzione callback viene eseguita ogni volta che un blocco di dati viene ricevuto.

Esempio:
COM4 viene dichiarata in protocol_mode 1 e le viene assegnato il numero di centrale 0x20.
Viene dichiarata la callback mycom4sub che intercetterà tutti i messaggi in arrivo su COM4 con indirizzo 0x2X e ne metterà il contenuto in rx4

```
int nrx4=0;    //numero di blocchi ricevuti
int sizerx4=0; //dimensione in bytes dell' ultimo blocco ricevuto
char rx4[100]; //dati dell' ultimo blocco ricevuto

// ----------------------------------------+
//  protocollo ricezione COM4
// ----------------------------------------+
void mycom4sub(COM *com)
{
   int n4=bload(com,rx4,100);
   if (n4) {nrx4++;sizerx4=n4;}
}

int main(…)
{
   …
   com_open(COM4,1000000);
   protocol_mode(COM4,1);     //bload-bsave
   COM4->centr=0x20;          //N. CENTRALE
   onrx(COM4,mycom4sub);       //routine da eseguire dopo la ricezione di un pacchetto
   com_block(COM4,1);         //** non indispensabile ma snellisce il carico CPU
   com_enable(COM4);          //ABILITA LA COM
   …
}
```

int bsave(struct COM *porta,void *buffer,int len);

Se per la porta è stato dichiarato protocol_mode(porta,1):
Questa funzione invia un messaggio di lunghezza len contenente i dati presenti in buffer sulla com specificata alla periferica comx->perif (bit 0-3 = tipo di messaggio, bit 4-7=destinatario del messaggio)

Esempio:

```
char tx4[100]; //dati da trasmettere

int main(…)
{
…
   com_open(COM4,1000000);
   protocol_mode(COM4,1);     //bload-bsave
   com_block(COM4,1);         //** non indispensabile ma snellisce il carico CPU
   com_enable(COM4);          //ABILITA LA COM
   …
       sprintf(tx4,"Ciao periferica 0x1x, ti invio un messaggio di tipo 4”);
       COM4->perif=0x14;      //N. PERIFERICA e TIPO di MESSAGGIO
       bsave(COM4,tx4,strlen(tx4));
   …
}
```

int com_rxlen(struct COM *porta);

Permette di conoscere la lunghezza del messaggio ricevuto prima di leggerlo con bload().
Può servire per allocare la dimensione del vettore in cui leggere il messaggio.

int com_rxcode(struct COM *porta);

Permette di conoscere il tipo di messaggio ricevuto prima di leggerlo con bload().
Può servire per fare un bload su tipi di strutture diverse a seconda del tipo di messaggio.

int onrx(struct COM *porta,void(*subroutine)());

Vedi bload().

I2C

L' interfaccia seriale I2C permette l' accesso ai devices I2C eventualmente presenti sulla scheda. Come per l' SPI non è un bus adatto a lunghe distanze e normalmente non è disponibile come porta esterna.
La bit rate è fissata a 100 KBaud. Una singola interfaccia I2C permette la gestione di più periferiche che abbiano indirizzi differenti (fino a 256 devices).

Funzioni relative a I2C:

- **int i2c_start(struct I2cDevice* dev,char *filename,int addr,int reg_width,int val_width);**
- **void i2c_stop(struct I2cDevice* dev);**
- **int i2c_read(struct I2cDevice* dev, unsigned char *buf, size_t buf_len);**
- **int i2c_write(struct I2cDevice* dev, unsigned char *buf, size_t buf_len);**
- **int i2c_rr(struct I2cDevice* dev,unsigned int firstreg, void *buffer, int nregs);**
- **int i2c_wr(struct I2cDevice* dev,unsigned int firstreg, void *buffer, int nregs);**

int i2c_start(struct I2cDevice* dev,char *filename,int addr,int reg_width,int val_width);

Inizializza ed attiva il collegamento con un device collegato ad una porta I2C.
Un device I2C è organizzato come una serie di parole (o registri) di memoria che possono essere lette o scritte.
Se il numero di tali parole è superiore a 256 l' indirizzo della parola è di 16 bit, altrimenti è di 8 bit
Ogni parola può essere un byte o un word. Per un device tutte le parole hanno comunque la stessa lunghezza (tutte bytes o tutte words).
Se una parola è di tipo word è organizzata in modalità big-endian (prima gli 8 bit più significativi).
Le funzioni qui descritte comunque traslano automaticamente i valori nel formato little endian che è il formato usato nei programmi scritti per Linux)
Prima di scrivere le funzioni per un device è bene verificare se il device e la porta esistono.
A tale scopo è possibile usare il comando Linux i2cdetect. Ad esempio se vogliamo verificare che esista la porta /dev/i2c-1 e che su tale porta sia presente il device i2c con indirizzo 0x33, scriviamo:

```
pi> i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- 33 -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Il parametro **dev** punta ad una struttura di tipo I2cDevice:

```
struct I2cDevice
{
        char* filename; //< Path of the I2C bus, eg: /dev/i2c-0
        uint16_t addr;  //< Address of the I2C slave, eg: 0x48
        int fd;         //< File descriptor for the I2C bus
        int reg_width;
        int val_width;
```

```
};
```

il parametro **filename** è il nome della porta I2C, che deve essere presente nella directory /dev della macchina, ad esempio: /dev/i2c-1

il parametro addr è l' indirizzo fisico del device, che è compreso tra 0 e 127 (0x00-0x7f)

Il parametro **reg_width** indica la lunghezza in bit dell' indirizzo delle parole del device e può essere:

- I2C_8BIT quando il device ha meno di 256 parole (o registri)
- I2C_16BIT quando il device ha più di 256 parole (o registri)

Il parametro **val_width** indica la lunghezza in bit di ogni singola parola (o registro) del device e può essere:

- I2C_8BIT quando il device ha meno di 256 parole (o registri)
- I2C_16BIT quando il device ha più di 256 parole (o registri)

Se questi due ultimi parametri sono omessi, si considerano di 8 bit.

Esempio:

```
struct I2cDevice dev;
i2c_start(&dev,"/dev/i2c-1",0x33,I2C_16BIT,I2C_16BIT);
```

void i2c_stop(struct I2cDevice* dev);

Disattiva il device attivato con i2c_start.
Per riattivarlo nuovamente è sufficiente un comando **i2c_start(dev);**

int i2c_rr(struct I2cDevice* dev,unsigned int firstreg, void *buffer, int nregs);

Legge in buffer nregs parole (o registri) a partire dal registro numero firstreg dal device dev.
Ritorna con il numero di registri letti.

int i2c_wr(struct I2cDevice* dev,unsigned int firstreg, void *buffer, int nregs);

Scrive nregs parole (o registri) a partire dal registro numero firstreg dal device dev il contenuto di buffer.
Ritorna con il numero di registri scritti.

int i2c_read(struct I2cDevice* dev, unsigned char *buf, size_t buf_len);

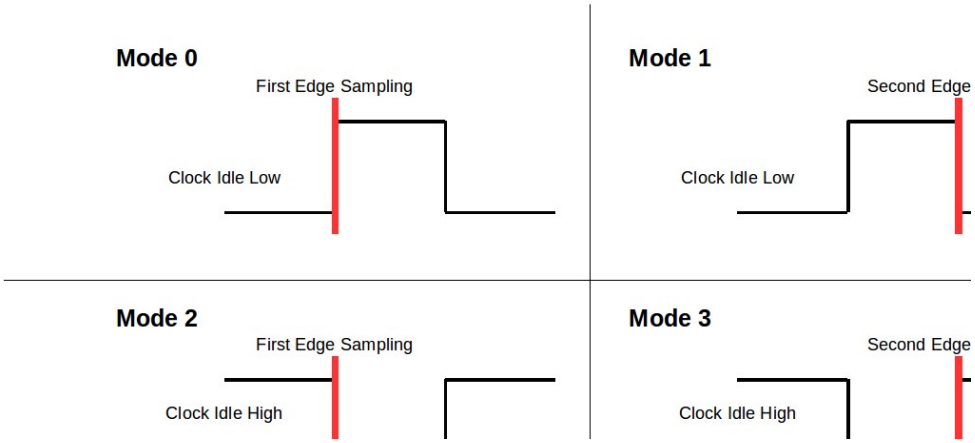
Non usato, solo per compatibilità e per scopi particolari.

int i2c_write(struct I2cDevice* dev, unsigned char *buf, size_t buf_len);

Non usato, solo per compatibilità e per scopi particolari.

SPI

Una porta SPI, analogamente ad una porta I2C, ha lo scopo di scambiare dati con devices posti sulla scheda del controller. A differenza dell' I2C:
Una porta SPI è capace di colloquiare con un solo device (esistono SPI con più di un segnale select capaci di colloquiare con più devices, ma non è il caso delle CPU che ci interessano).
Il colloquio è full-duplex e l' invio dei dati e la ricezione della risposta avviene in un' unica fase.
La baud rate può essere qualsiasi, da un minimo di 100 KBps. Ad un massimo i 100 MBps.
Esistono 4 possibili modalità di colloquio, che si differenziano per la polarità del clock e per il fronte del clock su cui i dati si considerano validi.



I due comandi usati per gestire l' SPI sono:

- **int SPISetupMode (int channel, int speed, int mode);**
- **int SPIRW(int channel, unsigned char *txdata,unsigned char *rxdata, int len);**

int SPISetupMode (int channel, int speed, int mode);

Channel è il numero della porta SPI.
Speed è la bit-rate espressa in bit per secondo
Mode è il modo (se omesso per default viene usato il modo 0 che è il più comune)

int SPIRW(int channel, unsigned char *txdata,unsigned char *rxdata, int len);

Channel è il noumero della porta SPI.
Txdata è il vettore dei dati da inviare
Rxdata è il vettore in cui saranno messi i dati ricevuti.
Len è il numero totale di bytes del messaggio (dati inviati + risposta).
Rxdata e Txdata possono anche essere lo stesso vettore.
Il buffer della risposta contiene all' inizio i dati inviati.
Non è specificato il numero dei bytes inviati, quindi il buffer dei dati inviati deve essere lungo come l' intero messaggio, e dopo la parte contenente i bytes da inviare (che dipende dal tipo di messaggio e dal device) può contenere dati qualsiasi (normalmente viene riempito con caratteri nulli).

Questa routine gestisce solo SPI 3 wire (CON MISO E MOSI SEPARATI).

Esempio:

```
char txbuffer[10]={0x01,0x02,0,0,0,0,0,0};
char rxbuffer[10]={0,0,0,0,0,0,0,0};
SPISetupMode(1,1000000,0);
SPIRW (1, txbuffer,rxbuffer, 8);
```

ETHERNET

Vi sono numerosissime funzioni di Linux per gestire l' ethernet.
In questo capitolo prenderemo in considerazione solo le funzioni semplificate ad alto livello che gestiscono i messaggi ethernet come normali files di dati. Queste funzioni sono concepite per consentire una gestione estremamente semplificata di colloqui TCP ed UDP.

Le connessioni ethernet si possono dividere in 4 classi:

- **UDP client**
- **UDP server**
- **TCP client**
- **TCP server**

Una connessione UDP è paragonabile ad una connessione seriale RS-232 in cui vengono scambiati pacchetti di dati senza una gerarchia master-slave. La distinzione client-server indica solo chi prende l' iniziativa di collegarsi (il client) e chi accetta il collegamento dietro richiesta (il server). Non è garantita la ricezione dei messaggi in nessuno dei due sensi, per cui questo tipo di connessione è indicata nel caso in cui una eventuale perdita di dati non sia importante.

Una connessione TCP è paragonabile all' accesso ad una memoria di massa. Il client apre una connessione e quando questa viene accettata può iniziare ad inviare e ricevere dati dal server senza una gerarchia master-slave. A differenza di una connessione UDP la consegna dei dati nei due sensi è garantita da un protocollo di acknowledgement che reinvia i dati in caso di mancata conferma di ricezione. Ciò non implica comunque una gerarchia master-slave in quanto ognuna delle due parti collegate può inviare dati spontaneamente. Anche in questo caso la distinzione client-server indica solo chi prende l' iniziativa di collegarsi (il client) e chi accetta il collegamento dietro richiesta (il server).

Una volta aperta una connessione il client ed il server possono fare sul file associato, con la stessa sintassi usata per gestire i files di dati, una serie operazioni**: eof(), fseek(), fread(), fwrite(), fclose()** per chiudere la connessione, e ,solo per TCP, **fgetc()** . Questo è sufficiente per una completa gestione dello scambio dei dati.

UDP client

Apertura della connessione:

FILE *fopen(char *ip,unsigned short port);

dove ip è una stringa ascii del tipo: **UDP:ip1.ip2.ip3.ip4**

chiusura della connessione:

fclose(FILE *file);

invio di un telegram:

int result=fwrite(char *txmsg,num,len,FILE *file);

la sintassi è la stessa di quella per la scrittura su memoria di massa, vengono inviati num blocchi di lunghezza len e il risultato è il numero di bytes realmente inviati.

ricezione di un telegram:

int result=fread(char *rxmsg,len,num,FILE *file);

la sintassi è la stessa di quella per la lettura da memoria di massa, vengono chiesti num blocchi di lunghezza len e il risultato è il numero di bytes realmente ricevuti.

Esempio:

tento un collegamento UDP con il server all' indirizzo ip 192.168.1.33 sulla porta 1555.
Se il collegamento è accettato invio 10 bytes , attendo un po', leggo l' eventuale risposta e chiudo il collegamento.

```
char txmsg[10]={1,2,3,4,5,6,7,8,9,10};
char rxmsg[10];
int inviati,ricevuti;

FILE *f = fopen("UDP:192.168.1.133" , 1555);
if (f)
{
   inviati=fwrite(txmsg, 1, 10, f);
   wait(100);
   ricevuti=fread(rxmsg, 1, 10, f);
   fclose(f);
}
```

UDP server

Dichiarazione della callback di ricezione

FILE *f= fopen(char *port,int sub);

dove port è una stringa ascii del tipo **UDP-SERVER:n** dove n è il numero della porta del server e sub è il nome della funzione di callback che verrà chiamata quando si riceve un messaggio sulla porta n.

La callback di ricezione:

int sub(char *rxbuf,char *txbuf,int flag,int port,int len)

dove rxbuf è il puntatore al buffer che contiene il messaggio ricevuto
txbuf è il puntatore al buffer dove scrivere la risposta
flag è il tipo di connessione (non serve in quanto evidentemente è di tipo UDP)
port è il numero della porta (serve solo se la stessa subroutine è usata con porte differenti)
len è la lunghezza in bytes del messaggio ricevuto
la subroutine esamina il messaggio ed eventualmente mette in txbuf la risposta
la subroutine ritorna con la lunghezza in bytes della risposta (zero se non c' è risposta)
In questo modo è gestito un protocollo master-slave in cui il client invia richieste e il server dà le risposte.
Tuttavia il server può in ogni momento inviare spontaneamente un telegram sulla porta aperta con la dichiarazione della callback, come mostrato nell' esempio sottostante.

Esempio:

viene dichiarato il server per la porta 0x2765, il quale quando riceve un telegram lo rimanda al client con un messaggio che ne indica la lunghezza.
Dopo 10 secondi di inattività da parte del client manda spontaneamente un messaggio segnalando che il server è ancora attivo.

```
FILE *f;
void timeout_2765(void)
{
     If (f) fwrite("ci sono ancora!\n",1,16,f);
}
int my_sub_2765(char *rxbuf,char *txbuf,int flag,int port,int len)
{
    sprintf(txbuf,"received %d bytes %s",strlen(rxbuf),rxbuf);
    exectimer (timeout_2765,ONCE,10000);
    return strlen(txbuf);
}
.....
f = fopen("UDP-SERVER:0x2765",(int)my_sub_2765);
```

Il server può gestire contemporaneamente più connessioni su porte diverse e più connessioni sulla stessa porta da client diversi, a condizione che la funzione di callback sia rientrante, ovvero non usi variabili globali né variabili statiche.

TCP client

Apertura della connessione:

FILE *fopen(char *ip,unsigned short port);

dove ip è una stringa ascii del tipo: **TCP:ip1.ip2.ip3.ip4**

chiusura della connessione:

fclose(FILE *file);

invio di dati:

int result=fwrite(char *txmsg,num,len,FILE *file);

la sintassi è la stessa di quella per la scrittura su memoria di massa, vengono inviati num blocchi di lunghezza len e il risultato è il numero di bytes realmente inviati.

ricezione di dati:

int result=fread(char *rxmsg,len,num,FILE *file);

la sintassi è la stessa di quella per la lettura da memoria di massa, vengono chiesti num blocchi di lunghezza len e il risultato è il numero di bytes realmente ricevuti.

Esempio:
tento un collegamento UDP con il server all' indirizzo ip 192.168.1.33 sulla porta 1555.
Se il collegamento è accettato invio 10 bytes , attendo un po', leggo l' eventuale risposta e chiudo il collegamento.

```
char txmsg[10]={1,2,3,4,5,6,7,8,9,10};
```

```
char rxmsg[10];
int inviati,ricevuti;

FILE *f = fopen("TCP:192.168.1.133" , 1555);
if (f)
{
   inviati=fwrite(txmsg, 1, 10, f);
   wait(100);
   ricevuti=fread(rxmsg, 1, 10, f);
   fclose(f);
}
```

La gestione di un collegamento TCP è identica a quella di un collegamento UDP.
La sola differenza è che un collegamento TCP garantisce che nessun dato sarà perso.

TCP server

Dichiarazione del task del server:

FILE *f= fopen(char *port,int task);

dove port è una stringa ascii del tipo **TCP-SERVER:n** dove n è il numero della porta del server e task è il nome del task che gestirà il colloquio con il client quando il client aprirà la connessione

Il task del server:

void mytask1(FILE *fx)

fx è il task aperto per la connessione al client che vuole connettersi.
Da non confondersi con il file f eventualmente ma non necessariamente dichiarato in fopen() che è unico e gestito internamente, mentre i files fx possono essere molteplici se più client vogliono connettersi contemporaneamente sulla stessa porta. È stato impostato un limite di 50 per il numero di clienti contemporanei sulla stessa porta, oltre il quale il server risulta occupato.

Esempio:
In questo esempio il server accetta connessioni sulla porta 502 e **per ogni client apre una nuova istanza** del task mytask1. All' inizio invia un messaggio "hello!" al client e per ogni messaggio ricevuto risponde al client la lunghezza del messaggio stesso, e si scollega se il client gli manda il messaggio "bye!".
ATTENZIONE: Il file fz dichiarato nella open del server è il gestore del reskeduling di tutte le connessioni e chiuderlo mentre vi sono ancora connessioni in corso può portare al crash del sistema.

```
 void mytask1(FILE *fx)
{
   char buffrx[101];
   char bufftx[101];
   FILE *f=fx;
   if(f==0) return 0;                    //connessione abortita
   sprintf(bufftx,"hello!",n);
   fwrite(bufftx,1,strlen(bufftx)+1,f);
   while(n=fread(buffrx, 1, 100, f)>0)   //legge dati arrivati in modo blocking
   {                                     //se < o = 0 significa che la connessione è caduta
      sprintf(bufftx,"received  %d bytes",n);
      fwrite(bufftx,1,strlen(bufftx)+1,f);
      if (strstr(buffrx,"bye!") break;
   }
   fclose(f);
}

…
FILE *fz = fopen("TCP-SERVER:502",(int)mytask1);
```